Spicy: A Unified Deep Packet Inspection Framework Dissecting All Your Data

Robin Sommer[§*], Johanna Amann[§], and Seth Hall[§]

TR-15-004

November 2015

Abstract

Deep packet inspection systems (DPI) process wire format network data from untrusted sources, collecting semantic information from a variety of protocols and file formats as they work their way upwards through the network stack. However, implementing corresponding dissectors for the potpourri of formats that today's networks carry, remains time-consuming and cumbersome, and also poses fundamental security challenges. We introduce a novel framework, Spicy, for dissecting wire format data that consists of (i) a format specification language that tightly integrates syntax and semantics; (ii) a compiler toolchain that generates efficient and robust native dissector code from these specifications just-in-time; and (iii) an extensive API for DPI applications to drive the process and leverage results. Furthermore, Spicy can reverse the process as well, assembling wire format from the high-level specifications. We pursue a number of case studies that show-case dissectors for network protocols and file formats – individually, as well as chained into a dynamic stack that processes raw packets up to application-layer content. We also demonstrate a number of example host applications, from a generic driver program to integration into Wireshark and Bro. Overall, this work provides a new capability for developing powerful, robust, and reusable dissectors for DPI applications. We publish Spicy as open-source under BSD license.

§ International Computer Science Institute, 1947 Center Street, Suite 600, Berkeley, California, 94704
* Lawrence Berkeley National Laboratory, 1 Cyclotron Rd, Berkeley, CA 94720

This work was partially supported by funding provided to ICSI through National Science Foundation grants CNS-0831535, CNS-0915667, and CNS-1228792. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors or originators and do not necessarily reflect the views of the National Science Foundation.

1 Introduction

Deep packet inspection systems—firewalls, intrusion detection systems, inline virus scanners and proxies—process wire format network data from untrusted sources. As they work their way from raw packets upwards through the network stack, they collect semantic information from a variety of protocols, regularly going far into the application-layer to extract, e.g., the bodies of HTTP sessions or attachments from emails. Increasingly, DPI systems now also proceed beyond the network level, mining file content–documents, images, executables, and archives—for high-level context. On the developer's side, however, such rich analyses means writing a large number of individual *dissectors*—i.e., format-specific parsing code—for the potpourri of protocols and file formats that today's networks carry. Implementing a dissector remains a daunting task that not only proves time-consuming and cumbersome, yet also poses fundamental security challenges when facing real-world data that—inadvertently or maliciously—regularly fails to follow standards and RFCs. As recent vulnerabilities in Wireshark [31], Suricata [28], and Bro [22], demonstrate, coming to correct and memory-safe dissection code remains a challenge.[1]

A range of past efforts have developed approaches and tools for creating more robust dissectors, often automating parts of the implementation process by generating parsing code from higher-level specifications. Yet existing work in this space tends to focus on improving only specific aspects of the task, while rarely integrating with efforts targeting different pieces. As a result, in practice, application writers still approach dissectors as they always have: writing each one manually, from scratch, in low-level code. Returning to the three examples from above, Wireshark ships with hundreds of dissectors in C; the more recent Suricata implements all its dissectors once again in C and without reusing code from similar systems; and and while Bro employs a basic parser generator, it still requires extensive custom C++ code and does not generalize to file formats.

In this work, we introduce a novel, comprehensive framework for dissecting wire format data that integrates and unifies capabilities, approaches, and lessons-learned from existing efforts as well as from our experiences developing and deploying DPI software in production. This framework, *Spicy*, consists of *(i)* a novel type-based specification language that integrates syntax and semantics into a unified processing model expressing a format's structure; *(ii)* a just-in-time compiler toolchain that, from these specifications, creates robust and efficient native code for parsing and generating wire format; and *(iii)* an extensive API for applications to drive the process and integrate its output. Spicy supports analyzing both network protocols and file formats; facilitates reuse across applications; integrates into offline and real-time applications; parses streaming data incrementally; reassembles out-of-order data transparently; dynamically decapsulates inner layers; and handles and recovers from errors. Spicy dissectors deal robustly with non-conforming input; remain thread-safe; and support run-time introspection.

[1]See CVEs 2014-4174, 2014-6603, 2014-9586, respectively.

We evaluate Spicy through a set of case studies that demonstrate dissectors for network protocols and file formats; first individually, then in combination forming a complete stack that processes PCAP traces all the way up to file content inside application-layer protocols. We also show-case a number of example DPI host applications, from a generic driver to integration into Wireshark. Finally, to demonstrate Spicy's potential beyond the DPI domain, we present an HTTP proxy server employing it.

Overall, this work provides implementators of DPI applications with a new capability for developing powerful, robust, efficient, and reusable dissectors for protocols and file formats. We are releasing our Spicy prototype implementation as open-source under a BSD license [27].

We structure the remainder of this paper as follows. §2 introduces Spicy's conceptual capabilities. §3 presents our prototype implementation, and §4 evaluates the framework through a set of case studies. §5 discusses related efforts and §6 concludes.

2 The Spicy Model

In this section, we discuss the main elements of Spicy's model, with a focus on its capabilities for expressing flexible, stateful dissectors. We discuss design objectives in §2.1, introduce the language through examples in §2.2, and then walk through its main features. Appendix A summarizes most of Spicy's language elements. In the following, we use *format* to refer to both network protocols and file formats, as Spicy generally does not distinguish between them. A *host application* is an application that integrates Spicy for dissecting or assembly (e.g., an IDS).

2.1 Objectives

At a high-level, Spicy's design targets the following objectives to cater to complex host applications:

Declarative, high-level model. Spicy employs a type-based style that expresses elements at the semantic level of typical format specifications (e.g., RFCs).

Unification of syntax and semantics. Spicy expresses syntax and semantics inside a single language, requiring external code only for interfacing with a host application.

Support for composition. Spicy caters to layering by supporting processing pipelines through composition.

```
# cat smtp.spicy
module SMTP;

export type Greeting = unit {
             : /220 /;
    domain   : /[^\r\n ]*/;
             : / ?/;
    protocol: /(E?SMTP)?/;
             : / ?/;
    software: /[^\r\n]*/;

    on %done { print self; }
};

# echo "220 mx.foo.com ESMTP Postfix" \
  | spicy-driver smtp.spicy
<domain=mx.foo.com, protocol=ESMTP,
software=Postfix>
```

Figure 1: Dissecting an SMTP greeting.

Format-agnostic. Spicy's model captures both network protocols and file formats, binary and ASCII.

Robustness. Execution remains safe when facing unexpected input, and supports error handling and recovery.

Extension and reuse. Spicy allows customization and integration without modifying existing code.

2.2 Examples

Figure 1 shows a simple Spicy example that dissects greeting banners of SMTP connections. Generally, a Spicy dissector comes in the form of a *module* that defines one or more *unit* types. A unit represents a semantic entity of the target format (e.g., a PDU). Units consist of *attributes* to parse from wire format. The SMTP example defines a single unit type, `SMTP::Greeting`, that specifies the protocol's greeting structure as a sequence of regular expressions, assigning names to attributes for later access (here, `domain`, `protocol`, and `software`; the others remain anonymous). The unit also defines a *hook* that, at runtime, will execute custom code during the dissection process. In this case, the `%done` hook will run once dissecting has completed an instance of the unit, with `self` providing access to that instance inside the hook. Figure 1 also shows `spicy-driver`, a generic driver program bundled with the Spicy implementation that compiles a module just-in-time into executable code and then passes its standard input to the dissector code. The output represents the hook's `print` statement, confirming that the dissector picks out the named attributes.

Figure 2 demonstrates a more complex example: fully functional Spicy code for dissecting *tar* files. The `tar` module defines three units, with the `export` keyword marking `Archive` as the top-level entry point. The module follows tar's structure: An `Archive` consists of a list of files terminated by a null byte plus padding. Each file in turn consists of a `Header` followed by its binary content (`data`). The `Header` contains meta information, such as file name and modification time.

The `tar` module demonstrates a number of Spicy's key features. We see how the language expresses syntax and semantics inside a unified computation model, tying them together through the unit's attributes. For example, for attributes of type `bytes`—which represent raw binary data—one can specify their length through a Spicy expression accessing the current dissector state (see, e.g., ❶). Likewise, `&convert` post-processes a just parsed attribute using a provided expression; it stores the result rather than the original value (e.g., ❷ strips out trailing padding before storing the value; ❸, in addition, turns the file size from its octal ASCII-representation into an unsigned integer value). Inside the `&convert` expression, `$$` references the parsed value. Hooks can further normalize attributes (❹), and also derive additional unit-wide instance variables (i.e., synthesized attributes; ❺). Generally, the Spicy language provides the elements of typical scripting languages, yet extends standard constructs with further domain-specific features. For example, the `enum` type safely converts from integers (❼), along with boolean access returning true if matching a known identifier (❹).

The `tar` module also demonstrates a unit deploying look-ahead parsing: the `Archive` unit expresses the archive's content as a list of files (❽), yet without any *explicit* condition that would specify when to terminate the list (e.g., a list length). However, Spicy recognizes that a null byte (❽) must follow, making that the terminator.

In the following, we discuss the main capabilities of Spicy's language and processing model in more detail. Our discussion here focuses on conceptual properties that the framework exposes to users writing Spicy modules. We discuss implementation aspects subsequently in §3.

2.3 Dissecting Data

As the previous examples show, Spicy structures modules around units to dissect as input arrives. Unit attributes support a variety of types for dissecting, including atomic types, containers, and other units. Attributes often support further annotations that define specifics of the corresponding wire format, such as byte order or the length of lists.

In terms of atomic types we introduced the `bytes` type §2.2, which holds raw data; others include strings, signed/unsigned integers, IP addresses transparently handling v4 and v6, and bitsets to address sub-byte values. As seen in Figure 2, the `&convert` property converts a value according to a given expression; the attribute's type changes to that of the expression, not the one used for parsing the wire format. When specifying an attribute as a constant value (including regular expressions), Spicy expects a corresponding match at the current position inside the input stream. Dissection can also fill containers (lists, vectors) with other types, as well as recurse into sub-units.

```
# cat tar.spicy
module tar;

export type Archive = unit {
 files: list<File>; # ❽
      : uint<8>(0x0); # Null byte ❾
      : bytes &length=511;
};

type File = unit {
 header: Header;
 data  : bytes &length=self.header.size; # ❶
       : bytes &length=512-(self.header.size mod 512)
};

type Type = enum {
 REG=0, LNK=1, SYM=2, CHR=3, BLK=4, DIR=5, FIFO=6
};

bytes depad(b: bytes) { # ❻
 return b.match(/^[^\x00 ]+/); # Strip trailing padding
}

type Header = unit {
 name   : bytes &length=100 &convert=depad($$); # ❷
 mode   : bytes &length=8   &convert=depad($$);
 uid    : bytes &length=8   &convert=depad($$);
 gid    : bytes &length=8   &convert=depad($$);
 size   : bytes &length=12  &convert=depad($$).to_uint(8);#❸
 mtime  : bytes &length=12  &convert=depad($$).to_time(8);
 chksum: bytes &length=8    &convert=depad($$).to_uint(8);
 tflag  : bytes &length=1   &convert=Type($$.to_uint()); # ❼
 lname  : bytes &length=100 &convert=depad($$);
        : bytes &length=88; # Skip fields for brevity.
 prefix: bytes &length=155 &convert=depad($$);
        : bytes &length=12; # Padding.

 var full_path: bytes;

 on %done {
  if ( ! self.tflag ) # Default to REG if unknown. ❹
   self.tflag = Type::REG;

  self.full_path = self.prefix + b"/" + self.name; #❺
 }
};
```

```
module PrintTar;

import tar;

on tar::Archive::%done {
 print self.files;
}

# tar tvf mp.tar
foobar/staff       0 2015-08-15 18:58 mp/
foobar/staff   39548 2015-08-15 18:58 mp/part01.txt
foobar/staff   39503 2015-08-15 18:58 mp/part02.txt

# cat mp.tar | spicy-driver tar.spicy print-tar.spicy
[<header=<name=b"mp/", mode=b"000755", uid=b"000771",
  gid=b"000024", size=0, mtime=2015-08-16T02:58:19Z,
  chksum=5100, tflag=DIR>, data=b"">,

 <header=<name=b"mp/part01.txt", mode=b"000644",
 uid=b"000771", gid=b"000024", size=39548,
 mtime=2015-08-16T02:58:19Z, chksum=6351, tflag=REG>,
 data=b"A seashore. Some way out to sea [...]"

 <header=<name=b"mp/part02.txt", mode=b"000644",
 uid=b"000771", gid=b"000024", size=39503,
 mtime=2015-08-16T02:58:11Z, chksum=6348, tflag=REG>,
 data=b"A man appears on the top of a sand [...]’]
```

Figure 2: Dissecting a tar file. Fully functional example. Output slightly edited for clarity.

```
# cat switch.spicy
module Test;

type A = unit { xa: b"a"; ya: bytes &length=1; };
type B = unit { xb: b"b"; yb: bytes &length=2; };
type C = unit { xc: b"c"; yc: bytes &length=3; };

export type Foo = unit {
  switch { a: A; b: B; c: C; };
};

on Foo::%done { print self; }

# echo "b12" | spicy-driver switch.spicy
<b=<xb=b"b", yb=b"12">>
```

Figure 3: Examples of switch statement with look-ahead.

By default, the dissector will proceed through attributes in order. However, Spicy offers constructs that alter the flow, including marking attributes as optional with a conditional expression, and a *switch/case* construct to branch accordingin to either an expression or the token next in the input stream (see Figure 3). The dissector can also branch tentatively, and backtrack later if necessary.

Typically, a dissector will proceed sequentially through its input. However, a unit can instead provide a custom `bytes` value to an attribute to parse instead (e.g., to first strip an outer encoding and then parse the result differently). Spicy also allows for random access within data passed into a unit through methods to retrieve and relocate the current input position (e.g., to parse DNS labels pointing to a previous location inside the payload).Finally, *sinks* represent a construct to dynamically interface independent units; we discuss them in §2.5.

2.4 Embedding Semantics

Spicy ties a format's semantics to its syntax, which *(i)* overcomes limitations of context-free languages by enabling conflict resolution, and *(ii)* allows accumulating state for the host application to leverage. To capture semantics, Spicy deploys two mechanisms that work in tandem: *(i) hooks* embed semantic actions into a unit that execute at well-defined times during processing; and *(ii)* unit variables persistently record derived state information (similar to "synthesized attributes" in attribute grammars [12]). We have already introduced `%done` hooks in §2.2, which execute once a unit finishes parsing. A corresponding `%init` is provided for initialization logic. Furthermore, one can associate hooks with any unit attribute, which will trigger right after they receive their values. For example, the code excerpt in Figure 4 dis-

```
type Message = unit {
  [...]
  var has_body: bool           &default=False;
  var content_length: int<64> &default=-1;
};

# Define regular expressions.
const Name  = /[^:\r\n]+/;
const WhiteSpace = /[ \t]+/;
const Value = /[^\r\n]*/;
const RestOfLine = /[^\r\n]*/;

type Header = unit(msg: Message) {
  name : Name;
       : WhiteSpace;
       : /:/;
       : WhiteSpace;
  value: Value;
       : RestOfLine;

  on value {
    if ( self.name.lower() == b"content-length" ) {
      #Record information in parent unit (see text)
      msg.has_body = True;
      msg.content_length = self.value.to_uint();
      }
    }
```

Figure 4: Recording state in unit variables.

sects HTTP header lines with additional processing for `Content-Length` using a hook on `value`.

Spicy provides further hook types for error handling, stream reassembly, and debugging; we discuss them below. For modularization, one can add hooks to a unit externally by qualifying the identifier accordingly (as with `PrintTar` in Figure 2). This works across module boundaries and facilitates application-specific customization without touching existing grammars. For implementing hooks, Spicy's language provides similar statements as other high-level scripting languages, including control flow constructs for loops and branching, operators overloaded by argument types, and an object-oriented model operating on types via method calls. We do not discuss these language elements in detail in this paper.

Typically, hooks inspect or modify state that dissectors accumulate. Through the `self` keyword hooks have access to all attributes available so far, as well as to the current values of unit variables. Units can access parent units higher up in the parse tree through unit parameters (similar to "inherited attributes" in attribute grammars); see, e.g., the `msg` parameter in Figure 4. Spicy also provides *global* state that persists across individual dissectors. This enables correlating information over time, for example for associating requests with replies or reassembly across input streams.

2.5 Composing Units

Many formats rely on nesting: after stripping off an outer layer, one finds more data to dissect, typically now in a different format. Network protocol stacks represent the standard example, yet many file formats exhibit similar

```
type Header = unit(msg: Message) { ... }

type Message = unit {
  headers:    list<Header(self)>;
  :           /\r?\n/; # End of header
  body:       bytes &length=self.content_length
                    -> self.data; # Pass to sink.

  on headers {
    # Activate units by their %mimetype.
    self.data.connect_mime_type(self.content_type);
  }

  var data: sink; # Sink receives the body content.

  # Variables set when parsing Headers.
  var content_type: bytes;      # MIME type.
  var content_length: uint<64>; # Body length.
};

type JPEG = unit { %mimetype="image/jpeg"; ...};
type tar = unit { %mimetype="application/x-tar"; ...};
```

Figure 5: Dissecting HTTP message bodies dynamically through sinks. (Units simplified for illustration.).

structures (e.g., tar archives contain more files; images may contain EXIF information). Spicy supports decapsulating such chains in two ways: statically via attributes of a corresponding sub-unit type, and dynamically through a dedicated *sink* data type. The former naturally expresses a hierarchical relationship (`File::header` in Figure 2 is an example). The latter provides a dynamic interface between independent unit instances, commonly in different modules. Conceptually, sinks operate like pipes: an outer unit writes wire format data into a sink, to which an inner unit attaches for dissecting it. Figure 5 shows a dissector for HTTP message bodies as an example, defining the sink `data` for decapsulating the content. After dissecting the HTTP message headers, the code informs the sink of the body's MIME type. Spicy's runtime then dynamically attaches the unit type that can handle that type; the two candidates in the example are `JPEG` for `image/jpeg`, and `tar` for `application/x-tar`. Subsequently, as `Message` extracts the body, it forwards the data to the sink for dissection using the `->` operator.

A unit type can declare support for one or more MIME types. If there are multiple unit types associated with the same MIME type, the sink will attach them *all*, each receiving a copy of the data (we use this mechanism for content identification; see §2.7). Even though we use the term "MIME type" here, we extend its traditional file-centric notion to a more general naming scheme for dissectors by introducing Spicy-specific top-level media types. For example, for application-layer protocols operating on top of TCP, we define types of the form `tcp/<port>`, where `<port>` represent a protocol's well-known port (e.g., the HTTP dissector declares `%mimetype = "tcp/80"`). With that convention, the TCP dissector can use a corresponding `connect_mime_type("tcp/<port>")` call to dynamically dispatch processing by port.

In addition to this dynamic dispatching, sinks also can be statically connected to a specific target unit. This can, e.g., be useful when decapsulating layers in a single protocol like necessary for record fragments in TLS.

For further flexibility, Spicy extends the sink model with three additional capabilities. First, sinks can preprocess their input before passing it on. Currently, our implementation supports two such *filters*: base64 decoding and unzipping. Second, in addition to the arrow operator, there is also an explicit `write()` method offering more fine-grained control. In particular, `write()` can associate sequence numbers with the input, in which case the sink will internally put the data in order before passing it on. In §4.2 we use that for implementing TCP stream reassembly. Sinks provide a number of options for fine-tuning out-of-order processing, including defining initial sequence numbers as well as knobs and hooks to handle gaps and ambiguities. Finally, writers can insert meta-information in the form of positional *marks* into the sink's data, e.g., at PDU boundaries. This is, e.g., helpful for resynchronizing after errors by skipping ahead (see §2.6).

2.6 Error Handling and Recovery

When processing untrusted input—such as packets arriving on the network or files attached to incoming emails—robustness is key for ensuring integrity. Spicy provides a secure execution environment for catching errors, as well as recovery mechanisms to continue processing afterwards. Spicy defines a well-defined, statically type-safe environment that prevents unintended data and control flows. The language mediates all memory accesses, and the runtime garbage collects memory automatically. Spicy differentiates between two kinds of errors: *parse errors* and *logic errors*. Dissectors trigger the former when encountering input not aligning with their units. They propagate up the unit tree, giving upper levels a chance to handle them; if unhandled, they become fatal. Logic errors trigger on coding errors, such as an out-of-bounds array access or a division by zero. Logic errors abort immediately, passing control back to the host application.

Units can handle parse errors in one of two ways. First, they can define explicit `%error` hooks to execute, for example to set an internal flag or clear state. Second, a dissector can define a recovery strategy that resynchronizes processing at a subsequent point within the input stream. For that, it needs to add a pair of annotations. First, one unit defines a strategy to skip ahead in the input stream to a position where it could resume dissecting after an error. Consider HTTP as an example: if, e.g., a gap occurs due to packet-loss inside a pipelined HTTP session, a reasonable strategy might be jumping ahead to the beginning of the next HTTP request. Figure 6 implements this in `RequestLine`, which defines a regular expression to

```
type Requests = unit {
    requests: list<Request> &synchronize;
};

type Request = unit {
    request: RequestLine;
    message: Message;
};

type RequestLine = unit {
    %synchronize-at = /(GET|POST|HEAD) /;
    method:  Token;
    :        WhiteSpace;
    uri:     Token;
    :        WhiteSpace;
    :        /HTTP\//;
    version: /[0-9]+\.[0-9]*/;
    :        NewLine;
};

type Message = unit { ... }
```

Figure 6: Resynchronizing at the next HTTP request.

scan for in that case through the `%synchronize-at` property. Second, a higher-level unit annotates an attribute with `&synchronize` to allow that to initiate the process; in the example, that is `Requests::requests`. With these two in place, when an error occurs while dissecting a `Request`, Spicy will propagate it up to `Requests::requests`. There, it will determine the *next* attribute that would normally get parsed; here, another `Request`. Examining the parse tree down from there, Spicy will find that `RequestLine` supports resynchronization. Accordingly, it will begin looking for its regular expression, skipping ahead in the input to the first location that matches. Once found, normal processing will resume by dissecting a `Request`.

In addition to using regular expressions, Spicy's recovery mechanism also supports scanning for other resynchronization hints, including constants and meta information embedded into the input. For example, a better resynchronization strategy for HTTP would be skipping ahead to the next request that also aligns with the beginning of a TCP packet; this avoids false positives when an HTTP body happens to contain content resembling a request (the canonical example is downloading HTTP RFCs). To implement that, the TCP dissector would insert marks into the input stream corresponding to packet boundaries for the HTTP dissector to skip ahead to.

2.7 Dynamic Format Detection

Spicy can identify the appropriate dissector for an input stream by inspecting the content. While traditionally, DPI applications used to rely on well-known ports for choosing an application-layer dissector ("if it's on port 80, it must be HTTP"), a significant portion of traffic now uses non-standard ports—sometimes deliberately to evade monitoring systems. File formats face a similar challenge: as file extensions can be misleading, some applications resort to content inspection for identifying types (via, e.g., `libmagic`; or "content sniffing" in web browsers).

Spicy's identification approach generalizes Bro's *Dynamic Protocol Detection (DPD)* [5], which follows a simple idea: given that the application already has dissectors for the relevant formats, have them all *try* to parse the input; the one that succeeds will have implicitly identified the format, whereas the others will likely bail out quickly. We integrate this scheme into Spicy by providing it with a variant of the `connect_mime_type` sink method (see §2.5). Like the original version, the variant attaches all matching dissectors. However, it initially runs them in a *trial mode*, in which a parse error does not trigger normal error handling but instead silently disables the dissector. Units can exit trial mode by calling a *confirm()* method once they deem it reasonably certain that they indeed "own" the input; usually that will be after parsing a few attributes successfully. For further customization, Spicy also offers two hooks that trigger on confirmation and when a dissector gets disabled, respectively.

We add another optimization phase to this process. As fully activating all potential dissectors can turn out expensive, units may first quickly scan the input to see whether it resembles their expectation. We reuse Spicy's resynchronization support here (see §2.6): In trial mode, if a unit that attaches to a sink also defines a resynchronization strategy, it will begin its processing in resynchronization mode. If it finds the corresponding hints, it will switch to the standard dissector—just as during error recovery, except that it will remain in trial mode. If resynchronization fails to find its hints, the unit will silently disable itself. [2]This approach offers with an additional novel capability: By starting out in resynchronization mode, dissectors can start processing input midstream even when a host application has missed the beginning. This solves a common problem of DPI systems, which typically cannot analyze long-lived connections already active at start up.

2.8 Assembling Wire Format

In addition to dissecting data, Spicy also supports the opposite way: *assembling* wire format data according to a unit's layout. To use that, a host application can either pass in unit instances with attributes appropriately prefilled, or start with data dissected previously. The latter enables semantic changes to input in wire format, such as anonymizing user names and passwords inside a connection's payload, or replacing individual values with variations for fuzzying or regression testing. In the spirit of Pang et al. [19], Spicy leverages the observation that productions in context-free grammars can not only drive a parser, yet also a generator for the language they define. However, different from [19], Spicy's unification of syntax and semantics allows supporting any Spicy module, whereas the former requires significant additional low-level, per-format customization work.

For simple cases, Spicy can reverse the dissection process directly. Consider the SMTP example in Figure 1: to assemble the banner, one can concatenate the attributes in the order that parsing takes them apart. To demonstrate this, we extended the `spicy-driver` program so that after dissecting input, it proceeds to put it back together. Executing this with the example input in Figure 1 indeed returns the original input back. Furthermore, if we add a statement `self.version = b"Spicy"` to the `%done` hook—which executes after dissection, yet before assembly—the output changes accordingly to `220 mx.example.com ESMTP Spicy`.

Assembly works automatically for units that do not rely on semantic expressions or actions for control flow or attribute access. The `tar` module in Figure 2 is close to that, however the `&convert` annotations prove problematic because the unit stores their results, not the values that the dissector extracted originally. Furthermore, if one modifies the derived value of `Header::full_path`, the change will not carry over to the fields that contribute to it, `prefix` and `name` (❺ in Figure 2). For such cases, Spicy allows to augment a unit with help for the assembly process. First, one can annotate hooks and fields for consideration only during dissecting *or* assembling, allowing to fine-tune the two use cases. This solves the `full_path` problem, as one can add an assembly-only hook that breaks the value back down, writing into `self.prefix` and `self.name`. Second, for `&convert` attributes, one can add a corresponding `&convert_back` expression for reversing the computation. In `tar::Header`, this could pad `Header::name` could back to 100 bytes. In some cases, Spicy can indeed derive the reversal automatically. For example, our implementation reverses `enum` conversions (e.g., ❼ in Figure 2).

2.9 Debugging Support

Even with a high-level language, writing dissectors can remain challenging if a format is complex or ill-documented. Spicy supports the development process with debugging facilities that it can optionally compile into dissectors it generates. First, one can mark hooks as pertaining to debugging, in which case dissectors will only include them in development builds, yet omit them from production code. Debug hooks can, e.g., print additional output for tracing state updates, or perform consistency checks helping with development. Second, during development builds dissectors also include additional code to log their

[2]For more efficient matching, Bro compiles all its regular expressions into a single finite automata; an optimization that we plan to add to Spicy, too. Spicy's scheme is, however, more general than the corresponding phase in Bro, which hardcodes scanning for regular expressions.

progress at runtime, recording names and values of individual attributes as they parse or assemble them. This provides visibility into the process and makes it easy to pinpoint locations where input or output deviates unexpectedly. A more detailed version of this output can also report internals of the dissection and assembly processes, such as current productions and look-aheads (see §3).

3 Implementation

We now discuss our implementation of Spicy. While still a prototype, the current system provides all functionality discussed in this paper. We focus on the main conceptual parts of the implementation: the Spicy architecture, the backend for emitting machine code, generation of dissectors, and the C API for host applications.

3.1 Objectives

For the implementation we target a set of system-level objectives to ensure that Spicy can indeed support real-world host applications:

Robustness. Spicy-generated dissection code remains robust against unexpected, potentially malformed, input.

Stream processing. Spicy's dissectors process input incrementally, without buffering beyond tokens.

Just-in-time compilation. The Spicy toolchain compiles dissectors on the fly as a host application starts up.

Thread-safety. Dissectors are reentrant and thread-safe, and hence support concurrent processing.

Integration. Host applications integrate and control Spicy dissectors through a comprehensive C interface.

Efficiency. Spicy targets high-volume settings, with performance similar to manually written dissectors.

3.2 Backend

Spicy leverages our HILTI abstract machine [26] as its backend for code generation. HILTI provides an abstract machine model with instruction set and data types tailored to the networking domain; as well as a compiler toolchain, built on top of LLVM [29], that turns HILTI programs into optimized native code. HILTI fits Spicy well, as it provides much of the low-level functionality the system needs. We achieve several of our objectives through HILTI. First, its safe, memory-managed execution environment provides the robustness for Spicy's dissectors; by expressing their logic inside HILTI's model, they benefit from its runtime guarantees. Second, HILTI supports just-in-time compilation through LLVM. Third, HILTI's execution model comes with concurrency support, is thread-safe by design, and generated code with a performance generally aligning with hand-written code [26].

We already used Spicy in [26] as one of the example applications for HILTI (then still called *BinPAC++*). Spicy itself was, however, not part of the paper's contributions—we utilized it only as a use-case to verify viability, correctness, and performance of the HILTI abstract machine.

3.3 Architecture

Internally, the Spicy implementation consists of two main components. The *code generator* compiles Spicy modules into intermediary HILTI code; it consists of about 30,000 lines of C++ code. The *runtime library* provides static functionality to the generated dissectors, as well as the API to host application and consists of about 2,500 lines of C code. Applications like, e.g., `spicy-driver` from Figure 1 integrate both these components: Applications *(i)* employ the compiler to translate Spicy modules to HILTI; *(ii)* use HILTI's toolchain to generate machine code; *(iii)* link that against the runtime library; *(iv)* instantiate a dissector; and *(v)* pass input into it.

3.4 Code Generation

Spicy's code generator constitutes the core of the implementation. While a large part of it constitutes an application of standard parsing technology, Spicy's approach is unique in that its dissectors represent a hybrid of two traditional schemes. Traditionally, parsers in the networking domain tend to operate without an explicit lexer phase. While that approach is efficient and of low complexity, it puts more burden on the programmer to ensure deterministic parsing in ambiguous situations. Recall the list `Archive::files` in Figure 2. Normally, a dissector would express the look-ahead for the end marker explicitly in its code, whereas a lexer-based parser could instead predict the correct production itself.

Spicy combines the two approaches—with and without lexer—by integrating a lexer into the generated parsers, yet only at places that need to choose between productions. Spicy's parser generation proceeds in two steps: Spicy first derives a context-free grammar (CFG) for each top-level unit. For each grammar, it then generates a non-backtracking recursive-descent LL(1) parser, in the form of HILTI code closely following the production structure. Figure 7 shows the CFG for a simple example unit dissecting a sequence of strings `foo`, terminated by a string `bar` followed by a final 64-bit integer. At runtime, the dissector code extracts tokens as needed. For example, when dissecting `_Foo_foo`, the dissector matches the regular expression `[Ff]oo` against the current input position, without prior tokenization as there is no alternative choice. Likewise, when in `_Test_count`, the dissector directly extracts the next four bytes and interprets them as an integer value. The more interesting case

```
type Foo = unit { foo: /[Ff]oo/; };

type Test = unit {
    foos : list<Foo>;
    bar  : /[Bb]ar/;
    count: uint<64>;
    };

Grammar:

  _Test_foos   → _Test_foos_1
  _Test_foos_1 → _Foo _Test_foos_2
  _Test_foos_2 → _Test_foos_1 | ε
  _Test_bar    → /[Bb]ar/
  _Test_count  → <uint<64>>
  _Foo         → _Foo_foo
  _Foo_foo     → /[Ff]oo/
```

Figure 7: A unit as a left-factored right-recursive grammar. Non-terminals with underscore; without otherwise.

is `_Test_foos_2`, where the dissector needs to decide which alternative to follow. To that end, Spicy determines at compile time the first terminal in each alternative's derivation.[3] In the example, there is only one for the first case: on `[Ff]oo`, the dissector follows `_Test_foos_1`. Hence, it will match this expression at the current position and either proceed accordingly if found, or end the production. (The look-ahead sets must be recognizable tokens; if, e.g., we removed `bar` from the unit, the lexer would need to identity a `uint<64>` for making the decision; as one cannot make that decision by just looking at bytes without further context, Spicy would reject the unit.)

Spicy's dissectors operate fully incrementally so that host applications can pass them input in arbitrary chunks. The dissector will process the data as far as possible, advance its internal state accordingly, and return back to the host application for more input. Normally, it will buffer input only at the token level (e.g., if dissecting a 64-bit integer, it will buffer up to 4 bytes). While even this could be avoided, that would increase complexity of the generated code without much benefit. The one case where the dissector needs to buffer larger amounts of input is random access: if a unit uses Spicy features that may need to seek back before the current position, that data must remain available (an example is dissecting DNS labels with pointers referring backwards). Internally, Spicy dissectors achieve incremental processing by maintaining a sliding window over the input stream. At any time, two HILTI iterators mark the current start and end positions, with the former moving ahead as the dissector processes bytes, and the latter advancing as the host application adds further data. HILTI's garbage collection releases the memory as soon as it becomes unreachable.

[3]We deploy the standard methodology for LL(1) parsing here (i.e., compute FIRST_1 and FOLLOW tables). See, e.g., [9] for details.

3.5 Host interface

We provide Spicy with a flexible API that enables host applications to compile dissectors, pass in data for processing, and access the results. An application can perform the compilation step either offline ahead of time, using the Spicy toolchain; or just-in-time at startup using Spicy's C++ API. For each dissector, the host application receives a pointer to a C function that initiates parsing. For incremental processing, a second C function handles resuming once a new chunk becomes available. If assembly is requested, a third function provides that.

There are two approaches for the host application to access a dissector's results. Most directly, the dissector's parse function returns a pointer to a C structure with attributes corresponding to the unit definition, including a suitable C prototype of the *struct* type. In that structure, the Spicy type of each attribute maps to a corresponding C type, using either C equivalents for simple atomic types (e.g, a Spicy `int<64>` maps to a C `int64_t`), or C-level representations of Spicy's higher-level types, along with corresponding accessors functions through the runtime library (e.g., there's a C-level list type corresponding to Spicy's lists, with functions to iterate over the elements). Spicy reuses most of these C data types from HILTI's runtime, extending them where no direct equivalent exists (e.g., for sinks). It also augments HILTI types with further runtime information to capture specific semantics (e.g., HILTI does not differentiate between signed and unsigned integers, yet Spicy does).

Spicy's hooks provide the second approach for accessing dissector results. As hooks can call out to external functions, a host application can provide its own C runtime library with custom callbacks. It would then add corresponding hooks to the unit that trigger these callbacks. To help tie back a callback's execution to the host application's state, the top-level C dissector function accepts a user-supplied "cookie" parameter that it will transparently pass through the dissector to all custom C functions. The callback approach is most suitable for processing streaming data, as it can pass on information continuously, vs. just at the end, by triggering callbacks as soon as the information becomes available, vs. making the results available only at the end of the dissection process.

The Spicy runtime library provides comprehensive introspection capabilities. While some applications may hardcode the use of specific dissectors, Spicy becomes most powerful with generic host applications operating on arbitrary Spicy modules. Indeed, most of our examples in §4 are of that type. In that case the application must learn at runtime which dissectors it has available, how to use them, and how to interpret their results. The Spicy runtime provides access to a dissector registry that reports

Protocols	BACnet (ASHRAE/ANSI 135), DNS, Ethernet (IEEE 802.3), PCAP, HTTP, IPv4, RTMP (handshake), SSH (banner), SMTP (greeting), SMB2, TCP, TLS, TFTP, UDP
File Formats	ASF (headers & metadata), ASN.1, Gzip (header), ZIP (header), MS Certificate Store, Tar, X.509 certificates

Table 1: Dissectors we have implemented in Spicy.

all available units along with further meta information included in their definition, such as a textual description, MIME types to associate with them, or well-known ports.

4 Evaluation

As Spicy aims to support a variety of formats inside real-world applications, we consider its primary contribution the combination of a comprehensive language model with a flexible integration mechanism. Consequently, we evaluate the system with an emphasis on the formats and applications it can support. In addition, our implementation comes with a test suite of almost 300 unit tests. In the following, we first discuss experiences writing a set of standalone dissectors for Spicy in §4.1. In §4.2 we chain dissectors into a complete stack and §4.3 demonstrates assembling wire format by anonymizing DNS traffic. In §4.4, we present a set of host applications integrating Spicy, including Wireshark, Bro, and an HTTP proxy.

4.1 Dissectors

We examine Spicy's support for different types of formats by developing dissectors for a diverse set of network protocols and file formats. Table 1 summarizes the dissectors we have implemented; we make their Spicy source code available online as part of the Spicy distribution [21]. As we were developing them, we identified a number of common patterns where Spicy proves particularly helpful. To begin with, the language's rich types enable concise and robust code. For example, turning integers into enumeration values via `&convert` allows later `switch` statements to branch via descriptive labels, while automatically catching unexpected values. More generally, `&convert`'s postprocessing simplifies a range of cases, from turning ASCII-encoded numbers into actual integer values (e.g., file sizes in tar) to rendering timestamps human-readable (e.g., converting Gzip's UNIX times into Spicy's time type). Bitfields provide access to arbitrary bit ranges inside integer values, which is helpful for many binary formats, such as with DNS' flags. All of Spicy's types handle byte order transparently and allow for switching on the fly if necessary (e.g., PCAP specifies byte order in its header). Spicy's ability to feed data already processed back into the dissector for other fields and units, proves powerful for expressing complex relationships (e.g., ASN.1 uses variable length byte strings with different interpretations depending on a preceding tag; and DNS' label pointers refer back to earlier data).

Spicy's support for state management provides a second common building block. At the attribute level, Spicy's expressions enable custom computations to drive decisions and results (e.g., the HTTP dissector derives from a previous header the character sequence that represents a MIME multipart boundary; RTMP computes the length of fields dynamically; TLS disables inspection of record-layer payload once encryption activates). At a higher level, Spicy enables a module to maintain state over the lifetime of a session. A common example is defragmenting PDUs—a task that many protocols require, yet traditionally remains a challenge for implementors. Historically, TLS client and server software in particular comes with a track record of defective record layer defragmentation (e.g., [16, 17, 6]). Our Spicy dissector implements that robustly within just 41 lines of Spicy code, out of of a total of 404 for TLS. The BACnet dissector deploys a similar desegmentation scheme.

Finally, Spicy facilitates reuse. While we currently do not share significant functionality between our dissectors—a result of striving for diversity—it will be possible to import them into other formats in the future (e.g., dissectors needing ASN.1 can import that module).

Overall, we find that Spicy significantly simplifies the process of implementing dissectors. We were able to implement the Spicy X.509 dissector in about 330 lines of code total, split between 145 lines for dissecting certificates and 180 for ASN.1. For comparison, Bro's corresponding code requires 550 lines even though it outsources the actual certificate parsing to OpenSSL (i.e., 550 lines just for using their API to obtain the information). As another example, we originally wrote the Spicy dissector for the Microsoft Certificate Store format [15] out of an actual need when we were unable to find an external tool parsing it. Using Spicy it took us about 15 minutes to extract the certificates to disk.

4.2 Composition

Next, we use Spicy for dissecting complete protocol stacks. For this, we implemented a dynamic stack of dissectors that parses network captures starting from PCAP traces up to files contained in application-layer protocols. We implemented this stack fully inside Spicy, using dissectors from Table 1. Figure 8 visualizes the data flows that we support for this case study.

The setup deploys the two composition methods we discuss in §2.5. First, we choose to statically interface the dissectors for PCAP, Ethernet, IP, and TCP/UDP. A

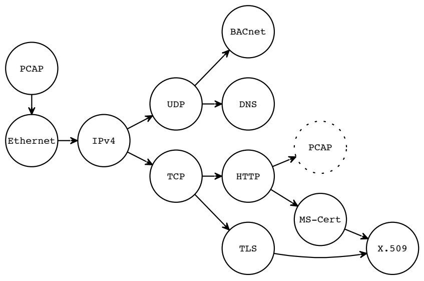


Figure 8: Spicy protocol stack.

packet trace first enters the PCAP dissector, which splits into three units. The first two parse the global PCAP file header, including identifying its byte order; the third dissects individual packets. For each packet, a `switch` statement branches to link-layer dissectors according to the data link type (only Ethernet here). The Ethernet dissector then branches again based on the EtherType (only IPv4 here). The IPv4 dissector repeats this process once more, passing the IPv4 payload on to either UDP or TCP, depending on the protocol field.

The TCP dissector performs TCP stream reassembly. It maintains a global map associating connection 4-tuples with a sink for that session, initializing the sink's initial sequence number from the TCP header. The dissector then writes the payload of subsequent packets to the corresponding sink, using their respective sequence numbers. That way, if packets arrive out-of-order, Spicy's runtime will automatically reorder their payloads.

For the application-layer, we deploy dynamic composition via sinks, as that allows extending dissectors independently and also enables dynamic format detection (see §2.7). For this case study, our stack remains port-based, though, using the generalized MIME type notion from §2.7 for choosing dissectors (e.g., `udp/53` for DNS).

For our case study, we experimented with composing several of the application-layer dissectors from Table 1. For UDP, we added both DNS as well as BACnet to the stack and verified that they work correctly. For TCP, we added HTTP and TLS to the stack, both of which further decapsulate content through their own internal sinks. The HTTP dissector extracts message bodies, passing their content on to file dissectors based on the MIME type that the HTTP message specifies. To test this, we added the dissectors for the Microsoft Certificate Store and X.509 certificates to the stack. We recorded a packet trace of a web session downloading a copy of the Windows XP certificate root store and verified that the combined PCAP/Ethernet/IP/TCP/HTTP/MSCS/X.509 stack correctly extracted the certificates. As a final example, we looked at TLS, which passes server certificates on to X.509. We created a PCAP file containing an HTTP connection that, in turn, contains a download of another PCAP file containing a TLS connection. We could indeed parse the certificate information out of the inner PCAP, demonstrating Spicy's flexibility.

4.3 Anonymization

In §2.8 we discuss Spicy's support for *assembly*: turning dissection around to produce wire format from unit specification. To test this, we extended our DNS module with support for anonymizing IP addresses and labels inside resource records. We choose DNS as an example of a common binary protocol with some particular idiosyncrasies. Starting with the DNS module originally developed only for dissecting, we identified two locations where Spicy required hints for reversing the process. First, when dissecting lists of labels, the original code did not store the final null byte, hence omitting it during assembly; we changed the unit to maintain it. Second, as the DNS format overloads a label's first two bytes to signal compression, we added an assembly-only attribute to adjust output for that distinction. In total, we added or changed 4 lines out of the module's 127 lines of code.

With that in place, we can proceed with anonymization. We added two new library functions to Spicy: `sha256` hashes a `bytes` value, and `anonymize_addr` permutates IP addresses; both take seeds for randomization. Then we defined two short hooks:

```
on DNS::Label::label {
 # Hash label in RR into bytes value of same length.
 self.label = sha256(self.label, b"seed", |self.label|);
}

on DNS::ResourceRecord::a {
 # Permutate IP address in RR with seed.
 self.a = anonymize_addr(self.a, 42);
}
```

Running this through `spicy-driver` now modifies labels and IP addresses in DNS payload on standard input.

We note that this example remains a bit simplified: by keeping the length of labels the same, we ensure that the DNS pointer structure does not change. However, with more assembly-specific logic, Spicy could adjust the pointers to lift that constraint.

4.4 Host Applications

We next turn to Spicy's support for host applications, examining as case-studies a tcpdump-like tool, integration into Bro and Wireshark, and an HTTP proxy.

spicy-dump. We wrote a simple tool, `spicy-dump`, that prints out a format's attributes in human-readable format—similar to `tcpdump` and `tshark` for network traffic. However, unlike these tools, `spicy-dump` does not hardcode any formats, yet operates generically with any Spicy module. The following excerpts shows the Spicy SMB2 dissector on a `WRITE_REQUEST` command:

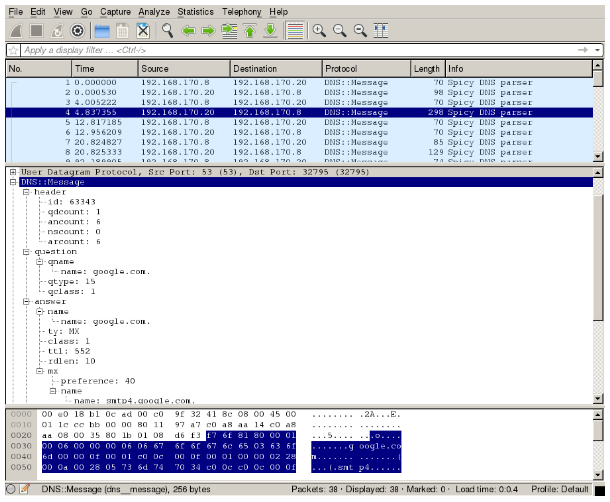
```
# cat writerequest-payload.dat | pac-dump smb2.spicy
<header=<protocol=b"\xfeSMB", head_length=64, [...],
  status=(Severity=Success, Customer=0, [...]),
  command=WRITE, flags=(ResponseToRedir=0, [...])
  [...]>

message=<request=<write=<structure_size=49,
  data_offset=112, data_len=512, file_id=<
    persistent=b"\xfd\x00\x00\x00\x00\x00\x00\x00",
    volatile=b"\x99\x00\x00\x00\xff\xff\xff\xff">
  [...] >>>>
```

`spicy-dump` can also produce machine-parseable output in JSON format, turning it into a preprocessor for other applications. Internally, `spicy-dump` leverages Spicy's introspection API to identify the dissectors available, as well as the information they provide.

Bro. We have integrated Spicy into Bro, which enables the system to dynamically add new dissectors at startup by loading and compiling Spicy modules. To the Bro user, Spicy's dissectors remain transparent: They hook into Bro's event engine similar to the system's traditional protocol analyzers, sending events to Bro's scripting language for passing on information. Indeed, combining Spicy's language for specifying protocol dissectors with Bro's language for expressing higher-level analyses tasks creates a flexible platform that no longer hardcodes any major part of the traffic analysis pipeline. As [26] already discusses the Bro integration of Spicy parsers through HILTI, we omit further details here.

Wireshark. We have integrated Spicy into Wireshark by developing a proof-of-concept Wireshark dissector plugin that works with any Spicy module. Figure 9 shows a screenshot of Spicy's DNS dissector operating inside Wireshark. At startup, our plugin compiles Spicy modules just-in-time, and then extracts names and attributes of all top-level units using Spicy's introspection API. Spicy dissectors can convey their well-known ports to a host application by defining a `%ports` unit property. Our Wireshark plugin registers them accordingly with the Wireshark core, so that it receives control for corresponding packets. For each packet, it executes the unit's dissector function and then iterates over the resulting attributes, adding each to the GUI's tree display. Currently, our Wireshark plugin supports UDP protocols; extending it further would just require interfacing appropriately with more of Wireshark's dissector API.

HTTP Proxy. Finally, we implemented a proof-of-concept HTTP proxy in C++ that deploys Spicy for HTTP dissection and assembly, using a stripped down version of the full HTTP Spicy module. The proxy receives HTTP client requests via its listening socket; parses them through the `HTTP::Request` dissector function that Spicy compiles; retrieves the HTTP method and path, as well as the `Host` header; adds a custom `Connection-Proxied` header to the dissected mes-


File Edit View Go Capture Analyze Statistics Telephony Help
Apply a display filter ... <Ctrl-/>

```
No.   Time         Source            Destination       Protocol       Length Info
    1 0.000000     192.168.170.8     192.168.170.20    DNS::Message       70 Spicy DNS parser
    2 0.000530     192.168.170.20    192.168.170.8     DNS::Message       98 Spicy DNS parser
    3 4.005222     192.168.170.8     192.168.170.20    DNS::Message       70 Spicy DNS parser
    4 4.837355     192.168.170.20    192.168.170.8     DNS::Message      298 Spicy DNS parser
    5 12.817185    192.168.170.8     192.168.170.20    DNS::Message       70 Spicy DNS parser
    6 12.956209    192.168.170.20    192.168.170.8     DNS::Message       70 Spicy DNS parser
    7 20.824827    192.168.170.8     192.168.170.20    DNS::Message       85 Spicy DNS parser
    8 20.825333    192.168.170.20    192.168.170.8     DNS::Message      129 Spicy DNS parser

User Datagram Protocol, Src Port: 53 (53), Dst Port: 32795 (32795)
DNS::Message
  header
    id: 63343
    qdcount: 1
    ancount: 6
    nscount: 0
    arcount: 6
  question
    qname
      name: google.com.
    qtype: 15
    qclass: 1
  answer
    name
      name: google.com.
    ty: MX
    class: 1
    ttl: 552
    rdlen: 10
    mx
      preference: 40
      name
        name: smtp4.google.com.

0000  00 e0 18 b1 0c ad 00 c0  9f 32 41 8c 08 00 45 00   ........ .2A...E.
0010  01 1c cc bb 00 00 80 11  97 a7 c0 a8 aa 14 c0 a8   ........ ........
0020  aa 08 00 35 80 1b 01 08  d6 f3 f7 6f 81 80 00 01   ...5.... ..o.....
0030  00 06 00 00 00 06 06 67  6f 6f 67 6c 65 03 63 6f   .......g oogle.co
0040  6d 00 00 0f 00 01 c0 0c  00 0f 00 01 00 00 02 28   m....... .......(
0050  00 0a 00 28 05 73 6d 74  70 34 c0 0c c0 0c 00 0f   ...(.smt p4......

DNS::Message (dns__message), 256 bytes    Packets: 38 · Displayed: 38 · Marked: 0 · Load time: 0:0.4    Profile: Default
```

Figure 9: Wireshark using Spicy dissector plugin.

sage; establishes a connection to the target server; reassembles the request (including the new header) through the `HTTP::Request` assembly function; and finally sends that out to the target server. Once the server's replies, the same process proceeds in reverse to forward the response to the client. (The proxy again dissects and assembles the reply, even though in this case it does not further modify its content.) We also added an internal cache to the proxy that records previous replies in reassembled wire format, to return for identical requests.

The HTTP proxy demonstrates how Spicy can support server applications. The main part of the proxy's implementation concerns managing inbound and outbound TCP connections. The code for protocol parsing and assembly consists only of function calls to the corresponding Spicy functions, which significantly reduces the lines of code compared to a manually written proxy.

4.5 Performance

Our earlier work on HILTI [26] already evaluates the performance of Spicy dissectors executing on top of the HILTI platform when processing large volumes of real-world HTTP and DNS traffic from a live campus environment. We found that the Spicy parsers need 1.28 and 3.03 times more CPU cycles, respectively, when processing about 340K/65M HTTP/DNS request-reply pairs. For HTTP, that means Spicy already comes close to the performance of manually written C++ code. The main cause for the significant slowdown in DNS is more frequent instatiation of dynamic objects–a similar effect as existing parser generators exhibit as well, with similar solutions we could apply [25]. Furthermore, currently the Spicy compiler generates code supporting incremental parsing even when it could optimize for UDP where one sees complete PDUs at a time (which Bro's standard parser indeed does). Overall, in [26] we deemed Spicy's performance satisfactory in comparision to production code, given the current prototype state of the compiler implementation. We refer to that work for further discussion.

With those former results available, rather than now repeating similar measurement for the DPI use case once more, for this work we chose to instead examine performance for the HTTP proxy application that we discuss in §4.4, conducting a load test that stresses both dissection and assembly code in a different setting. To that end, we proxied connections against a local web-server running Nginx 1.4.7 on a machine with 16 Intel Xeon E5-2650 CPUs, serving a static HTML page of 3,700 bytes. We performed the test using `ab`, the Apache HTTP server benchmarking tool [1] in single-thread mode. With caching disabled, our proxy could sustain 676 req/s; enabling caching increased that number to 2,505 req/s. For comparison, with no proxy in place, Nginx could sustain a baseline of 3,403 req/s, indicating that the proof-of-concept proxy already fares well in comparison with a highly optimized web server implementation. (Note that the proxy performs the parse/assembly process twice, once for the request and once for the response)

5 Related Work

Our work provides a platform for dissecting network protocols and file formats that integrates results and experiences from a range of previous efforts. Generally, *grammars* represent the standard way to specify parsing strategies. General-purpose parser generators—such as Yacc, Bison, and Antlr—compile grammars into parsers that applications then integrate. However, their domain-independent nature comes with conceptual and technical limitations that render them an ill fit for dissecting protocols and file formats. Approaches specific to the networking domain sometimes work from Augmented Backus-Naur Form (ABNF)For example, Zebu [4] generates parsers from annotated versions of ABNF. More commonly in the networking domain, applications represent formats as collections of *types*, which they then make accessible programmatically. As Fisher et al. observe [8], since types can describe both external and in-memory representation simultaneously, it proves natural for programmers to define data layouts in that form. Examples of type-based systems include: PacketTypes [14], which compiles PDU specifications into corresponding C dissectors, with a focus on matching packets to a particular protocol; DataScript [2], which generates Java libraries for dissection and assembly from specifications of binary file formats; BinPAC [20], a "yacc for network protocols" that turns protocol specifications into C++ parsers; PADS [7], which targets C and focuses on robust error handling, a variety of encodings, and tool support; and GAPA [3], an interpreted system that emphasizes type-safety and absence of infinite loops during the dissection process. Spicy follows the type-based approach as well, generalizing it beyond what these systems can express.

To the best of our knowledge, BinPAC is the only of these type-based systems with a wide-spread deployment base today, due to its integration into Bro [22]. A number of later efforts extend BinPAC in specific ways. For example, UltraPAC [13] narrows the target application to improve efficiency; and Schear et al. improve its memory management [25]. Spicy takes a conceptually different approach from BinPAC by unifying syntax and semantics inside a single language, in constrast to depending on user C++ code to steer parsing and track state. Yet, the BinPAC/Bro combo inspired Spicy: it generalizes port-independent protocol analysis [5] and extends Pang et al.'s work [19] reversing the process to assemble wire format.

A variety of further related tools and projects exist. Haka [10] is an open-source security-oriented language for describing protocols and policies. It avoids the pitfalls of C by relying on Lua, which, however, leads to an unnatural mapping of Lua's low-level constructs to protocol elements. Hammer [11] is a C library for building recursive-descent LL(k) parsers for (primarily) binary protocols; a Hammer dissector consists of C code that parses elements through individual Hammer function calls. While Hammer provides bindings to other languages, performance then suffers and integration into existing systems gets more difficult. Microsoft's Open Protocol Notation [18] provides a description language that enables developers to model protocol architecture, behavior, and data. It is embedded into a complex system of .NET classes, and seems to primarily target extensions for Microsoft's Message Analyzer product. Conceptually however, the language is powerful, sharing features like decapsulation and LL(1) parsing with Spicy. Scapy [24] enables writing protocol dissectors in Python if performance is of no concern. As a different category of related efforts, tools such as Protocol Buffers [23] and Thrift [30] facilitate object serialization. However, with these, one *defines* the format, rather than dissecting existing ones.

On the backend side, Spicy leverages our HILTI [26] system, a domain-specific abstract machine model with a corresponding open-source compiler toolchain. HILTI provides Spicy with low-level abstractions and idioms for code generation, as well as with a safe execution model. The measurements in [26] assess Spicy's performance operating on top of HILTI. The contribution there, however, concerns the low-level abstract machine model, and the discussion thus treats Spicy's functionality at the level of the much more limited BinPAC system (hence its name there, "BinPAC++").

6 Conclusion

This work presents a novel framework for dissecting and assembling wire format data from high-level specifications, enabling DPI applications to support a wide range of protocols and file formats robustly and efficiently. Different from existing efforts, Spicy's language tightly couples syntax and semantics, and more generally integrates previously separate concepts and capabilities into a unified model. Spicy comes with a compiler toolchain that generates code for host applications, relieving their programmers from the error-prone, low-level work that this task entails. Spicy also facilitates reuse of dissectors, enabling applications to benefit from the expertise of others who have already implemented support for a format.

Going forward, we will continue this work in two directions. First, we aim to advance our implementation from its current prototype state to production quality code that can support the demands of high-performance network environments. Second, as Spicy captures a format's semantics in high-level terms, we see significant potential for employing domain-specific compiler optimization strategies that adapt Spicy's output to specifics of the target setting, for example by removing analyses of elements that a host application does not need.

References

[1] ab - Apache HTTP server benchmarking tool. `https://httpd.apache.org/docs/2.2/programs/ab.html`.

[2] G. Back. DataScript - A Specification and Scripting Language for Binary Data. In *ACM GPCE*, 2002.

[3] N. Borisov, D. J. Brumley, and H. J. Wang. A Generic Application-level Protocol Analyzer and Its Language. In *NDSS*, 2007.

[4] L. Burgy, L. Reveillere, J. Lawall, and G. Muller. Zebu: A Language-Based Approach for Network Protocol Message Processing. *IEEE Trans. Software Engineering*, 37(4):575–591, July 2011.

[5] H. Dreger, A. Feldmann, M. Mai, V. Paxson, and R. Sommer. Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection. In *USENIX Security Symposium*, 2006.

[6] P. Eronen. TLS Record Layer Bugs (Presentation at IETF67). `http://www.ietf.org/proceedings/67/slides/tls-3/tls-3.ppt`, 2006.

[7] K. Fisher and R. Gruber. PADS: A Domain-specific Language for Processing Ad Hoc Data. In *ACM PLDI*, 2005.

[8] K. Fisher, Y. Mandelbaum, and D. Walker. The Next 700 Data Description Languages. In *ACM POPL*, pages 2–15, 2006.

[9] D. Grune and J. Ceriel. *Parsing Techniques: A Practical Guide*. Springer Publishing Company, 2nd edition, 2010.

[10] Haka—Software Defined Security. `http://www.haka-security.org`.

[11] Hammer. `https://github.com/UpstandingHackers/hammer`.

[12] D. Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968.

[13] Z. Li et al. NetShield: Massive Semantics-Based Vulnerability Signature Matching for High-Speed Networks. In *ACM SIGCOMM*, 2010.

[14] P. J. McCann and S. Chandra. Packet Types: Abstract Specification of Network Protocol Messages. In *ACM SIGCOMM*, pages 321–333, 2000.

[15] Microsoft Developer Network – DocSigSerializedCertStore. `https://msdn.microsoft.com/en-us/library/dd922793%28v=office.12%29.aspx`.

[16] Microsoft support – TLS/SSL fragmentation update. `http://support.microsoft.com/kb/2541763`.

[17] OpenSSL Security Advisory. `https://www.openssl.org/news/secadv_20140806.txt`, Aug. 2014.

[18] PEF Architecture Tutorial. `https://msdn.microsoft.com/en-us/library/jj714800.aspx`.

[19] R. Pang and V. Paxson. A High-Level Programming Environment for Packet Trace Anonymization and Transformation. In *ACM SIGCOMM*, Aug. 2003.

[20] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: A yacc for Writing Application Protocol Parsers. In *ACM IMC*, 2006.

[21] Parsers in the Spicy GitHub Repository. `https://github.com/rsmmr/hilti/tree/master/libbinpac/parsers` and `https://github.com/rsmmr/hilti/tree/master/bro/pac2`.

[22] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23–24), 1999.

[23] Protocol Buffers. `https://code.google.com/p/protobuf`.

[24] Scapy. `http://www.secdev.org/projects/scapy`.

[25] N. Schear, D. Albrecht, and N. Borisov. High-Speed Matching of Vulnerability Signatures. In *RAID*, 2008.

[26] R. Sommer, M. Vallentin, L. De Carli, and V. Paxson. HILTI: An Abstract Execution Environment for Deep, Stateful Network Traffic Analysis. In *ACM IMC*, 2014.

[27] The Spicy Home Page. `http://www.icir.org/hilti`.

[28] The Open Information Security Foundation. `http://www.openinfosecfoundation.org`.

[29] The LLVM Compiler Infrastructure. `http://llvm.org`.

[30] Apache Thrift. `http://thrift.apache.org`.

[31] Wireshark. `http://www.wireshark.org`.

A Summary of the Spicy Language

Unit Syntax

`<name>:<type> [<attribute annotations>]`	
`[ [>] { <hook-code> } ]`	Attribute to dissect/assemble according to type; execute hook after dissection, or before assembly (w/ >).
`<name>:<type> if (<cond>)`	Skip attribute if `<cond>` is false.
`<name><<type> [...];`	Attribute to use only when dissecting.
`<name>><type> [...];`	Attribute to use only when assembling.
`switch (<expr>) {<cases>}`	Branch on expression value.
`switch { <cases> }`	Branch on look-ahead token.
`on <attribute> { <code> }`	Alternative form of attribute hook.
`on %<unit-hook> { <code> }`	Predefined callbacks; see top right.
`var <name>:<type>[=<expr>]`	Define instance variable; unset by default.

Inline unit properties

`%byteorder=<expr>`	Default byte order for unit.
`%description=<expr>`	Textual description of unit/dissector.
`%mimetype=<expr>`	Content type for attaching to sinks.
`%port=<expr>`	Well-known port for host application.
`%sync-(after|at) = (<regexp>|<mark>|<object>)`	On error, continue at/after hint.

Types

`addr`, **`bitfield`**, **`bool`**, **`bytes`**, **`double`**, **`enum`**, **`int`**`<N>`, **`interator`**`<T>`, **`interval`**, **`list`**`<T>`, **`map`**`<T_1,T_2>` **`regexp`**, **`set`**`<T>`, **`sink`**, **`string`**, **`time`**, **`uint`**`<N>`, **`unit`**`<T_1,...,T_n>`, **`vector`**`<T>`

Unit Hooks

`%confirmed`	Unit confirmed input format.
`%disable`	Unit disabled due to not parsing expected format.
`%done`	Unit finished dissection/assembly.
`%error`	Parse error occured.
`%gap(seq, len)`	Sink reassembly reports gap.
`%init`	Unit begins dissection/assembly.
`%overlap(seq, s1, s2)`	Sink reassembly reports overlapping segments.
`%skip(seq)`	Sink reassembly skipped over data.
`%synced`	Error recovery has succesfully resynchronized.
`%undelivered(seq, d)`	Sink reassembly failed putting data chunk in order.

Attribute Annotations

`&length=<expr>`	Parse attribute from next N bytes.
`&byteorder=<expr>`	Assume given byte order.
`&bitorder=<expr>`	For bitfields, define bit order.
`&convert=<expr>`	When dissecting, replace value with expression.
`&convert_back=<expr>`	When assembling, replace value with expression.
`&default=<expr>`	Initialize attribute with default expression.
`&chunked`	For bytes attributes, store data incrementally.
`&eod`	For bytes attributes, take all until end of input.
`&ipv4/&ipv6`	For address attributes, parse as IPv4/IPv6 address.
`&parse=<expr>`	Parse attribute from data that expression returns.
`&transient`	Allow optimizing by not storing attributes values.
`&synchronize`	Allow synchronization here after parse errors.
`&try`	Remember current position for later backtracking.
`&count=<expr>`	For containers, parse given number of elements.
`&until=<expr>`	For containers, parse until expression yields true.
`&while=<expr>`	For containers, parse while expression yields true.