Spicy

Release 1.7.0

Zeek Project

Jun 28, 2023

CONTENTS

Jun 28, 2023.

```
# http-request.spicy

module HTTP;

const Token      = /[^ \t\r\n]+/;
const WhiteSpace = /[ \t]+/;
const NewLine    = /\r?\n/;

public type RequestLine = unit {
    method:  Token;
    :        WhiteSpace;
    uri:     Token;
    :        WhiteSpace;
    version: Version;
    :        NewLine;

    on %done { print self; }
};

type Version = unit {
    :       /HTTP\//;
    number: /[0-9]+\.[0-9]+/;
};
```

```
# echo "GET /index.html HTTP/1.0" | spicy-driver http-request.spicy
[$method=b"GET", $uri=b"/index.html", $version=[$number=b"1.0"]]
```

Overview Spicy is a parser generator that makes it easy to create robust C++ parsers for network protocols, file formats, and more. Spicy is a bit like a "yacc for protocols", but it's much more than that: It's an all-in-one system enabling developers to write attributed grammars that describe both syntax and semantics of an input format using a single, unified language. Think of Spicy as a domain-specific scripting language for all your parsing needs.

The Spicy toolchain turns such grammars into efficient C++ parsing code that exposes an API to host applications for instantiating parsers, feeding them input, and retrieving their results. At runtime, parsing proceeds fully incrementally—and potentially highly concurrently—on input streams of arbitrary size. Compilation of Spicy parsers takes place either just-in-time at startup (through a C++ compiler); or ahead-of-time either by creating pre-compiled shared libraries, or by giving you generated C++ code that you can link into your application.

Spicy comes with a Zeek plugin that enables adding new protocol and file analyzers to Zeek without having to write any C++ code. You define the grammar, specify which Zeek events to generate, and Spicy takes care of the rest. There's also a Zeek analyzers package that provides Zeek with several new, Spicy-based analyzers.

See our *collection of example grammars* to get a sense of what Spicy looks like.

License Spicy is open source and released under a BSD license, which allows for pretty much unrestricted use as long as you leave the license header in place. You fully own any parsers that Spicy generates from your grammars.

History Spicy was originally developed as a research prototype at the International Computer Science Institute with funding from the U.S. National Science Foundation. Since then, Spicy has been rebuilt from the ground up by Corelight, which has contributed the new implementation to the Zeek Project.

CHAPTER ONE

GETTING IN TOUCH

Having trouble using Spicy? Have ideas how to make Spicy better? We'd like to hear from you!

- Report issues on the GitHub ticket tracker.
- Ask the #spicy channel on Zeek's Slack.
- Propose ideas, and show what you're doing, on GitHub's Discussions.
- Visit the [Zeek community](https://community.zeek.org) to discuss Spicy under the [Spicy tag](https://community.zeek.org/c/spicy/).
- To follow development, subscribe to the commits mailing list (it can be noisy!).

CHAPTER

TWO

DOCUMENTATION

2.1 Installation

Spicy can be installed from pre-built binaries (Linux, macOS) or with Homebrew (macOS), executed via Docker containers (Linux), or built from source (Linux, macOS, FreeBSD):

We generally aim to follow Zeek's platform policy on which platforms to support and test.

Note: Most of the installation options discussed in this chapter do *not* include the Zeek plugin for Spicy. We recommend installing the plugin through Zeek's package manager; see its *installation instructions*.

2.1.1 Pre-built binaries

Linux

We provide pre-built Spicy binaries for a range of Linux distributions, both for the current release version and for development builds made from the Git `main` branch.

These binary artifacts are distributed as either DEB or RPM packages for the corresponding distribution; or, in a couple cases, as TAR archives. To install the binaries, download the corresponding package and execute one of the following:

DEB packages

```
# dpkg --install spicy.deb
```

RPM packages

```
# rpm -i spicy.rpm
```

TAR archives The TAR archives need to be unpacked into `/opt/spicy`. Any previous installation must be removed first:

```
# rm -rf /opt/spicy && mkdir /opt/spicy
# tar xf spicy.tar.gz -C /opt/spicy --strip-components=1
```

The binaries may require installation of additional dependencies; see the `Dockerfile` for the respective platform for what's needed.

Platform	Release Version	Development Version	Dockerfile
Alpine 3.12	TAR	TAR	Dockerfile
CentOS Stream 8	TAR	RPM	Dockerfile
Debian 9	DEB	DEB	Dockerfile
Debian 10	DEB	DEB	Dockerfile
Debian 11	DEB	DEB	Dockerfile
Fedora 35	RPM	RPM	Dockerfile
Fedora 36	RPM	RPM	Dockerfile
Ubuntu 16	DEB	DEB	Dockerfile
Ubuntu 18	DEB	DEB	Dockerfile
Ubuntu 20	DEB	DEB	Dockerfile

macOS

Homebrew

We provide a Homebrew formula for installation of Spicy. After installing Homebrew add the Zeek tap:

```
# brew tap zeek/zeek
```

To install the most recent Spicy release version, execute:

```
# brew install spicy
```

To instead install the current development version, execute:

```
# brew install --HEAD spicy
```

Pre-built binaries

We provide TAR archives with pre-built binaries for the following macOS versions:

macOS	Release Version	Development Version
Monterey (12)		TAR
Ventura (13)		TAR

The TAR archives need to be unpacked into `/opt/spicy`. Any previous installation must be removed first. To prevent macOS from quarantining the files, you should download and unpack via the command line:

```
# curl -L <link-per-above> -o spicy.tar.gz
# rm -rf /opt/spicy && mkdir /opt/spicy
# tar xf spicy.tar.gz -C /opt/spicy --strip-components 1
```

For JIT support, these binaries require an Xcode installation.

2.1.2 Using Docker

The Zeek Docker images include Spicy. See their documentation on how to run them.

Note: Docker Desktop for Mac uses a VM behind the scenes to host the Docker runtime environment. By default it allocates 2 GB of RAM to the VM. This is not enough to compile Spicy analzers and will cause an error that looks something like this:

```
c++: internal compiler error: Killed (program cc1plus)
Please submit a full bug report,
with preprocessed source if appropriate.
See <file:///usr/share/doc/gcc-7/README.Bugs> for instructions.
```

This is due to the VM hitting an out-of-memory condition. To avoid this you will need to allocate more RAM to the VM. Click on the Docker Icon in your menubar and select “Preferences”. Click on the “Advanced” tab and then use the slider to select 8 GB of RAM. Docker Desktop will restart and then you will be ready to go.

2.1.3 Building from source

Prerequisites

To build Spicy from source, you will need:

- For compiling the toolchain:
  - A C++ compiler that supports C++17 (known to work are Clang >= 9 and GCC >= 9)
  - CMake >= 3.15
  - Bison >= 3.0
  - Flex >= 2.6

  - Python >= 3.4
  - Zlib (no particular version)
- For testing:
  - BTest >= 0.66 (`pip install btest`)
  - Bash (for BTest)
- For building the documentation:
  - Sphinx >= 1.8
  - Pygments >= 2.5
  - Read the Docs Sphinx Theme (`pip install sphinx_rtd_theme`)

In the following we record how to get these dependencies in place on some popular platforms. Please file an issue if you have instructions for platforms not yet listed here.

Linux

See the corresponding *Dockerfiles*.

macOS

Make sure you have Xcode installed, including its command-line tools (`xcode-select --install`).

If you are using Homebrew:

```
# brew install bison flex cmake ninja python@3.8 sphinx-doc
# pip3 install btest sphinx_rtd_theme
```

If you are using MacPorts:

```
# port install flex bison cmake ninja python38 py38-pip py38-sphinx py38-sphinx_rtd_
↪theme
# pip install btest
```

FreeBSD

See the prepare script coming with the Spicy distribution.

Building Spicy

Get the code:

```
# git clone --recursive https://github.com/zeek/spicy
```

The short version to build Spicy is the usual process then:

```
# ./configure && make && make install
```

However, you may want to customize the build a bit, see the output `./configure --help` for the available options. In particular, you can use `--prefix=/other/path` to install into something else than `/usr/local`.

The final `configure` output will summarize your build's configuration.

Note: For developers, the following `configure` options may be particular useful:

- `--enable-ccache`: use the `ccache` compiler cache to speed up compilation
- `--enable-debug`: compile a non-optimized debug version
- `--enable-sanitizer`: enable address & leak sanitizers
- `--generator=Ninja`: use the faster `ninja` build system instead of `make`

Using Ninja and `ccache` will speed up compile times. On Linux, compiling will also be quite a bit faster if you have the "Gold linker" available. To check if you do, see if `which ld.gold` returns anything. If yes, `configure` will automatically pick it up.

Once you have configured Spicy, running `make` will change into the newly created `build` directory and start the compilation there. Once finished, `make test` will execute the test suite. It will take a bit, but all tests should be passing (unless explicitly reported as expected to fail). Finally, `make install` will install Spicy system-wide into the configured prefix. If you are installing into a non-standard location, make sure that `<prefix>/bin` is in your `PATH`.

Note: You can also use the Spicy tools directly out of the build directory without installing; the binaries land in `build/bin`.

To build Spicy's documentation, run `make` inside the `docs/` directory. Documentation will then be located in `build/doc/html`.

2.1.4 Parser development setup

In order to speed up compilation of Spicy parsers, users can create a cache of precompiled files. This cache is tied to a specific Spicy version, and needs to be recreated each time Spicy is updated.

To precompile the files execute the following command:

```
# spicy-precompile-headers
```

Note: By default the cache is located in the folder `.cache/spicy/<VERSION>` inside the user's home directory. This location can be overridden by setting the environment variable `SPICY_CACHE` to a different folder path, both when executing `spicy-precompile-headers` and Spicy toolchain commands.

2.2 Getting Started

The following gives a short overview how to write and use Spicy parsers. We won't use many of Spicy's features yet, but we we'll walk through some basic code examples and demonstrate typical usage of the Spicy toolchain.

2.2.1 Hello, World!

Here's a simple "Hello, world!" in Spicy:

```
module Test;

print "Hello, world!";
```

Assuming that's stored in `hello.spicy`, you can compile and execute the code with Spicy's standalone compiler `spicyc`:

```
# spicyc -j hello.spicy
Hello, world!
```

`spicyc -j` compiles the source code into native code on the fly using your system's C++ compiler, and then directly executes the result. If you run `spicyc -c hello.spicy`, you will see the C++ code that Spicy generates behind the scenes.

You can also precompile the code into an object file, and then load that for immediate execution:

```
# spicyc -j -o hello.hlto hello.spicy
# spicyc -j hello.hlto
Hello, world!
```

To compile Spicy code into an actual executable on disk, use `spicy-build`:

```
# spicy-build -o a.out hello.spicy
# ./a.out
Hello, world!
```

`spicy-build` is a small shell script that wraps `spicyc -c` and runs the resulting code through the system's C++ compiler to produce an executable.

Note:

Internally, Spicy employs another intermediary language called *HILTI* that sits between the Spicy source code and the generated C++ output. For more complex Spicy grammars, the HILTI code is often far easier to comprehend than the final C++ code, in particular once we do some actual parsing. To see that intermediary HILTI code, execute `spicy -p hello.spicy`. The `.hlto` extension comes from HILTI as well: It's an HILTI-generated object file.

2.2.2 A Simple Parser

To actually parse some data, we now look at a small example dissecting HTTP-style request lines, such as: `GET /index.html HTTP/1.0`.

Generally, in Spicy you define parsers through types called "units" that describe the syntax of a protocol. A set of units forms a *grammar*. In practice, Spicy units typically correspond pretty directly to protocol data units (PDUs) as protocol specifications tend to define them. In addition to syntax, a Spicy unit type can also specify semantic actions, called *hooks*, that will execute during parsing as the corresponding pieces are extracted.

Here's an example of a Spicy script for parsing HTTP request lines:

Listing 1: my-http.spicy

```
module MyHTTP;

const Token      = /[^ \t\r\n]+/;
const WhiteSpace = /[ \t]+/;
const NewLine    = /\r?\n/;

type Version = unit {
    :       /HTTP\//;
    number: /[0-9]+\.[0-9]+/;
};

public type RequestLine = unit {
    method:  Token;
    :        WhiteSpace;
    uri:     Token;
    :        WhiteSpace;
    version: Version;
    :        NewLine;

    on %done {
        print self.method, self.uri, self.version.number;
        }
};
```

In this example, you can see a number of things that are typical for Spicy code:

- A Spicy input script starts with a `module` statement defining a namespace for the script's content.
- The layout of a piece of data is defined by creating a `unit` type. The type lists individual *fields* in the order they are to be parsed. The example defines two such units: `RequestLine` and `Version`.
- Each field inside a unit has a type and an optional name. The type defines how that field will be parsed from raw input data. In the example, all fields use regular expressions instead of actual data types (`uint32` would be an actual type), which means that the generated parser will match these expressions against the input stream. Assuming a match, the corresponding value will then be recorded with type `bytes`, which is Spicy's type for binary data. Note how the regular expressions can either be given directly as a field's type (as in `Version`), or indirectly via globally defined constants (as in `RequestLine`).
- If a field has a name, it can later be referenced to access its value. Consequently, in this example all fields with semantic meanings have names, while those which are unlikely to be relevant later do not (e.g., whitespace).
- A unit field can have another unit as its type; here that's the case for the `version` field in `RequestLine`; we say that `Version` is a *subunit* of `RequestLine`. The meaning for parsing is straight-forward: When parsing the top-level unit reaches the field with the subunit, it switches to processing that field according to the subunit's definition. Once the subunit is fully parsed, the top-level unit's next field is processed as normal from the remaining input data.

- We can specify code to be executed when a unit has been completely parsed by implementing a hook called `%done`. Inside the hook's code body, statements can refer to the unit instance currently being parsed through an implicitly defined `self` identifier. Through `self`, they can then access any fields already parsed by using a standard attribute notation (`self.<field>`). As the access to `version` shows, this also works for getting to fields nested inside subunits. In the example, we tell the generated parser to print out three of the parsed fields whenever a `RequestLine` has been fully parsed.
- The `public` keyword exposes the generated parser of a unit to to external host applications wanting to deploy it. Only public units can be used as the starting point for feeding input; non-public subunits cannot be directly instantiated by host applications.

Now let us see how we turn this into an actual parser that we can run. Spicy comes with a tool called `spicy-driver` that acts as a generic, standalone host application for Spicy parsers: It compiles Spicy scripts into code and then feeds them its standard input as data to parse. Internally, `spicy-driver` uses much of the same machinery as `spicyc`, but provides additional code kicking off the actual parsing as well.

With the above Spicy script in a file `my-http.spicy`, we can use `spicy-driver` on it like this:

```
# echo "GET /index.html HTTP/1.0" | spicy-driver my-http.spicy
GET, /index.html, 1.0
```

As you see, the `print` statement inside the `%done` hook wrote out the three fields as we would expect (`print` automatically separates its arguments with commas). If we pass something into the driver that's malformed according to our grammar, the parser will complain:

```
# echo "GET XXX/1.0" | spicy-driver my-http.spicy
[fatal error] terminating with uncaught exception of type spicy::rt::ParseError:␣
↪parse error: failed to match regular expression (my-http.spicy:7)
```

Using `spicy-driver` in this way relies on Spicy's support for just-in-time compilation, just like `spicyc -j`. In the background, there's C++ code being generated and compiled without that we see it. Just like in the earlier example, we can also either use `spicyc` to precompile the C++ code into an object file that `spicy-driver` can then load, or use `spicy-build` to give us an actual executable:

```
# spicyc -j -o my-http.hlto  my-http.spicy
# echo "GET /index.html HTTP/1.0" | spicy-driver my-http.hlto
GET, /index.html, 1.0
```

```
# spicy-build -o a.out my-http.spicy
# echo "GET /index.html HTTP/1.0" | ./a.out
GET, /index.html, 1.0
```

Spicy also comes with another tool *spicy-dump* that works similar to `spicy-driver`, but prints out the parsed fields at the end, either in a custom ASCII representation or as JSON:

```
# echo "GET /index.html HTTP/1.0" | spicy-dump my-http.hlto
MyHTTP::RequestLine {
    method: GET
    uri: /index.html
    version: MyHTTP::Version {
        number: 1.0
    }
}

# echo "GET /index.html HTTP/1.0" | spicy-dump -J my-http.hlto
{"method":"GET","uri":"/index.html","version":{"number":"1.0"}}
```

If you want to see the actual parsing code that Spicy generates, use `spicyc` again: `spicyc -c my-http.spicy` will show the C++ code, and `spicyc -p my-http.spicy` will show the intermediary HILTI code.

2.2.3 Zeek Integration

Now let's use our `RequestLine` parser with Zeek. For that we need to prepare some input and then we can use the grammar that we already got to add a new protocol analyzer to Zeek.

Preparations

Because Zeek works from network packets, we first need a packet trace with the payload we want to parse. We can't just use a normal HTTP session as our simple parser wouldn't go further than just the first line of the protocol exchange and then bail out with an error. So instead, for our example we create a custom packet trace with a TCP connection that carries just a single HTTP request line as its payload:

```
# tcpdump -i lo0 -w request-line.pcap port 12345 &
# nc -l 12345 &
# echo "GET /index.html HTTP/1.0" | nc localhost 12345
# killall tcpdump nc
```

This gets us `this trace file`.

Next we need to make sure that Zeek can load Spicy parsers. As of version 5.0 Zeek by default bundles Spicy and spicy-plugin which makes Spicy parsers available to Zeek.

> # zeek -N <. . . > Zeek::Spicy - Support for Spicy parsers (`*.spicy`, `*.evt`, `*.hlto`) (built-in)

Depending on the Zeek build configuration this could also report

> Zeek::Spicy - Support for Spicy parsers (`*.spicy`, `*.evt`, `*.hlto`) (dynamic, version 1.3.13)

If you do not see the Spicy plugin listed you need to install the Zeek plugin spicy-plugin yourself, see *the Zeek package installation guide*.

Adding a Protocol Analyzer

Now we can go ahead and add a new protocol analyzer to Zeek. We already got the Spicy grammar to parse our connection's payload, it's in `my-http.spicy`. In order to use this with Zeek, we have two additional things to do: (1) We need to let Zeek know about our new protocol analyzer, including when to use it; and (2) we need to define at least one Zeek event that we want our parser to generate, so that we can then write a Zeek script working with the information that it extracts.

We do both of these by creating an additional control file for Zeek:

Listing 2: my-http.evt

```
1  protocol analyzer spicy::MyHTTP over TCP:
2      parse originator with MyHTTP::RequestLine,
3      port 12345/tcp;
4
5  on MyHTTP::RequestLine -> event MyHTTP::request_line($conn, self.method, self.uri,␣
   ↪self.version.number);
```

The first block (lines 1-3) tells Zeek that we have a new protocol analyzer to provide. The analyzer's Zeek-side name is `spicy::MyHTTP`, and it's meant to run on top of TCP connections (line 1). Lines 2-3 then provide Zeek with more specifics: The entry point for originator-side payload is the `MyHTTP::RequestLine` unit type that our Spicy grammar defines (line 2); and we want Zeek to activate our analyzer for all connections with a responder port of 12345 (which, of course, matches the packet trace we created).

The second block (line 5) tells the Spicy plugin that we want to define one event. On the left-hand side of that line we give the unit that is to trigger the event. The right-hand side defines its name and arguments. What we are saying here is that every time a RequestLine line has been fully parsed, we'd like a MyHTTP::request_line event to go to Zeek. Each event instance will come with four parameters: Three of them are the values of corresponding unit fields, accessed just through normal Spicy expressions (inside an event argument expression, self refers to the unit instance that has led to the generation of the current event). The first parameter, $conn, is a "magic" keyword that lets the Spicy plugin pass the Zeek-side connection ID (conn_id) to the event.

Now we got everything in place that we need for our new protocol analyzer—except for a Zeek script actually doing something with the information we are parsing. Let's use this:

Listing 3: my-http.zeek

```
event MyHTTP::request_line(c: connection, method: string, uri: string, version: string)
        {
        print fmt("Zeek saw from %s: %s %s %s", c$id$orig_h, method, uri, version);
        }
```

You see an Zeek event handler for the event that we just defined, having the expected signature of four parameters matching the types of the parameter expressions that the *.evt file specifies. The handler's body then just prints out what it gets.

Finally we can put together our pieces by pointing Zeek to all the files we got:

```
# zeek -Cr request-line.pcap my-http.spicy my-http.evt my-http.zeek
Zeek saw from 127.0.0.1: GET /index.html 1.0
```

When Zeek starts up here, it passes any *.spicy and *.evt on to the Spicy plugin, which then first kicks off all of its code generation. Afterwards the plugin registers the new analyzer with the Zeek event engine. Zeek then begins processing the packet trace as usual, now activating our new analyzer whenever it sees a TCP connection on port 12345. Accordingly, the MyHTTP::request_line event gets generated once the parser gets to process the session's payload. The Zeek event handler then executes and prints the output we would expect.

Note: By default, the Zeek plugin suppresses any output from Spicy-side print statements. You can add Spicy::enable_print=T to the command line to see it. In the example above, you would then get an additional line of output: GET, /index.html, 1.0.

If you tried the above, you will have noticed that Zeek took a little while to start up. That's of course because we're compiling C++ code in the background again before any packet processing can even begin. To accelerate the startup, we can once more precompile our analyzer similar to what we did before with spicyc. We'll use a different tool here, though: spicyz is a small standalone application for precompiling analyzers for the Spicy plugin to later load. We give spicyz (1) the *.spicy and *.evt inputs that we handed to Zeek above; and (2) an output *.hlto file to write the compiled analyzer into:

```
# spicyz -o my-http-analyzer.hlto my-http.spicy my-http.evt
# zeek -Cr request-line.pcap my-http-analyzer.hlto my-http.zeek
Zeek saw from 127.0.0.1: GET /index.html 1.0
```

That zeek execution is now happening instantaneously.

2.2.4 Custom Host Application

Spicy parsers expose a C++ API that any application can leverage to send them data for processing. The specifics of how to approach this depend quite a bit on the particular needs of the application (Is it just a single, static parser that's needed; or a set not known upfront, and compiled dynamically? Just a single input stream, or many? All data in one buffer, or coming in incrementally? How does the application want to access the parsed information?). That said, the most basic use case is quite straight-forward: feeding data into a specific parser. Here's a small C++ program that parses input with our `RequestLine` parser:

Listing 4: my-http.cc

```
#include <iostream>

#include <hilti/rt/libhilti.h>
#include <spicy/rt/libspicy.h>

using spicy::rt::fmt;

int main(int argc, char** argv) {
    hilti::rt::init();
    spicy::rt::init();

    spicy::rt::Driver driver;
    auto parser = driver.lookupParser("MyHTTP::RequestLine");
    assert(parser);

    try {
        std::ifstream in("/dev/stdin", std::ios::in);
        driver.processInput(**parser, in);
    } catch ( const std::exception& e ) {
        std::cerr << e.what() << std::endl;
    }

    spicy::rt::done();
    hilti::rt::done();
    return 0;
}
```

```
# spicy-build -S -o a.out my-http.cc my-http.spicy
# echo "GET /index.html HTTP/1.0" | ./a.out
GET, /index.html, 1.0
# echo 'Hello, World!' | ./a.out
parse error: failed to match regular expression (my-http.spicy:7)
```

We are using `-S` with `spicy-build` because we're providing our own `main` function.

The code in `my-http.cc` is the core of what `spicy-driver` does if we ignore the dynamic JIT compilation. See *Custom Host Applications* for more.

2.3 Frequently Asked Questions

2.3.1 Spicy Language

Are Spicy's global variables *really* global?

Indeed, they are. Changes to global variables become visible to all Spicy code; their values are not associated with specific connections or other dynamic state. If they are public, they can even be accessed from other, unrelated modules as well. This all means that globals often won't be the right tool for the job; it's rare that a parser needs truly global state. Take a look at *Contexts* for a different mechanism tying state to the current connection, which is a much more common requirement.

What do the numbers in `foo.spicy:37:1` or `foo.spicy:37:1-42:19` in messages mean?

These are source locations or source ranges. You might encounter them in e.g., error messages from the Spicy compiler or in errors when parsing fails.

In this case `foo.spicy:37:1` is a source location, in particular it refers to line 37, character 1 in the file `foo.spicy`; `foo.spicy:37:1-42:19` is a source range in the file `foo.spicy` starting in line 37, character 1 and ending in line 42, character 19 in the same file.

2.3.2 Toolchain

Is there a way to speed up compilation of Spicy code?

Depending on the complexity of the Spicy code, processing through `spicyc`/`spicyz`/`spicy-driver` may take a bit. The bulk of the time time tends to be spent on compiling the generated C++ code; often about 80-90%. Make sure to run *spicy-precompile-headers* to speed that up a little. During development of new parsers, it also helps quite a bit to build non-optimized debug versions by adding `--debug` to the command-line.

If you want to see a break-down of where Spicy spends its time, run the tools with `--report-times`. (In the output at the end, `jit` refers to compiling generated C++ code).

2.3.3 Zeek

Do I need to install Spicy and its Zeek plugin to use Spicy parsers in Zeek?

As of version 5.0 Zeek by default bundles Spicy and its Zeek plugin. If that is the case the folder containing the `zeek` binary should also contain e.g., `spicyc` (provided by Spicy) and `spicyz` (provided by Spicy plugin). To check that the Spicy plugin is active look for `Zeek::Spicy` in the output of `zeek -N`:

```
# zeek -N
<...>
Zeek::Spicy - Support for Spicy parsers (``*.spicy``, ``*.evt``, ``*.hlto``) (built-in)
```

If `spicyc` is missing, you need to *install Spicy*; if `spicyz` is missing or `Zeek::Spicy` is not listed you need to *install Spicy plugin*.

Do I need a Spicy installation for using the Zeek plugin?

No, if the Zeek plugin was compiled with `--build-toolchain=no`, it will not require Spicy to be installed on the system. It will only be able to load pre-compiled analyzers then (i.e., `*.hlto` files), which you can create on a similar system that has Spicy installed through *spicyz*. The build process will leave a binary distribution inside your build directory at `zeek/plugin/Zeek_Spicy.tgz`.

Does Spicy support *Dynamic Protocol Detection (DPD)*?

Yes, see the *corresponding section* on how to add it to your analyzers.

Can I write a Layer 2 protocol analyzer with Spicy?

Yes, you can. In Zeek terminology a layer 2 protocol analyzer is a packet analyzer, see the *corresponding section* on how to declare such an analyzer.

I have `print` statements in my Spicy grammar, why do I not see any output when running Zeek?

The Zeek plugin by default disables the output of Spicy-side `print` statements. To enable them, add `Spicy::enable_print=T` to the Zeek command line (or `redef Spicy::enable_print=T;` to a Zeek script that you are loading).

My analyzer recognizes only one or two TCP packets even though there are more in the input.

The Zeek Spicy plugin parses the sending and receiving sides of a TCP connection each according to the given Spicy grammar. This means that if more than one message can be sent per side the grammar needs to allow for that. For example, if the grammar parses messages of the protocol as `Message`, the top-level parsing unit given in the EVT file needs to be able to parse a list of messages `Message[]`.

A simple way to accomplish this is to introduce a parser which wraps messages of the protocol:

```
type Message = unit {
  # Fields for messages of the protocol.
};

# Parser used e.g., in EVT file.
public type Messages = unit {
  messages: Message[];
};
```

2.4 Tutorial: A Real Analyzer

In this chapter we will develop a simple protocol analyzer from scratch, including full Zeek integration. Our analyzer will parse the *Trivial File Transfer Protocol (TFTP)* in its original incarnation, as described in RFC 1350. TFTP provides a small protocol for copying files from a server to a client system. It is most commonly used these days for providing boot images to devices during initialization. The protocol is sufficiently simple that we can walk through it end to end. See its Wikipedia page for more background.

Contents

2.4.1 Creating a Spicy Grammar

We start by developing Spicy grammar for TFTP. The protocol is packet-based, and our grammar will parse the content of one TFTP packet at a time. While TFTP is running on top of UDP, we will leave the lower layers to Zeek and have Spicy parse just the actual UDP application-layer payload, as described in Section 5 of the protocol standard.

Parsing One Packet Type

TFTP is a binary protocol that uses a set of standardized, numerical opcodes to distinguish between different types of packets—a common idiom with such protocols. Each packet contains the opcode inside the first two bytes of the UDP payload, followed by further fields that then differ by type. For example, the following is the format of a TFTP "Read Request" (RRQ) that initiates a download from a server:

```
 2 bytes     string    1 byte     string   1 byte    (from RFC 1350)
 ------------------------------------------------
| Opcode |  Filename  |   0  |    Mode    |   0  |
 ------------------------------------------------
```

A Read Request uses an opcode of 1. The *filename* is a sequence of ASCII bytes terminated by a null byte. The *mode* is another null-terminated byte sequence that usually is either `netascii`, `octet`, or `mail`, describing the desired encoding for data that will be received.

Let's stay with the Read Request for a little bit and write a Spicy parser just for this one packet type. The following is a minimal Spicy unit to parse the three fields:

```
module TFTP;                             # [1]

```

(continues on next page)

(continued from previous page)

```
public type ReadRequest = unit {        # [2]
    opcode:    uint16;                  # [3]
    filename: bytes &until=b"\x00";     # [4]
    mode:      bytes &until=b"\x00";    # [5]

    on %done { print self; }            # [6]
};
```

Let's walk through:

- [1] All Spicy source files must start with a module line defining a namespace for their content. By convention, the namespace should match what is being parsed, so we call ours TFTP. Naming our module TFTP also implies saving it under the name tftp.spicy, so that other modules can find it through import TFTP;. See *Modules* for more on all of this.
- [2] In Spicy, one will typically create a unit type for each of the main data units that a protocol defines. We want to parse a Read Request, so we call our type accordingly. We declare it as public because we want to use this unit as the starting point for parsing data. The following lines then lay out the elements of such a request in the same order as the protocol defines them.
- [3] Per the TFTP specification, the first field contains the opcode as an integer value encoded over two bytes. For multi-byte integer values, it is important to consider the byte order for parsing. TFTP uses network byte order which matches Spicy's default, so there is nothing else for us to do here. (If we had to specify the order, we would add the *&byte-order* attribute).
- [4] The filename is a null-terminated byte sequence, which we can express directly as such in Spicy: The filename field will accumulate bytes until a null byte is encountered. Note that even though the specification of a Read Request shows the 0 as separate element inside the packet, we don't create a field for it, but rather exploit it as a terminator for the file name (which will not be included into the filename stored).
- [5] The mode operates just the same as the filename.
- [6] Once we are done parsing a Read Request, we print out the result for debugging.

We should now be able to parse a Read Request. To try it, we need the actual payload of a corresponding packet. With TFTP, the format is simple enough that we can start by faking data with printf and pipe that into the Spicy tool *spicy-driver*:

```
# printf '\000\001rfc1350.txt\000octet\000' | spicy-driver tftp.spicy
[$opcode=1, $filename=b"rfc1350.txt", $mode=b"octet"]
```

Here, spicy-driver compiles our ReadRequest unit into an executable parser and then feeds it with the data it is receiving on standard input. The output of spicy-driver is the result of our print statement executing at the end.

What would we do with a more complex protocol where we cannot easily use printf to create some dummy payload? We would probably have access to some protocol traffic in pcap traces, however we can't just feed those into spicy-driver directly as they will contain all the other network layers as well that our grammar does not handle (e.g., IP and UDP). One way to test with a trace would be proceeding with Zeek integration at this point, so that we could let Zeek strip off the base layers and then feed our parser only the TFTP payload. However, during development it is often easier at first to extract application-layer protocol data from the traces ourselves, write it into files, and then feed those files into spicy-driver.

We can leverage Zeek for doing this extraction into files. If we had a TCP-based protocol, doing so would be trivial because Zeek has that functionality built in: When you run Zeek on a pcap trace and add Conn::default_extract=T to the command line, it will write out all the TCP streams into individual files. As TFTP is UDP-based, however, we will use a custom script, udp-contents.zeek. When you run Zeek with that script on trace, you will get one file per UDP packet each containing the corresponding application-layer UDP payload (make sure to use this with small traces only . . .).

Let's use the UDP script with `tftp_rrq.pcap`, a tiny TFTP trace containing a single file download from Wireshark's pcap archive. `tcpdump` shows us that the first packet indeed contains a Read Request:

```
# tcpdump -ttnr tftp_rrq.pcap
1367411051.972852 IP 192.168.0.253.50618 > 192.168.0.10.69:  20 RRQ "rfc1350.txtoctet
↪" [|tftp]
1367411052.077243 IP 192.168.0.10.3445 > 192.168.0.253.50618: UDP, length 516
1367411052.081790 IP 192.168.0.253.50618 > 192.168.0.10.3445: UDP, length 4
[...]
```

Running Zeek on the trace with the `udp-contents` scripts produces the expected content files:

```
# zeek -r tftp_rrq.pcap udp-contents
# ls udp-contents.orig.*
udp-contents.orig.1367411051.972852.dat
udp-contents.orig.1367411052.077243.dat
udp-contents.orig.1367411052.086300.dat
udp-contents.orig.1367411052.088995.dat
udp-contents.orig.1367411052.091675.dat
[...]
```

Per the timestamps included with the names, the first file is the one containing our Read Request. We can pass that into our Spicy parser:

```
# cat udp-contents.orig.1367411051.972852.dat | spicy-driver tftp.spicy
[$opcode=1, $filename=b"rfc1350.txt", $mode=b"octet"]
```

That gives us an easy way to test our TFTP parser.

Generalizing to More Packet Types

So far we can parse a Read Request, but nothing else. In fact, we are not even examining the `opcode` yet at all to see if our input actually *is* a Read Request. To generalize our grammar to other TFTP packet types, we will need to parse the `opcode` on its own first, and then use the value to decide how to handle subsequent data. Let's start over with a minimal version of our TFTP grammar that looks at just the opcode:

```
module TFTP;

public type Packet = unit {
    opcode: uint16;

    on %done { print self; }
};
```

```
# cat udp-contents.orig.1367411051.972852.dat | spicy-driver tftp.spicy
[$opcode=1]
```

Next we create a separate type to parse the fields that are specific to a Read Request:

```
type ReadRequest = unit {
    filename: bytes &until=b"\x00";
    mode:     bytes &until=b"\x00";
};
```

We do not declare this type as public because we will use it only internally inside our grammar; it is not a top-level entry point for parsing (that's `Packet` now).

Now we need to tie the two units together. We can do that by adding the `ReadRequest` as a field to the `Packet`, which will let Spicy parse it as a sub-unit:

```
module TFTP;

public type Packet = unit {
    opcode: uint16;
    rrq:    ReadRequest;

    on %done { print self; }
};
```

```
# cat udp-contents.orig.1367411051.972852.dat | spicy-driver tftp.spicy
[$opcode=1, $rrq=[$filename=b"rfc1350.txt", $mode=b"octet"]]
```

However, this does not help us much yet: it still resembles our original version in that it continues to hardcode one specific packet type. But the direction of using sub-units is promising, we only need to instruct the parser to leverage the `opcode` to decide what particular sub-unit to use. Spicy provides a `switch` construct for such dispatching:

```
module TFTP;

public type Packet = unit {
    opcode: uint16;

    switch ( self.opcode ) {
        1 -> rrq: ReadRequest;
    };

    on %done { print self; }
};
```

```
# cat udp-contents.orig.1367411051.972852.dat | spicy-driver tftp.spicy
[$opcode=1, $rrq=[$filename=b"rfc1350.txt", $mode=b"octet"]]
```

The `self` keyword always refers to the unit instance currently being parsed, and we use that to get to the opcode for switching on. If it is `1`, we descend down into a Read Request.

What happens if it is something other than `1`? Let's try it with the first server-side packet, which contains a TFTP acknowledgment (opcode 4):

```
# cat udp-contents.resp.1367411052.081790.dat | spicy-driver tftp.spicy
[fatal error] terminating with uncaught exception of type spicy::rt::ParseError:␣
↪parse error: no matching case in switch statement (:7:5-9:7)
```

Of course it is now easy to add another unit type for handling such acknowledgments:

```
public type Packet = unit {
    opcode: uint16;

    switch ( self.opcode ) {
        1 -> rrq: ReadRequest;
        4 -> ack: Acknowledgement;
    };

```

(continues on next page)

(continued from previous page)

```
    on %done { print self; }
};

type Acknowledgement = unit {
    num: uint16; # block number being acknowledged
};
```

```
# cat udp-contents.resp.1367411052.081790.dat | spicy-driver tftp.spicy
[$opcode=4, $rrq=(not set), $ack=[$num=1]]
```

As expected, the output shows that our TFTP parser now descended into the `ack` sub-unit while leaving `rrq` unset.

TFTP defines three more opcodes for other packet types: `2` is a Write Request, `3` is file data being sent, and `5` is an error. We will add these to our grammar as well, so that we get the whole protocol covered (please refer to the RFC for specifics of each packet type):

```
module TFTP;

public type Packet = unit {
    opcode: uint16;

    switch ( self.opcode ) {
        1 -> rrq:   ReadRequest;
        2 -> wrq:   WriteRequest;
        3 -> data:  Data;
        4 -> ack:   Acknowledgement;
        5 -> error: Error;
    };

    on %done { print self; }
};

type ReadRequest = unit {
    filename: bytes &until=b"\x00";
    mode:     bytes &until=b"\x00";
};

type WriteRequest = unit {
    filename: bytes &until=b"\x00";
    mode:     bytes &until=b"\x00";
};

type Data = unit {
    num:  uint16;
    data: bytes &eod; # parse until end of data (i.e., packet) is reached
};

type Acknowledgement = unit {
    num: uint16;
};

type Error = unit {
    code: uint16;
    msg:  bytes &until=b"\x00";
};
```

This grammar works well already, but we can improve it a bit more.

Using Enums

The use of integer values inside the `switch` construct is not exactly pretty: they are hard to read and maintain. We can improve our grammar by using an enumerator type with descriptive labels instead. We first declare an `enum` type that provides one label for each possible opcode:

```
type Opcode = enum { RRQ = 1, WRQ = 2, DATA = 3, ACK = 4, ERROR = 5 };
```

Now we can change the `switch` to look like this:

```
switch ( self.opcode ) {
        Opcode::RRQ   -> rrq:   ReadRequest;
        Opcode::WRQ   -> wrq:   WriteRequest;
        Opcode::DATA  -> data:  Data;
        Opcode::ACK   -> ack:   Acknowledgement;
        Opcode::ERROR -> error: Error;
        };
```

Much better, but there is a catch still: this will not compile because of a type mismatch. The switch cases' expressions have type `Opcode`, but `self.opcode` remains of type `uint16`. That is because Spicy cannot know on its own that the integers we parse into `opcode` match the numerical values of the `Opcode` labels. But we can convert the former into the latter explicitly by adding a *&convert* attribute to the `opcode` field:

```
public type Packet = unit {
    opcode: uint16 &convert=Opcode($$);
    ...
};
```

This does two things:

1. Each time an `uint16` gets parsed for this field, it is not directly stored in `opcode`, but instead first passed through the expression that `&convert` specifies. Spicy then stores the *result* of that expression, potentially adapting the field's type accordingly. Inside the `&convert` expression, the parsed value is accessible through the special identifier `$$`.
2. Our `&convert` expression passes the parsed integer into the constructor for the `Opcode` enumerator type, which lets Spicy create an `Opcode` value with the label that corresponds to the integer value.

With this transformation, the `opcode` field now has type `Opcode` and hence can be used with our updated switch statement. You can see the new type for `opcode` in the output as well:

```
# cat udp-contents.orig.1367411051.972852.dat | spicy-driver tftp.spicy
[$opcode=Opcode::RRQ, $rrq=[$filename=b"rfc1350.txt", $mode=b"octet"], $wrq=(not set),
↪ $data=(not set), $ack=(not set), $error=(not set)]
```

See *On-the-fly Type Conversion with &convert* for more on `&convert`, and *Enum* for more on the `enum` type.

Note: What happens when `Opcode($$)` receives an integer that does not correspond to any of the labels? Spicy permits that and will substitute an implicitly defined `Opcode::Undef` label. It will also retain the actual integer value, which can be recovered by converting the enum value back to an integer.

Using Unit Parameters

Looking at the two types `ReadRequest` and `WriteRequest`, we see that both are using exactly the same fields. That means we do not really need two separate types here, and could instead define a single `Request` unit to cover both cases. Doing so is straight-forward, except for one issue: when parsing such a `Request`, we would now lose the information whether we are seeing read or a write operation. For our Zeek integration later it will be useful to retain that distinction, so let us leverage a Spicy capability that allows passing state into a sub-unit: *unit parameters*. Here's the corresponding excerpt after that refactoring:

```
public type Packet = unit {
    opcode: uint16 &convert=Opcode($$);

    switch ( self.opcode ) {
        Opcode::RRQ   -> rrq:   Request(True);
        Opcode::WRQ   -> wrq:   Request(False);
        # ...
        };

    on %done { print self; }
};

type Request = unit(is_read: bool) {
    filename: bytes &until=b"\x00";
    mode:     bytes &until=b"\x00";

    on %done { print "We got a %s request." % (is_read ? "read" : "write"); }
};
```

We see that the `switch` now passes either `True` or `False` into the `Request` type, depending on whether it is a Read Request or Write Request. For demonstration, we added another `print` statement, so that we can see how that boolean becomes available through the `is_read` unit parameter:

```
# cat udp-contents.orig.1367411051.972852.dat | spicy-driver tftp.spicy
We got a read request.
[$opcode=Opcode::RRQ, $rrq=[$filename=b"rfc1350.txt", $mode=b"octet"], $wrq=(not set),
↪ $data=(not set), $ack=(not set), $error=(not set)]
```

Admittedly, the unit parameter is almost overkill in this example, but it proves very useful in more complex grammars where one needs access to state information, in particular also from higher-level units. For example, if the `Packet` type stored additional state that sub-units needed access to, they could receive the `Packet` itself as a parameter.

Complete Grammar

Combining everything discussed so far, this leaves us with the following complete grammar for TFTP, including the packet formats in comments as well:

```
# Copyright (c) 2021 by the Zeek Project. See LICENSE for details.
#
# Trivial File Transfer Protocol
#
# Specs from https://tools.ietf.org/html/rfc1350

module TFTP;

# Common header for all messages:
#
```

(continues on next page)

(continued from previous page)

```
#      2 bytes
# ---------------
# |  TFTP Opcode  |
#  ---------------

public type Packet = unit {    # public top-level entry point for parsing
    op: uint16 &convert=Opcode($$);
    switch ( self.op ) {
        Opcode::RRQ   -> rrq:   Request(True);
        Opcode::WRQ   -> wrq:   Request(False);
        Opcode::DATA  -> data:  Data;
        Opcode::ACK   -> ack:   Acknowledgement;
        Opcode::ERROR -> error: Error;
        };
};

# TFTP supports five types of packets [...]:
#
# opcode  operation
#   1     Read request (RRQ)
#   2     Write request (WRQ)
#   3     Data (DATA)
#   4     Acknowledgment (ACK)
#   5     Error (ERROR)
type Opcode = enum {
    RRQ = 0x01,
    WRQ = 0x02,
    DATA = 0x03,
    ACK = 0x04,
    ERROR = 0x05
};

# Figure 5-1: RRQ/WRQ packet
#
#  2 bytes     string    1 byte     string   1 byte
#  ------------------------------------------------
# | Opcode |  Filename  |   0  |    Mode    |   0  |
#  ------------------------------------------------

type Request = unit(is_read: bool) {
    filename: bytes &until=b"\x00";
    mode:     bytes &until=b"\x00";
};

# Figure 5-2: DATA packet
#
#  2 bytes     2 bytes      n bytes
#   ----------------------------------
#  | Opcode |   Block #  |   Data     |
#   ----------------------------------

type Data = unit {
    num:  uint16;
    data: bytes &eod;
};

# Figure 5-3: ACK packet
```

(continues on next page)

(continued from previous page)

```
#
#   2 bytes      2 bytes
#   ----------------------
#  | Opcode |    Block #   |
#   ----------------------

type Acknowledgement = unit {
    num: uint16;
};

#  Figure 5-4: ERROR packet
#
#   2 bytes      2 bytes        string    1 byte
#   -----------------------------------------
#  | Opcode |  ErrorCode |   ErrMsg   |   0  |
#   -----------------------------------------

type Error = unit {
    code: uint16;
    msg:  bytes &until=b"\x00";
};
```

2.4.2 Zeek Integration

To turn the Spicy-side grammar into a Zeek analyzer, we need to provide Spicy's Zeek plugin with a description of how to employ it. There are two parts to that: Telling Zeek when to activate the analyzer, and defining events to generate. In addition, we will need a Zeek-side script to do something with our new TFTP events. We will walk through this in the following, starting with the mechanics of compiling the Spicy analyzer for Zeek. While we will build up the files involved individually first, see the *final section* for how the Zeek package manager, *zkg*, can be used to bootstrap a new Zeek package with a skeleton of everything needed for an analyzer.

Before proceeding, make sure that the Zeek plugin is available. Since Zeek 5.0, it will show up by default:

```
# zeek -N Zeek::Spicy
Zeek::Spicy - Support for Spicy parsers (*.hlto) (built-in)
```

If you are using an older version (or your Zeek has it specifically disabled), follow the instructions to *install the Zeek plugin*. You should now be seeing output similar to this:

```
# zeek -NN Zeek::Spicy
Zeek::Spicy - Support for Spicy parsers (*.hlto) (dynamic, version x.y.z)
```

You should also have `spicyz` in your `PATH`:

```
# which spicyz
/usr/local/zeek/bin/spicyz
```

Note that you need a somewhat recent version of *zkg* to get `spicyz` into your `PATH` automatically; refer to the *plugin instructions plugin* for more.

Compiling the Analyzer

Spicy comes with a tool *spicyz* that compiles Spicy analyzers into binary code that Zeek can load through a Spicy plugin. The following command line produces a binary object file `tftp.hlto` containing the executable analyzer code:

```
# spicyz -o tftp.hlto tftp.spicy
```

Below, we will prepare an additional interface definition file `tftp.evt` that describes the analyzer's integration into Zeek. We will need to give that to `spicyz` as well, and our full compilation command hence becomes:

```
# spicyz -o tftp.hlto tftp.spicy tftp.evt
```

When starting Zeek, we add `tftp.hlto` to its command line:

```
# zeek -r tftp_rrq.pcap tftp.hlto
```

Note: If you get an error from Zeek here, see *Installation* to make sure the Spicy plugin is installed correctly.

Activating the Analyzer

In *Getting Started*, *we already saw* how to inform Zeek about a new protocol analyzer. We follow the same scheme here and put the following into `tftp.evt`, the analyzer definition file:

```
# Note: When line numbers change in this file, update the documentation that pulls it␣
↪in.

protocol analyzer spicy::TFTP over UDP:
```

The first line provides our analyzer with a Zeek-side name (`spicy::TFTP`) and also tells Zeek that we are adding an application analyzer on top of UDP (`over UDP`). `TFTP::Packet` provides the top-level entry point for parsing both sides of a TFTP connection. Furthermore, we want Zeek to automatically activate our analyzer for all sessions on UDP port 69 (i.e., TFTP's well known port). See *Analyzer Setup* for more details on defining such a `protocol analyzer` section.

With this in place, we can already employ the analyzer inside Zeek. It will not generate any events yet, but we can at least see the output of the `on %done { print self; }` hook that still remains part of the grammar from earlier:

```
# zeek -r tftp_rrq.pcap tftp.hlto Spicy::enable_print=T
[$opcode=Opcode::RRQ, $rrq=[$filename=b"rfc1350.txt", $mode=b"octet"], $wrq=(not set),
↪ $data=(not set), $ack=(not set), $error=(not set)]
```

As by default, the Zeek plugin does not show the output of Spicy-side `print` statements, we added `Spicy::enable_print=T` to the command line to turn that on. We see that Zeek took care of the lower network layers, extracted the UDP payload from the Read Request, and passed that into our Spicy parser. (If you want to view more about the internals of what is happening here, there are a couple kinds of *debug output available*.)

You might be wondering why there is only one line of output, even though there are multiple TFTP packets in our pcap trace. Shouldn't the `print` execute multiple times? Yes, it should, but it does not currently: Due to some intricacies of the TFTP protocol, our analyzer gets to see only the first packet for now. We will fix this later. For now, we focus on the Read Request packet that the output above shows.

Defining Events

The core task of any Zeek analyzer is to generate events for Zeek scripts to process. For binary protocols, events will often correspond pretty directly to data units specified by their specifications—and TFTP is no exception. We start with an event for Read/Write Requests by adding this definition to `tftp.evt`:

```
import TFTP;

on TFTP::Request -> event tftp::request($conn);
```

The first line makes our Spicy TFTP grammar available to the rest of the file. The line `on ...` defines one event: Every time a `Request` unit will be parsed, we want to receive an event `tftp::request` with one parameter: the connection it belongs to. Here, `$conn` is a reserved identifier that will turn into the standard connection record record on the Zeek side.

Now we need a Zeek event handler for our new event. Let's put this into `tftp.zeek`:

```
event tftp::request(c: connection)
        {
        print "TFTP request", c$id;
        }
```

Running Zeek then gives us:

```
# spicyz -o tftp.hlto tftp.spicy tftp.evt
# zeek -r tftp_rrq.pcap tftp.hlto tftp.zeek
TFTP request, [orig_h=192.168.0.253, orig_p=50618/udp, resp_h=192.168.0.10, resp_p=69/
↪udp]
```

Let's extend the event signature a bit by passing further arguments:

```
import TFTP;

on TFTP::Request -> event tftp::request($conn, $is_orig, self.filename, self.mode);
```

This shows how each parameter gets specified as a Spicy expression: `self` refers to the instance currently being parsed (`self`), and `self.filename` retrieves the value of its `filename` field. `$is_orig` is another reserved ID that turns into a boolean that will be true if the event has been triggered by originator-side traffic. On the Zeek side, our event now has the following signature:

```
event tftp::request(c: connection, is_orig: bool, filename: string, mode: string)
        {
        print "TFTP request", c$id, is_orig, filename, mode;
        }
```

```
# spicyz -o tftp.hlto tftp.spicy tftp.evt
# zeek -r tftp_rrq.pcap tftp.hlto tftp.zeek
TFTP request, [orig_h=192.168.0.253, orig_p=50618/udp, resp_h=192.168.0.10, resp_p=69/
↪udp], T, rfc1350.txt, octet
```

Going back to our earlier discussion of Read vs Write Requests, we do not yet make that distinction with the `request` event that we are sending to Zeek-land. However, since we had introduced the `is_read` unit parameter, we can easily separate the two by gating event generation through an additional `if` condition:

```
import TFTP;
```

This now defines two separate events, each being generated only for the corresponding value of `is_read`. Let's try it with a new `tftp.zeek`:

```
event tftp::read_request(c: connection, is_orig: bool, filename: string, mode: string)
        {
        print "TFTP read request", c$id, is_orig, filename, mode;
        }

event tftp::write_request(c: connection, is_orig: bool, filename: string, mode:␣
↪string)
        {
        print "TFTP write request", c$id, is_orig, filename, mode;
        }
```

```
# spicyz -o tftp.hlto tftp.spicy tftp.evt
# zeek -r tftp_rrq.pcap tftp.hlto tftp.zeek
TFTP read request, [orig_h=192.168.0.253, orig_p=50618/udp, resp_h=192.168.0.10, resp_
↪p=69/udp], T, rfc1350.txt, octet
```

If we look at the `conn.log` that Zeek produces during this run, we will see that the `service` field is not filled in yet. That's because our analyzer does not yet confirm to Zeek that it has been successful in parsing the content. To do that, we can call a library function that Spicy makes available once we have successfully parsed a request: *spicy::accept_input*. That function signals the host application—i.e., Zeek in our case——that the parser is processing the expected protocol. With that, our request looks like this now:

```
type Request = unit(is_read: bool) {
    filename: bytes &until=b"\x00";
    mode:     bytes &until=b"\x00";

    on %done { spicy::accept_input(); }
};
```

Let's try it again:

```
# spicyz -o tftp.hlto tftp.spicy tftp.evt
# zeek -r tftp_rrq.pcap tftp.hlto tftp.zeek
TFTP read request, [orig_h=192.168.0.253, orig_p=50618/udp, resp_h=192.168.0.10, resp_
↪p=69/udp], T, rfc1350.txt, octet
# cat conn.log
[...]
1367411051.972852  C1f7uj4uuv6zu2aKti  192.168.0.253  50618  192.168.0.10  69  udp ␣
↪spicy_tftp  -  -  -  S0  -  -0  D  1  48  0  0  -
[...]
```

Now the service field says TFTP! (There will be a 2nd connection in the log that we are not showing here; see the next section on that).

Turning to the other TFTP packet types, it is straight-forward to add events for them as well. The following is our complete `tftp.evt` file:

```
# Note: When line numbers change in this file, update the documentation that pulls it␣
↪in.

protocol analyzer spicy::TFTP over UDP:
    parse with TFTP::Packet,
    port 69/udp;

import TFTP;

on TFTP::Request if ( is_read )   -> event tftp::read_request($conn, $is_orig, self.
↪filename, self.mode);
```

(continues on next page)

(continued from previous page)

```
on TFTP::Request if ( ! is_read ) -> event tftp::write_request($conn, $is_orig, self.
↪filename, self.mode);

on TFTP::Data            -> event tftp::data($conn, $is_orig, self.num, self.data);
on TFTP::Acknowledgement -> event tftp::ack($conn, $is_orig, self.num);
on TFTP::Error           -> event tftp::error($conn, $is_orig, self.code, self.msg);
```

Detour: Zeek vs. TFTP

We noticed above that Zeek seems to be seeing only a single TFTP packet from our input trace, even though `tcpdump` shows that the pcap file contains multiple different types of packets. The reason becomes clear once we look more closely at the UDP ports that are in use:

```
# tcpdump -ttnr tftp_rrq.pcap
1367411051.972852 IP 192.168.0.253.50618 > 192.168.0.10.69:  20 RRQ "rfc1350.txtoctet
↪" [tftp]
1367411052.077243 IP 192.168.0.10.3445 > 192.168.0.253.50618: UDP, length 516
1367411052.081790 IP 192.168.0.253.50618 > 192.168.0.10.3445: UDP, length 4
1367411052.086300 IP 192.168.0.10.3445 > 192.168.0.253.50618: UDP, length 516
1367411052.088961 IP 192.168.0.253.50618 > 192.168.0.10.3445: UDP, length 4
1367411052.088995 IP 192.168.0.10.3445 > 192.168.0.253.50618: UDP, length 516
[...]
```

Turns out that only the first packet is using the well-known TFTP port 69/udp, whereas all the subsequent packets use ephemeral ports. Due to the port difference, Zeek believes it is seeing two independent network connections, and it does not associate TFTP with the second one at all due to its lack of the well-known port (neither does `tcpdump`!). Zeek's connection log confirms this by showing two separate entries:

```
# cat conn.log
1367411051.972852  CH3xFz3U1nYI1Dp1Dk  192.168.0.253  50618  192.168.0.10  69  udp ␣
↪spicy_tftp  -  -  -  S0  -  -  0  D  1  48  0  0  -
1367411052.077243  CfwsLw2TaTIeo3gE9g  192.168.0.10  3445  192.168.0.253  50618  udp ␣
↪-  0.181558  24795  196  SF  -  -  0  Dd  49  26167  49  1568  -
```

Switching the ports for subsequent packets is a quirk in TFTP that resembles similar behaviour in standard FTP, where data connections get set up separately as well. Fortunately, Zeek provides a built-in function to designate a specific analyzer for an anticipated future connection. We can call that function when we see the initial request:

```
function schedule_tftp_analyzer(id: conn_id)
        {
        # Schedule the TFTP analyzer for the expected next packet coming in on␣
↪different
        # ports. We know that it will be exchanged between same IPs and reuse the
        # originator's port. "Spicy_TFTP" is the Zeek-side name of the TFTP analyzer
        # (generated from "Spicy::TFTP" in tftp.evt).
        Analyzer::schedule_analyzer(id$resp_h, id$orig_h, id$orig_p, Analyzer::get_
↪tag("Spicy_TFTP"), 1min);
        }

event tftp::read_request(c: connection, is_orig: bool, filename: string, mode:␣
↪string)
        {
        print "TFTP read request", c$id, filename, mode;
        schedule_tftp_analyzer(c$id);
```

(continues on next page)

(continued from previous page)

```
        }

event tftp::write_request(c: connection, is_orig: bool, filename: string, mode:␣
↪string)
        {
        print "TFTP write request", c$id, filename, mode;
        schedule_tftp_analyzer(c$id);
        }

# Add handlers for other packet types so that we see their events being generated.
event tftp::data(c: connection, is_orig: bool, block_num: count, data: string)
        {
        print "TFTP data", block_num, data;
        }

event tftp::ack(c: connection, is_orig: bool, block_num: count)
        {
        print "TFTP ack", block_num;
        }

event tftp::error(c: connection, is_orig: bool, code: count, msg: string)
        {
        print "TFTP error", code, msg;
        }
```

```
# spicyz -o tftp.hlto tftp.spicy tftp.evt
# zeek -r tftp_rrq.pcap tftp.hlto tftp.zeek
TFTP read request, [orig_h=192.168.0.253, orig_p=50618/udp, resp_h=192.168.0.10, resp_
↪p=69/udp], rfc1350.txt, octet
TFTP data, 1, \x0a\x0a\x0a\x0a\x0a\x0aNetwork Working Group [...]
TFTP ack, 1
TFTP data, 2, B Official Protocol\x0a   Standards" for the  [...]
TFTP ack, 2
TFTP data, 3, protocol was originally designed by Noel Chia [...]
TFTP ack, 3
TFTP data, 4, r mechanism was suggested by\x0a   PARC's EFT [...]
TFTP ack, 4
[...]
```

Now we are seeing all the packets as we would expect.

Zeek Script

Analyzers normally come along with a Zeek-side script that implements a set of standard base functionality, such as recording activity into a protocol specific log file. These scripts provide handlers for the analyzers' events, and collect and correlate their activity as desired. We have created such `a script for TFTP`, based on the events that our Spicy analyzer generates. Once we add that to the Zeek command line, we will see a new `tftp.log`:

```
# spicyz -o tftp.hlto tftp.spicy tftp.evt
# zeek -r tftp_rrq.pcap tftp.hlto tftp.zeek
# cat tftp.log
#fields     ts      uid     id.orig_h       id.orig_p       id.resp_h       id.resp_p␣
↪       wrq     fname   mode    uid_data        size    block_sent      block_acked ␣
↪  error_code       error_msg
1367411051.972852   CKWH8L3AIekSHYzBU       192.168.0.253   50618   192.168.0.10    ␣
↪69     F       rfc1350.txt     octet   ClAr3P158Ei77Fq18h      24599   48      49␣
↪    -       -
```

(continues on next page)

(continued from previous page)

```

```

The TFTP script also labels the second session as TFTP data by adding a corresponding entry to the `service` field inside the Zeek-side connection record. With that, we are now seeing this in `conn.log`:

```
1367411051.972852  ChbSfq3QWKuNirt9Uh  192.168.0.253  50618  192.168.0.10  69  udp ␣
↪spicy_tftp  -  -  -  S0  -  -0  D  1  48  0  0  -
1367411052.077243  CowFQj20FHHduhHSYk  192.168.0.10  3445  192.168.0.253  50618  udp ␣
↪spicy_tftp_data  0.181558  24795  196  SF  --  0  Dd  49  26167  49  1568  -
```

The TFTP script ends up being a bit more complex than one would expect for such a simple protocol. That's because it tracks the two related connections (initial request and follow-up traffic on a different port), and combines them into a single TFTP transaction for logging. Since there is nothing Spicy-specific in that Zeek script, we skip discussing it here in more detail.

2.4.3 Creating a Zeek Package

We have now assembled all the parts needed for providing a new analyzer to Zeek. By adding a few further pieces, we can wrap that analyzer into a full *Zeek package* for others to install easily through *zkg*. To help create that wrapping, *zkg* provides a template for instantiating a skeleton analyzer package as a starting point. The skeleton comes in three different flavors, depending on which kind of analyzer you want to create: protocol, file, or packet analyzer. In each case, it creates all the necessary files along with the appropriate directory layout, and even includes a couple of standard test cases.

To create the scaffolding for our TFTP analyzer, execute the following command and provide the requested information:

```
# zkg create --features spicy-protocol-analyzer --packagedir spicy-tftp
"package-template" requires a "name" value (the name of the package, e.g. "FooBar" or
↪"spicy-http"):
name: spicy-tftp
"package-template" requires a "analyzer" value (name of the Spicy analyzer, which␣
↪typically corresponds to the protocol/format being parsed (e.g. "HTTP", "PNG")):
analyzer: TFTP
"package-template" requires a "protocol" value (transport protocol for the analyzer␣
↪to use: TCP or UDP):
protocol: UDP
"package-template" requires a "unit_orig" value (name of the top-level Spicy parsing␣
↪unit for the originator side of the connection (e.g. "Request")):
unit_orig: Packet
"package-template" requires a "unit_resp" value (name of the top-level Spicy parsing␣
↪unit for the responder side of the connection (e.g. "Reply"); may be the same as␣
↪originator side):
unit_resp: Packet
```

The above creates the following files (skipping anything related to `.git`):

```
spicy-tftp/CMakeLists.txt
spicy-tftp/COPYING
spicy-tftp/README
spicy-tftp/analyzer/CMakeLists.txt
spicy-tftp/analyzer/tftp.evt
spicy-tftp/analyzer/tftp.spicy
spicy-tftp/cmake/FindSpicyPlugin.cmake
spicy-tftp/scripts/__load__.zeek
```

(continues on next page)

(continued from previous page)

```
spicy-tftp/scripts/dpd.sig
spicy-tftp/scripts/main.zeek
spicy-tftp/testing/Baseline/tests.run-pcap/conn.log
spicy-tftp/testing/Baseline/tests.run-pcap/output
spicy-tftp/testing/Baseline/tests.standalone/
spicy-tftp/testing/Baseline/tests.standalone/output
spicy-tftp/testing/Baseline/tests.trace/output
spicy-tftp/testing/Baseline/tests.trace/tftp.log
spicy-tftp/testing/Files/random.seed
spicy-tftp/testing/Makefile
spicy-tftp/testing/Scripts/README
spicy-tftp/testing/Scripts/diff-remove-timestamps
spicy-tftp/testing/Scripts/get-zeek-env
spicy-tftp/testing/Traces/tcp-port-12345.pcap
spicy-tftp/testing/Traces/udp-port-12345.pcap
spicy-tftp/testing/btest.cfg
spicy-tftp/testing/tests/availability.zeek
spicy-tftp/testing/tests/standalone.spicy
spicy-tftp/testing/tests/trace.zeek
spicy-tftp/zkg.meta
```

Note the `*.evt`, `*.spicy`, `*.zeek` files: they correspond to the files we created for TFTP in the preceding sections; we can just move our versions in there. Furthermore, the generated scaffolding marks places with `TODO` that need manual editing: use `git grep TODO` inside the `spicy-tftp` directory to find them. We won't go through all the specific customizations for TFTP here, but for reference you can find the full TFTP package as created from the *zkg* template on GitHub.

If instead of a protocol analyzer, you'd like to create a file or packet analyzer, run zkg with `--features spicy-file-analyzer` or `--features spicy-packet-analyzer`, respectively. The generated skeleton will be suitably adjusted then.

2.4.4 Next Steps

This tutorial provides an introduction to the Spicy language and toolchain. Spicy's capabilities go much further than what we could show here. Some pointers for what to look at next:

- *Programming in Spicy* provides an in-depth discussion of the Spicy language, including in particular all the constructs for *parsing data* and a *reference of language elements*. Note that most of Spicy's *types* come with operators and methods for operating on values. The *Debugging* section helps understanding Spicy's operation if results do not match what you would expect.
- *Examples* summarizes grammars coming with the Spicy distribution.
- *Zeek Integration* discusses Spicy's integration into Zeek.

2.5 Programming in Spicy

This chapter summarizes the main concepts of writing Spicy grammars. We begin with a deep-dive on Spicy's main task, parsing: We walk through all the corresponding constructs & mechanisms available to grammar writers, without paying too much attention to other specifics of the Spicy language, such as data types and control flow constructs. Much of that should be sufficiently intuitive to readers familiar with standard scripting languages. However, once we finish the parsing section, we follow up with a comprehensive overview of the Spicy language, as well as a reference of pre-built library functionality that Spicy grammars can leverage.

2.5.1 Parsing

Basics

Type Declaration

Spicy expresses units of data to parse through a type called, appropriately, `unit`. At a high level, a unit is similar to structs or records in other languages: It defines an ordered set of fields, each with a name and a type, that during runtime will store corresponding values. Units can be instantiated, fields can be assigned values, and these values can then be retrieved. Here's about the most basic Spicy unit one can define:

```
type Foo = unit {
    version: uint32;
};
```

We name the type `Foo`, and it has just one field called `version`, which stores a 32-bit unsigned integer type.

Leaving parsing aside for a moment, we can indeed use this type similar to a typical struct/record type:

```
module Test;

type Foo = unit {
    version: uint32;
};

global f: Foo;
f.version = 42;
print f;
```

This will print:

```
[$version=42]
```

Fields are initially unset, and attempting to read an unset field will trigger a *runtime exception*. You may, however, provide a default value by adding a `&default` *attribute* to the field, in which case that will be returned on access if no value has been explicitly assigned:

```
module Test;

type Foo = unit {
    version: uint32 &default=42;
};

global f: Foo;
print f;
print "version is %s" % f.version;
```

This will print:

```
[$version=(not set)]
version is 42
```

Note how the field remains unset even with the default now specified, while the access returns the expected value.

Parsing a Field

We can turn this minimal unit type into a starting point for parsing data—in this case a 32-bit integer from four bytes of raw input. First, we need to declare the unit as `public` to make it accessible from outside of the current module—a requirement if a host application wants to use the unit as a parsing entry point.

```
module Test;

public type Foo = unit {
    version: uint32;

    on %done {
        print "0x%x" % self.version;
    }
};
```

Let's use *spicy-driver* to parse 4 bytes of input through this unit:

```
# printf '\01\02\03\04' | spicy-driver foo.spicy
0x1020304
```

The output comes of course from the `print` statement inside the `%done` hook, which executes once the unit has been fully parsed. (We will discuss unit hooks further below.)

By default, Spicy assumes integers that it parses to be represented in network byte order (i.e., big-endian), hence the output above. Alternatively, we can tell the parser through an attribute that our input is arriving in, say, little-endian instead. To do that, we import the `spicy` library module, which provides an enum type *spicy::ByteOrder* that we can give to a `&byte-order` field attribute for fields that support it:

```
module Test;

import spicy;

public type Foo = unit {
    version: uint32 &byte-order=spicy::ByteOrder::Little;

    on %done {
        print "0x%x" % self.version;
    }
};
```

```
# printf '\01\02\03\04' | spicy-driver foo.spicy
0x4030201
```

We see that unpacking the value has now flipped the bytes before storing it in the `version` field.

Similar to `&byte-order`, Spicy offers a variety of further attributes that control the specifics of how fields are parsed. We'll discuss them in the relevant sections throughout the rest of this chapter.

Non-type Fields

Unit fields always have a type. However, in some cases a field's type is not explicitly declared, but derived from what's being parsed. The main example of this is parsing a constant value: Instead of a type, a field can specify a constant of a parseable type. The field's type will then (usually) just correspond to the constant's type, and parsing will expect to find the corresponding value in the input stream. If a different value gets unpacked instead, parsing will abort with an error. Example:

```
module Test;

public type Foo = unit {
    bar: b"bar";
    on %done { print self.bar; }
};
```

```
# printf 'bar' | spicy-driver foo.spicy
bar
```

```
# printf 'foo' | spicy-driver foo.spicy
[fatal error] terminating with uncaught exception of type spicy::rt::ParseError:␣
↪parse error: expecting 'bar' (foo.spicy:5)
```

Regular expressions extend this scheme a bit further: If a field specifies a regular expression constant rather than a type, the field will have type *Bytes* and store the data that ends up matching the regular expression:

```
module Test;

public type Foo = unit {
    x: /Foo.*Bar/;
    on %done { print self; }
};
```

```
# printf 'Foo12345Bar' | spicy-driver foo.spicy
[$x=b"Foo12345Bar"]
```

There's also a programmatic way to change a field's type to something that's different than what's being parsed, see the *&convert attribute*.

Parsing Fields With Known Size

You can limit the input that a field receives by attaching a `&size=EXPR` attribute that specifies the number of raw bytes to make available. This works on top of any other attributes that control the field's parsing. From the field's perspective, such a size limit acts just like reaching the end of the input stream at the specified position. Example:

```
module Test;

public type Foo = unit {
    x: int16[] &size=6;
    y: bytes &eod;
    on %done { print self; }
};
```

```
# printf '\000\001\000\002\000\003xyz' | spicy-driver foo.spicy
[$x=[1, 2, 3], $y=b"xyz"]
```

As you can see, x receives 6 bytes of input, which it then turns into three 16-bit integers.

Normally, the field must consume all the bytes specified by &size, otherwise a parse error will be triggered. Some types support an additional &eod attribute to lift this restrictions; we discuss that in the corresponding type's section where applicable.

After a field with a &size=EXPR attribute, parsing will always move ahead the full amount of bytes, even if the field did not consume them all.

Todo: Parsing a regular expression would make a nice example for &size as well.

Defensively Limiting Input Size

On their own, parsers place no intrinsic upper limit on the size of variable-size fields or units. This can have negative effects like out-of-memory errors, e.g., when available memory is constrained, or for malformed input.

As a defensive mechanism you can put an upper limit on the data a field or unit receives by attaching a &max-size=EXPR attribute where EXPR is an unsigned integer specifying the upper limit of number of raw bytes a field or unit should receive. If more than &max-size bytes are consumed during parsing, an error will be triggered. This attribute works on top of any other attributes that control parsing. Example:

```
module Test;

public type Foo = unit {
    x: bytes &until=b"\x00" &max-size=1024;
    on %done { print self; }
};
```

```
# printf '\001\002\003\004\005\000' | spicy-driver foo.spicy
[$x=b"\x01\x02\x03\x04\x05"]
```

Here x will parse a NULL-terminated byte sequence (excluding the terminating NULL), but never more than 1024 bytes.

&max-size cannot be combined with &size.

Anonymous Fields

Field names are optional. If skipped, the field becomes an *anonymous* field. These still participate in parsing as any other field, but they won't store any value, nor is there a way to get access to them from outside. You can however still get to the parsed value inside a corresponding field hook (see *Unit Hooks*) using the reserved $$ identifier (see *Reserved Identifiers*).

```
module Test;

public type Foo = unit {
    x: int8;
     : int8 { print $$; } # anonymous field
    y: int8;
    on %done { print self; }
};
```

```
# printf '\01\02\03' | spicy-driver foo.spicy
2
[$x=1, $y=3]
```

Reserved Identifiers

Inside units, two reserved identifiers provide access to values currently being parsed:

self Inside a unit's type definition, `self` refers to the unit instance that's currently being processed. The instance is writable and maybe modified by assigning to any fields of `self`.

$$ Inside field attributes and hooks, `$$` refers to the just parsed value, even if it's not going to be directly stored in the field. The value of `$$` is writable and may be modified.

On-the-fly Type Conversion with &convert

Fields may use an attribute `&convert=EXPR` to transform the value that was just being parsed before storing it as the field's final value. With the attribute being present, it's the value of `EXPR` that's stored in the field, not the parsed value. Accordingly, the field's type also changes to the type of `EXPR`.

Typically, `EXPR` will use `$$` to access the value actually being parsed and then transform it into the desired representation. For example, the following stores an integer parsed in an ASCII representation as a `uint64`:

```
module Test;

import spicy;

public type Foo = unit {
    x: bytes &eod &convert=$$.to_uint();
    on %done { print self; }
};
```

```
# printf 12345 | spicy-driver foo.spicy
[$x=12345]
```

`&convert` also works at the unit level to transform a whole instance into a different value after it has been parsed:

```
module Test;

type Data = unit {
    data: bytes &size=2;
} &convert=self.data.to_int();

public type Foo = unit {
    numbers: Data[];

    on %done { print self.numbers; }
};
```

```
# printf 12345678 | spicy-driver foo.spicy
[12, 34, 56, 78]
```

Note how the `Data` instances have been turned into integers. Without the `&convert` attribute, the output would have looked like this:

```
[[$data=b"12"], [$data=b"34"], [$data=b"56"], [$data=b"78"]]
```

Enforcing Parsing Constraints

Fields may use an attribute `&requires=EXPR` to enforce additional constraints on their values. `EXPR` must be a boolean expression that will be evaluated after the parsing for the field has finished, but before any hooks execute. If `EXPR` returns `False`, the parsing process will abort with an error, just as if the field had been unparsable in the first place (incl. executing any *%error* hooks). `EXPR` has access to the parsed value through *$$*. It may also retrieve the field's final value through `self.<field>`, which can be helpful when *&convert* is present.

Example:

```
module Test;

import spicy;

public type Foo = unit {
    x: int8 &requires=($$ < 5);
    on %done { print self; }
};
```

```
# printf '\001' | spicy-driver foo.spicy
[$x=1]
```

```
# printf '\010' | spicy-driver foo.spicy
[fatal error] terminating with uncaught exception of type spicy::rt::ParseError:␣
↪parse error: &required failed ($$ == 8) (foo.spicy:7:13)
```

One can also enforce conditions globally at the unit level through a attribute `&requires = EXPR`. `EXPR` will be evaluated once the unit has been fully parsed, but before any `%done` hook executes. If `EXPR` returns `False`, the unit's parsing process will abort with an error. As usual, `EXPR` has access to the parsed instance through `self`. More than one `&requires` attribute may be specified.

Example:

```
module Test;

import spicy;

public type Foo = unit {
    x: int8;
    on %done { print self; }
} &requires = self.x < 5;
```

```
# printf '\001' | spicy-driver foo.spicy
[$x=1]
```

```
# printf '\010' | spicy-driver foo.spicy
[error] terminating with uncaught exception of type spicy::rt::ParseError: parse␣
↪error: &requires failed (foo.spicy:9:15)
```

Unit Hooks

Unit hooks provide one of the most powerful Spicy tools to control parsing, track state, and retrieve results. Generally, hooks are blocks of code triggered to execute at certain points during parsing, with access to the current unit instance.

Conceptually, unit hooks are somewhat similar to methods: They have bodies that execute when triggered, and these bodies may receive a set of parameters as input. Different from functions, however, a hook can have more than one body. If multiple implementations are provided for the same hook, all of them will execute successively. A hook may also not have any body implemented at all, in which case there's nothing to do when it executes.

The most commonly used hooks are:

`on %init() { ... }` Executes just before unit parsing will start.

`on %done { ... }` Executes just after unit parsing has completed successfully.

`on %error { ... }` or `on %error(msg: string) { ... }` Executes when a parse error has been encountered, just before the parser either aborts processing. If the second form is used, a description of the error will be provided through the string argument.

`on %finally { ... }` Executes once unit parsing has completed in any way. This hook is most useful to modify global state that needs to be updated no matter the success of the parsing process. Once *%init* triggers, this hook is guaranteed to eventually execute as well. It will run *after* either `%done` or `%error`, respectively. (If a new error occurs during execution of `%finally`, that will not trigger the unit's `%error` hook.)

`on %print { ... }` Executes when a unit is about to be printed (and more generally: when rendered into a string representation). By default, printing a unit will produce a list of its fields with their current values. Through this hook, a unit can customize its appearance by returning the desired string.

`on <field name> { ... }` (field hook) Executes just after the given unit field has been parsed. The parsed value is accessible through the `$$`, potentially with any relevant type conversion applied (see *On-the-fly Type Conversion with &convert*). The same will also have been assigned to the field already.

`on <field name> foreach { ... }` (container hook) Assuming the specified field is a container (e.g., a vector), this executes each time a new container element has been parsed, and just before it's been added to the container. The parsed element is accessible through the `$$` identifier, and can be modified before it's stored. The hook implementation may also use the *stop* statement to abort container parsing, without the current element being added anymore.

In addition, Spicy provides a set of hooks specific to the `sink` type which are discussed in the *section on sinks*, and hooks which are executed during *error recovery*.

There are three locations where hooks can be implemented:

- Inside a unit, `on <hook name> { ... }` implements the hook of the given name:

```
type Foo = unit {
    x: uint32;
    v: uint8[];

    on %init { ... }
    on x { ... }
    on v foreach { ... }
    on %done { ... }
}
```

- Field and container hooks may be directly attached to their field, skipping the `on ...` part:

```
type Foo = unit {
    x: uint32 { ... }
```

(continues on next page)

(continued from previous page)

```
    v: uint8[] foreach { ... }
}
```

- At the global module level, one can add hooks to any available unit type through `on <unit type>::<hook name> { ... }`. With the definition of `Foo` above, this implements hooks externally:

  ```
  on Foo::%init { ... }
  on Foo::x { ... }
  on Foo::v foreach { ... }
  on Foo::%done { ... }
  ```

  External hooks work across module boundaries by qualifying the unit type accordingly. They provide a powerful mechanism to extend a predefined unit without changing any of its code.

If multiple implementations are provided for the same hook, by default it remains undefined in which order they will execute. If a particular order is desired, you can specify priorities for your hook implementations:

```
on Foo::v priority=5 { ... }
on Foo::v priority=-5 { ... }
```

Implementations then execute in order of their priorities: The higher a priority value, the earlier it will execute. If not specified, a hook's priority is implicitly taken as zero.

Note: When a hook executes, it has access to the current unit instance through the `self` identifier. The state of that instance will reflect where parsing is at that time. In particular, any field that hasn't been parsed yet, will remain unset. You can use the `?.` unit operator to test if a field has received a value yet.

Unit Variables

In addition to unit field for parsing, you can also add further instance variables to a unit type to store arbitrary state:

```
module Test;

public type Foo = unit {
    on %init { print self; }
    x: int8 { self.a = "Our integer is %d" % $$; }
    on %done { print self; }

    var a: string;
};
```

```
# printf \05 | spicy-driver foo.spicy
[$x=(not set), $a=""]
[$x=48, $a="Our integer is 48"]
```

Here, we assign a string value to `a` once we have parsed `x`. The final `print` shows the expected value. As you can also see, before we assign anything, the variable's value is just empty: Spicy initializes unit variables with well-defined defaults. If you would rather leave a variable unset by default, you can add `&optional`:

```
module Test;

public type Foo = unit {
    on %init { print self; }
```

(continues on next page)

(continued from previous page)

```
    x: int8 { self.a = "Our integer is %d" % $$; }
    on %done { print self; }

    var a: string &optional;
};
```

```
# printf \05 | spicy-driver foo.spicy
[$x=(not set), $a=(not set)]
[$x=48, $a="Our integer is 48"]
```

You can use the `?.` unit operator to test if an optional unit variable remains unset, e.g., `self?.x` would return `True` if field `x` is set and `False` otherwise.

Unit variables can also be initialized with custom expressions when being defined. The initialization is performed just before the containing unit starts parsing (implying that the expressions cannot access parse results of the unit itself yet)

```
module Test;

public type Foo = unit {
    x: int8;
    var a: int8 = 123;
    on %done { print self; }
};
```

```
# printf \05 | spicy-driver foo.spicy
[$x=48, $a=123]
```

Unit Parameters

Unit types can receive parameters upon instantiation, which will then be available to any code inside the type's declaration:

```
module Test;

type Bar = unit(msg: string, mult: int8) {
    x: int8 &convert=($$ * mult);
    on %done { print "%s: %d" % (msg, self.x); }
};

public type Foo = unit {
    y: Bar("My multiplied integer", 5);
};
```

```
# printf '\05' | spicy-driver foo.spicy
My multiplied integer: 25
```

This example shows a typical idiom: We're handing parameters down to a subunit through parameters it receives. Inside the submodule, we then have access to the values passed in.

Note: It's usually not very useful to define a top-level parsing unit with parameters because we don't have a way to pass anything in through `spicy-driver`. A custom host application could make use of them, though.

This works with subunits inside containers as well:

```
module Test;

type Bar = unit(mult: int8) {
    x: int8 &convert=($$ * mult);
    on %done { print self.x; }
};

public type Foo = unit {
    x: int8;
    y: Bar(self.x)[];
};
```

```
# printf '\05\01\02\03' | spicy-driver foo.spicy
5
10
15
```

Unit parameters follow the same passing conventions as *function parameters*. In particular, they are read-only by default. If the subunit wants to modify a parameter it receives, it needs to be declared as `inout` (e.g., `Bar(inout foo: Foo)`

Note: `inout` parameters need to be reference types which holds for other units types, but currently not for basic types (#%s674). In order to pass a basic type as unit parameter it needs to be declared as a reference (e.g., `string&`) and explicitly wrapped when being set:

```
module Test;

type X = unit(inout msg: string&) {
    n : uint8 {
      local s = "Parsed %d" % $$;
      msg = new s;
    }
};

global msg = new "Nothing parsed, yet";

public type Y = unit {
    x: X(msg);
    on %done { print msg; }
};
```

```
# printf '\x2a' | spicy-driver foo.spicy
Parsed 42
```

Note: A common use-case for unit parameters is passing the `self` of a higher-level unit down into a subunit:

```
type Foo = unit {
    ...
    b: Bar(self);
    ...
}

type Bar = unit(foo: Foo) {
```

(continues on next page)

(continued from previous page)

```
    # We now have access to any state in "foo".
}
```

That way, the subunit can for example store state directly in the parent.

Unit Attributes

Unit types support the following type attributes:

&byte-order=ORDER Specifies a byte order to use for parsing the unit where `ORDER` is of type *spicy::ByteOrder*. This overrides the byte order specified for the module. Individual fields can override this value by specifying their own byte-order. Example:

```
type Foo = unit {
    version: uint32;
} &byte-order=spicy::ByteOrder::Little;
```

&convert=EXPR Replaces a unit instance with the result of the expression `EXPR` after parsing it from inside a parent unit. See *On-the-fly Type Conversion with &convert* for an example. `EXPR` has access to `self` to retrieves state from the unit.

&requires=EXPR

Enforces post-conditions on the parsed unit. `EXPR` must be a boolean expression that will be evaluated after the parsing for the unit has finished, but before any hooks execute. More than one `&requires` attributes may be specified. Example:

```
type Foo = unit {
    a: int8;
    b: int8;
} &requires=self.a==self.b;
```

See the *section on parsing constraints* for more details.

&size=N Limits the unit's input to `N` bytes, which it must fully consume. Example:

```
type Foo = unit {
    a: int8;
    b: bytes &eod;
} &size=5;
```

This expects 5 bytes of input when parsing an instance of `Foo`. The unit will store the first byte into `a`, and then fill `b` with the remaining 4 bytes.

The expression `N` has access to `self` as well as to the unit's parameters.

Meta data

Units can provide meta data about their semantics through *properties* that both Spicy itself and host applications can access. One defines properties inside the unit's type through either a `%<property> = <value>;` tuple, or just as `%<property>;` if the property does not take an argument. Currently, units support the following meta data properties:

`%mime-type = STRING` A string of the form `"<type>/<subtype>"` that defines the MIME type for content the unit knows how to parse. This may include a `*` wildcard for either the type or subtype. We use a generalized notion of MIME types here that can include custom meanings. See *Sinks* for more on how these MIME types are used to select parsers dynamically during runtime.

You can specify this property more than once to associate a unit with multiple types.

`%description = STRING` A short textual description of the unit type (i.e., the parser that it defines). Host applications have access to this property, and `spicy-driver` includes the information into the list of available parsers that it prints with the `--list-parsers` option.

`%port = PORT_VALUE [&originator|&responder]` A *Port* to associate this unit with, optionally including a direction to limit its use to the corresponding side. This property has no built-in effect, but host applications may make use of the information to decide which unit type to use for parsing a connection's payload.

`%skip = ( REGEXP | Null );` Specifies a pattern which should be skipped when encountered in the input stream in between parsing of unit fields. This overwrites a value set at the module level; use `Null` to reset the property, i.e., not skip anything.

`%skip-pre = ( REGEXP | Null );` Specifies a pattern which should be skipped when encountered in the input stream before parsing of a unit begins. This overwrites a value set at the module level; use `Null` to reset the property, i.e., not skip anything.

`%skip-post = ( REGEXP | Null );` Specifies a pattern which should be skipped when encountered in the input stream after parsing of a unit has finished. This overwrites a value set at the module level; use `Null` to reset the property, i.e., not skip anything.

`%synchronize-at = EXPR;` Specifies a literal to synchronize on if the unit is used as a synchronization point during *error recovery*. The literal is left in the input stream.

`%synchronize-after = EXPR;` Specifies a literal to synchronize on if the unit is used as a synchronization point during *error recovery*. The literal is consumed and will not be present in the input stream after successful synchronization.

Units support some further properties for other purposes, which we introduce in the corresponding sections.

Parsing Types

Several, but not all, of Spicy's *data types* can be parsed from binary data. In the following we summarize the types that can, along with any options they support to control specifics of how they unpack binary representations.

Address

Spicy parses *addresses* from either 4 bytes of input for IPv4 addresses, or 16 bytes for IPv6 addresses. To select the type, a unit field of type `addr` must come with either an `&ipv4` or `&ipv6` attribute.

By default, addresses are assumed to be represented in network byte order. Alternatively, a different byte order can be specified through a `&byte-order` attribute specifying the desired *spicy::ByteOrder*.

Example:

```
module Test;

import spicy;

public type Foo = unit {
    ip: addr &ipv6 &byte-order=spicy::ByteOrder::Little;
    on %done { print self; }
};
```

```
# printf '1234567890123456' | spicy-driver foo.spicy
[$ip=3635:3433:3231:3039:3837:3635:3433:3231]
```

Bitfield

Bitfields parse an integer value of a given size, and then make selected smaller bit ranges within that value available individually through dedicated identifiers. For example, the following unit parses 4 bytes as an `uint32` and then makes the value of bit 0 available as `f.x1`, bits 1 to 2 as `f.x2`, and bits 3 to 4 as `f.x3`, respectively:

```
module Test;

public type Foo = unit {
    f: bitfield(32) {
        x1: 0;
        x2: 1..2;
        x3: 3..4;
    };

    on %done {
        print self.f.x1, self.f.x2, self.f.x3;
        print self;
    }
};
```

```
# printf '\01\02\03\04' | spicy-driver foo.spicy
0, 2, 0
[$f=(0, 2, 0)]
```

Generally, a field `bitfield(N)` field is parsed like an `uint<N>`. The field then supports dereferencing individual bit ranges through their labels. The corresponding expressions (`self.x.<id>`) have the same `uint<N>` type as the parsed value itself, with the value shifted to the right so that the lowest extracted bit becomes bit 0 of the returned value. As you can see in the example, the type of the field itself becomes a tuple composed of the values of the individual bit ranges.

By default, a bitfield assumes the underlying integer comes in network byte order. You can specify a `&byte-order` attribute to change that (e.g., `bitfield(32) { ... } &byte-order=spicy::ByteOrder::Little`). Furthermore, each bit range can also specify a `&bit-order` attribute to specify the *ordering* for its bits; the default is `spicy::BitOrder::LSB0`.

The individual bit ranges support the `&convert` attribute and will adjust their types accordingly, just like a regular unit field (see *On-the-fly Type Conversion with &convert*). For example, that allows for mapping a bit range to an enum, using `$$` to access the parsed value:

```
module Test;

import spicy;

type X = enum { A = 1, B = 2 };

public type Foo = unit {
    f: bitfield(8) {
        x1: 0..3 &convert=X($$);
        x2: 4..7 &convert=X($$);
    } { print self.f.x1, self.f.x2; }
};
```

```
# printf '\x21' | spicy-driver foo.spicy
X::A, X::B
```

Bytes

When parsing a field of type *Bytes*, Spicy will consume raw input bytes according to a specified attribute that determines when to stop. The following attributes are supported:

&eod Consumes all subsequent data until the end of the input is reached.

&size=N Consumes exactly `N` bytes. The attribute may be combined with `&eod` to consume up to `N` bytes instead (i.e., permit running out of input before the size limit is reached).

(This attribute *works for fields of all types*. We list it here because it's particularly common to use it with *bytes*.)

&until=DELIM Consumes bytes until the specified delimiter is found. `DELIM` must be of type `bytes` itself. The delimiter will not be included into the resulting value, but consumed.

&until-including=DELIM Similar to `&until`, but this does include the delimiter `DELIM` into the resulting value.

At least one of these attributes must be provided.

On top of that, bytes fields support the attribute `&chunked` to change how the parsed data is processed and stored. Normally, a bytes field will first accumulate all desired data and then store the final, complete value in the field. With `&chunked`, if the data arrives incrementally in pieces, the field instead processes just whatever is available at a time, storing each piece directly, and individually, in the field. Each time a piece gets stored, any associated field hooks execute with the new part as their `$$`. Parsing with `&chunked` will eventually still consume the same number of bytes overall, but it avoids buffering everything in cases where that's either infeasible or simply not not needed.

Bytes fields support parsing constants: If a `bytes` constant is specified instead of a field type, parsing will expect to find the corresponding value in the input stream.

Integer

Fields of *integer type* can be either signed (`intN`) or unsigned (`uintN`). In either case, the bit length `N` determines the number of bytes being parsed. By default, integers are expected to come in network byte order. You can specify a different order through the `&byte-order=ORDER` attribute, where `ORDER` is of type *spicy::ByteOrder*.

Integer fields support parsing constants: If an integer constant is specified instead the instead of a field type, parsing will expect to find the corresponding value in the input stream. Since the exact type of the integer constant is important, you should use their constructor syntax to make that explicit (e.g., `uint32(42)`, `int8(-1)`; vs. using just `42` or `-1`).

Real

Real values are parsed as either single or double precision values in IEEE754 format, depending on the value of their `&type=T` attribute, where `T` is one of *spicy::RealType*.

Regular Expression

When parsing a field through a *Regular Expression*, the expression is expected to match at the current position of the input stream. The field's type becomes `bytes`, and it will store the matching data.

Inside hooks for fields with regular expressions, you can access capture groups through `$1`, `$2`, `$3`, etc. For example:

```
x : /(a.c)(de*f)(h.j)/ {
    print $1, $2, $3;
    }
```

This will print out the relevant pieces of the data matching the corresponding set of parentheses. (There's no `$0`, just use `$$` as normal to get the full match.)

Matching an regular expression is more expensive if you need it to capture groups. If are using groups inside your expression but don't need the actual captures, add `&nosub` to the field to remove that overhead.

Unit

Fields can have the type of another unit, in which case parsing will descend into that subunit's grammar until that instance has been fully parsed. Field initialization and hooks work as usual.

If the subunit receives parameters, they must be given right after the type.

```
module Test;

type Bar = unit(a: string) {
    x: uint8 { print "%s: %u" % (a, self.x); }
};

public type Foo = unit {
    y: Bar("Spicy");
    on %done { print self; }
};
```

```
# printf '\01\02' | spicy-driver foo.spicy
Spicy: 1
[$y=[$x=1]]
```

See *Unit Parameters* for more.

Vector

Parsing a *vector* creates a loop that repeatedly parses elements of the specified type from the input stream until an end condition is reached. The field's value accumulates all the elements into the final vector.

Spicy uses a specific syntax to define fields of type vector:

```
NAME : ELEM_TYPE[SIZE]
```

`NAME` is the field name as usual. `ELEM_TYPE` is type of the vector's elements, i.e., the type that will be repeatedly parsed. `SIZE` is the number of elements to parse into the vector; this is an arbitrary Spicy expression yielding an integer value. The resulting field type then will be `vector<ELEM_TYPE>`. Here's a simple example parsing five `uint8`:

```
module Test;

public type Foo = unit {
    x: uint8[5];
    on %done { print self; }
};
```

```
# printf '\01\02\03\04\05' | spicy-driver foo.spicy
[$x=[1, 2, 3, 4, 5]]
```

It is possible to skip the `SIZE` (e.g., `x: uint8[]`) and instead use another kind of end conditions to terminate a vector's parsing loop. To that end, vectors support the following attributes:

`&eod` Parses elements until the end of the input stream is reached.

`&size=N` Parses the vector from the subsequent `N` bytes of input data. This effectively limits the available input to the corresponding window, letting the vector parse elements until it runs out of data. (This attribute *works for fields of all types*. We list it here because it's particularly common to use it with vectors.)

`&until=EXPR` Vector elements are parsed in a loop with `EXPR` being evaluated as a boolean expression after each parsed element, and before adding the element to the vector. Once `EXPR` evaluates to true, parsing stops *without* adding the element that was just parsed.

`&until-including=EXPR` Similar to `&until`, but does include the final element `EXPR` into the field's vector when stopping parsing.

`&while=EXPR` Continues parsing as long as the boolean expression `EXPR` evaluates to true.

If neither a size nor an attribute is given, Spicy will attempt to use *look-ahead parsing* to determine the end of the vector based on the next expected token. Depending on the unit's field, this may not be possible, in which case Spicy will decline to compile the unit.

The syntax shown above generally works for all element types, including subunits (e.g., `x: MyUnit[]`).

Note: The `x: (<T>)[]` syntax is quite flexible. In fact, `<T>` is not limited to subunits, but allows for any standard field specification defining how to parse the vector elements. For example, `x: (bytes &size=5)[];` parses a vector of 5-character `bytes` instances.

When parsing a vector, Spicy supports using a special kind of field hook, `foreach`, that executes for each parsed element individually. Inside that hook, `$$` refers to the just parsed element:

```
module Test;

public type Foo = unit {
    x: uint8[5] foreach { print $$, self.x; }
};
```

```
# printf '\01\02\03\04\05' | spicy-driver foo.spicy
1, []
2, [1]
3, [1, 2]
4, [1, 2, 3]
5, [1, 2, 3, 4]
```

As you can see, when a `foreach` hook executes the element has not yet been added to the vector. You may indeed use a `stop` statement inside a `foreach` hook to abort the vector's parsing without adding the current element anymore. See *Unit Hooks* for more on hooks.

Void

The *Void* type can be used as a place-holder for not storing any data. By default `void` fields do not consume any data, and while not very useful for normal fields, this allows branches in *switch* constructs to forego any parsing.

If a non-zero `&size` is specified, the given number of bytes of input data are consumed. This allows skipping over data without storing their result:

```
module Test;

public type Foo = unit {
    : void &size=2;
    x: uint8;

    on %done { print self; }
};
```

```
# printf '\01\02\03' | spicy-driver foo.spicy
[$x=3]
```

A `void` field can also terminate through an `&until=<BYTES>` attribute: it then skips all input data until the given deliminator sequence of bytes is encountered. The deliminator is extracted from the stream before parsing continues.

Finally, a `void` field can specify `&eod` to consume all data until the end of the current input.

`void` fields cannot have names.

Controlling Parsing

Spicy offers a few additional constructs inside a unit's declaration for steering the parsing process. We discuss them in the following.

Conditional Parsing

A unit field may be conditionally skipped for parsing by adding an `if ( COND )` clause, where `COND` is a boolean expression. The field will be only parsed if the expression evaluates to true at the time the field is next in line.

```
module Test;

public type Foo = unit {
    a: int8;
    b: int8 if ( self.a == 1 );
    c: int8 if ( self.a % 2 == 0 );
    d: int8;

    on %done { print self; }
};
```

```
# printf '\01\02\03\04' | spicy-driver foo.spicy
[$a=1, $b=2, $c=(not set), $d=3]

# printf '\02\02\03\04' | spicy-driver foo.spicy
[$a=2, $b=(not set), $c=2, $d=3]
```

Look-Ahead

Internally, Spicy builds an LR(1) grammar for each unit that it parses, meaning that it can actually look *ahead* in the parsing stream to determine how to process the current input location. Roughly speaking, if (1) the current construct does not have a clear end condition defined (such as a specific length), and (2) a specific value is expected to be found next; then the parser will keep looking for that value and end the current construct once it finds it.

“Construct” deliberately remains a bit of a fuzzy term here, but think of vector parsing as the most common instance of this: If you don’t give a vector an explicit termination condition (as discussed in *Vector*), Spicy will look at what’s expected to come *after* the container. As long as that’s something clearly recognizable (e.g., a specific value of an atomic type, or a match for a regular expression), it’ll terminate the vector accordingly.

Here’s an example:

```
module Test;

public type Foo = unit {
    data: uint8[];
        : /EOD/;
    x   : int8;

    on %done { print self; }
};
```

```
# printf '\01\02\03EOD\04' | spicy-driver foo.spicy
[$data=[1, 2, 3], $x=4]
```

For vectors, Spicy attempts look-ahead parsing automatically as a last resort when it doesn’t find more explicit instructions. However, it will reject a unit if it can’t find a suitable look-ahead symbol to work with. If we had written `int32` in the example above, that would not have worked as the parser can’t recognize when there’s a `int32` coming; it would need to be a concrete value, such as `int32(42)`.

See the *switch* construct for another instance of look-ahead parsing.

switch

Spicy supports a `switch` construct as way to branch into one of several parsing alternatives. There are two variants of this, an explicit branch and one driving by look-ahead:

Branch by expression

The most basic form of switching by expression looks like this:

```
switch ( EXPR ) {
    VALUE_1 -> FIELD_1;
    VALUE_2 -> FIELD_2;
    ...
    VALUE_N -> FIELD_N;
};
```

This evaluates `EXPR` at the time parsing reaches the `switch`. If there's a `VALUE` matching the result, parsing continues with the corresponding field, and then proceeds with whatever comes after the switch. Example:

```
module Test;

public type Foo = unit {
    x: bytes &size=1;
    switch ( self.x ) {
        b"A" -> a8: int8;
        b"B" -> a16: int16;
        b"C" -> a32: int32;
    };

    on %done { print self; }
};
```

```
# printf 'A\01' | spicy-driver foo.spicy
[$x=b"A", $a8=1, $a16=(not set), $a32=(not set)]

# printf 'B\01\02' | spicy-driver foo.spicy
[$x=b"B", $a8=(not set), $a16=258, $a32=(not set)]
```

We see in the output that all of the alternatives turn into normal unit members, with all but the one for the branch that was taken left unset.

If none of the values match the expression, that's considered a parsing error and processing will abort. Alternative, one can add a default alternative by using `*` as the value. The branch will then be taken whenever no other value matches.

A couple additional notes about the fields inside an alternative:

- In our example, the fields of all alternatives all have different names, and they all show up in the output. One can also reuse names across alternatives as long as the types exactly match. In that case, the unit will end up with only a single instance of that member.
- An alternative can match against more than one value by separating them with commas (e.g., `b"A", b"B" -> x: int8;`).
- Alternatives can have more than one field attached by enclosing them in braces, i.e.,: `VALUE -> { FIELD_1a; FIELD_1b; ...; FIELD_1n; }`.
- Sometimes one really just needs the branching capability, but doesn't have any field values to store. In that case an anonymous `void` field may be helpful(e.g., `b"A" -> : void { DoSomethingHere(); }`.

Branch by look-ahead

`switch` also works without any expression as long as the presence of all the alternatives can be reliably recognized by looking ahead in the input stream:

```
module Test;

public type Foo = unit {
    switch {
        -> a: b"A";
        -> b: b"B";
        -> c: b"C";
    };

    on %done { print self; }
};
```

```
# printf 'A' | spicy-driver foo.spicy
[$a=b"A", $b=(not set), $c=(not set)]
```

While this example is a bit contrived, the mechanism becomes powerful once you have subunits that are recognizable by how they start:

```
module Test;

type A = unit {
    a: b"A";
};

type B = unit {
    b: uint16(0xffff);
};

public type Foo = unit {
    switch {
        -> a: A;
        -> b: B;
    };

    on %done { print self; }
};
```

```
# printf 'A ' | spicy-driver foo.spicy
[$a=[$a=b"A"], $b=(not set)]

# printf '\377\377' | spicy-driver foo.spicy
[$a=(not set), $b=[$b=65535]]
```

Switching Over Fields With Common Size

You can limit the input any field in a unit switch receives by attaching an optional `&size=EXPR` attribute that specifies the number of raw bytes to make available. This is analog to the *field size attribute* and especially useful to remove duplication when each case is subject to the same constraint.

```
module Test;

public type Foo = unit {
    tag: uint8;
    switch ( self.tag ) {
       1 -> b1: bytes &eod;
       2 -> b2: bytes &eod &convert=$$.lower();
    } &size=3;

    on %done { print self; }
};
```

```
# printf '\01ABC' | spicy-driver foo.spicy
[$tag=1, $b1=b"ABC", $b2=(not set)]

# printf '\02ABC' | spicy-driver foo.spicy
[$tag=2, $b1=(not set), $b2=b"abc"]
```

Backtracking

Spicy supports a simple form of manual backtracking. If a field is marked with `&try`, a later call to the unit's `backtrack()` method anywhere down in the parse tree originating at that field will immediately transfer control over to the field following the `&try`. When doing so, the data position inside the input stream will be reset to where it was when the `&try` field started its processing. Units along the original path will be left in whatever state they were at the time `backtrack()` executed (i.e., they will probably remain just partially initialized). When `backtrack()` is called on a path that involves multiple `&try` fields, control continues after the most recent.

Example:

```
module Test;

public type test = unit {
    foo: Foo &try;
    bar: Bar;

    on %done { print self; }
};

type Foo = unit {
    a: int8 {
        if ( $$ != 1 )
            self.backtrack();
       }
    b: int8;
};

type Bar = unit {
    a: int8;
    b: int8;
};
```

```
# printf '\001\002\003\004' | spicy-driver backtrack.spicy
[$foo=[$a=1, $b=2], $bar=[$a=3, $b=4]]

#  printf '\003\004' | spicy-driver backtrack.spicy
[$foo=[$a=3, $b=(not set)], $bar=[$a=3, $b=4]]
```

`backtrack()` can be called from inside *%error hooks*, so this provides a simple form of error recovery as well.

Note: This mechanism is preliminary and will probably see refinement over time, both in terms of more automated backtracking and by providing better control where to continue after backtracking.

Changing Input

By default, a Spicy parser proceeds linearly through its inputs, parsing as much as it can and yielding back to the host application once it runs out of input. There are two ways to change this linear model: diverting parsing to a different input, and random access within the current unit's data.

Parsing custom data

A unit field can have either `&parse-from=EXPR` or `&parse-at=EXPR` attached to it to change where it's receiving its data to parse from. `EXPR` is evaluated at the time the field is reached. For `&parse-from` it must produce a value of type `bytes`, which will then constitute the input for the field. This can, e.g., be used to reparse previously received input:

```
module Test;

public type Foo = unit {
    x: bytes &size=2;
    y: uint16 &parse-from=self.x;
    z: bytes &size=2;

    on %done { print self; }
};
```

```
# printf '\x01\x02\x03\04' | spicy-driver foo.spicy
[$x=b"\x01\x02", $y=258, $z=b"\x03\x04"]
```

For `&parse-at`, `EXPR` must yield an iterator pointing to (a still valid) position of the current unit's input stream (such as retrieved through *`input()`*). The field will then be parsed from the data starting at that location.

Random access

While a unit is being parsed, you may revert the current input position backwards to any location between the first byte the unit has seen and the current position. You can use a set of built-in unit methods to control the current position:

`input()` Returns a stream iterator pointing to the current input position.

`set_input()` Sets the current input position to the location of the specified stream iterator. Per above, the new position needs to reside between the beginning of the current unit's data and the current position; otherwise an exception will be generated at runtime.

`offset()` Returns the numerical offset of the current input position relative to position of the first byte fed into this unit.

position() Returns iterator to the current input position in the stream fed into this unit.

For random access, you'd typically get the current position through `input()`, subtract from it the desired number of bytes you want to back, and then use `set_input` to establish that new position. By further storing iterators as unit variables you can decouple these steps and, e.g., remember a position to later come back to.

Here's an example that parses input data twice with different sub units:

```
module Test;

public type Foo = unit {
    on %init() { self.start = self.input(); }

    a: A { self.set_input(self.start); }
    b: B;

    on %done() { print self; }

    var start: iterator<stream>;
};

type A = unit {
    x: uint32;
};

type B = unit {
    y: bytes &size=4;
};
```

```
# printf '\00\00\00\01' | spicy-driver foo.spicy
[$a=[$x=1], $b=[$y=b"\x00\x00\x00\x01"], $start=<offset=0 data=b"\x00\x00\x00\x01">]
```

If you look at output, you see that `start` iterator remembers it's offset, relative to the global input stream. It would also show the data at that offset if the parser had not already discarded that at the time we print it out.

Note: Spicy parsers discard input data as quickly as possible as parsing moves through the input stream. Indeed, that's why using random access may come with a performance penalty as the parser now needs to buffer all of unit's data until it has been fully processed.

Filters

Spicy supports attaching *filters* to units that get to preprocess and transform a unit's input before its parser gets to see it. A typical use case for this is stripping off a data encoding, such as compression or Base64.

A filter is itself just a `unit` that comes with an additional property `%filter` marking it as such. The filter unit's input represents the original input to be transformed. The filter calls an internally provided unit method *`forward()`* to pass any transformed data on to the main unit that it's attached to. The filter can call `forward` arbitrarily many times, each time forwarding a subsequent chunk of input. To attach a filter to a unit, one calls the method *`connect_filter()`* with an instance of the filter's type. Putting that all together, this is an example of a simple a filter that upper-cases all input before the main parsing unit gets to see it:

```
module Test;

type Filter = unit {
    %filter;
```

(continues on next page)

(continued from previous page)

```
    : bytes &eod &chunked {
        self.forward($$.upper());
    }
};

public type Foo = unit {
    on %init { self.connect_filter(new Filter); }
    x: bytes &size=5 { print self.x; }
};
```

```
# printf 'aBcDe' | spicy-driver foo.spicy
ABCDE
```

There are a couple of predefined filters coming with Spicy that become available by importing the `filter` library module:

`filter::Zlib` Provides zlib decompression.

`filter::Base64Decode` Provides base64 decoding.

Sinks

Sinks provide a powerful mechanism to chain multiple units together into a layered stack, each processing the output of its predecessor. A sink is the connector here that links two unit instances: one side writing and one side reading, like a Unix pipe. As additional functionality, the sink can internally reassemble data chunks that are arriving out of order before passing anything on.

Here's a basic example of two units types chained through a sink:

```
module Test;

public type A = unit {
    on %init { self.b.connect(new B); }

    length: uint8;
    data: bytes &size=self.length { self.b.write($$); }

    on %done { print "A", self; }

    sink b;
};

public type B = unit {
        : /GET /;
    path: /[^\n]+/;

    on %done { print "B", self; }
};
```

```
# printf '\13GET /a/b/c\n' | spicy-driver -p Test::A foo.spicy
B, [$path=b"/a/b/c"]
A, [$length=11, $data=b"GET /a/b/c\x0a", $b=<sink>]
```

Let's see what's going on here. First, there's `sink b` inside the declaration of `A`. That's the connector, kept as state inside `A`. When parsing for `A` is about to begin, the `%init` hook connects the sink to a new instance of `B`; that'll be the receiver for data that `A` is going to write into the sink. That writing happens inside the field hook for `data`: once we have parsed that field, we write what will go to the sink using its built-in *`write()`* method. With that write operation, the data will emerge as input for the instance of `B` that we created earlier, and that will just proceed parsing it normally. As the output shows, in the end both unit instances end up having their fields set.

As an alternative for using the *`write()`* in the example, there's some syntactic sugar for fields of type `bytes` (like `data` here): We can just replace the hook with a `->` operator to have the parsed data automatically be forwarded to the sink: `data:  bytes &size=self.length -> self.b`.

Sinks have a number of further methods, see *Sink* for the complete reference. Most of them we will also encounter in the following when discussing additional functionality that sinks provide.

Note: Because sinks are meant to decouple processing between two units, a unit connected to a sink will *not* pass any parse errors back up to the sink's parent. If you want to catch them, install an *%error* hook inside the connected unit.

Using Filters

Sinks also support *filters* to preprocess any data they receive before forwarding it on. This works just like for units by calling the built-in sink method *`connect_filter()`*. For example, if in the example above, `data` would have been gzip compressed, we could have instructed the sink to automatically decompress it by calling `self.b.connect_filter(new filter::Zlib)` (leveraging the Spicy-provided `Zlib` filter).

Leveraging MIME Types

In our example above we knew which type of unit we wanted to connect. In practice, that may or may not be the case. Often, it only becomes clear at runtime what the choice for the next layer should be, such as when using well-known ports to determine the appropriate application-layer analyzer for a TCP stream. Spicy supports dynamic selection through a generalized notion of MIME types: Units can declare which MIME types they know how to parse (see *Meta data*) , and sinks have *`connect_mime_type()`* method that will instantiate and connect any that match their argument (if that's multiple, all will be connected and all will receive the same data).

"MIME type" can mean actual MIME types, such `text/html`. Applications can, however, also define their own notion of `<type>/<subtype>` to model other semantics. For example, one could use `x-port/443` as convention to trigger parsers by well-known port. An SSL unit would then declare `%mime-type = "x-port/443`, and the connection would be established through the equivalent of `connect_mime_type("x-port/%d" % resp_port_of_connection)`.

Todo: For this specific example, there's a better solution: We also have the `%port` property and should just build up a table index on that.

Reassembly

Reassembly (or defragmentation) of out-of-order data chunks is a common requirement for many protocols. Sinks have that functionality built-in by allowing you to associate a position inside a virtual sequence space with each chunk of data. Sinks will then pass their data on to connected units only once they have collected a continuous, in-order range of bytes.

The easiest way to leverage this is to simply associate sequence numbers with each *`write()`* operation:

```
module Test;

public type Foo = unit {

    sink data;

    on %init {
        self.data.connect(new Bar);
        self.data.write(b"567", 5);
        self.data.write(b"89", 8);
        self.data.write(b"012", 0);
        self.data.write(b"34", 3);
    }
};

public type Bar = unit {
    s: bytes &eod;
    on %done { print self.s; }
};
```

```
# spicy-driver -p Test::Foo foo.spicy </dev/null
0123456789
```

By default, Spicy expects the sequence space to start at zero, so the first byte of the input stream needs to be passed in with sequence number zero. You can change that base number by calling the sink method `set_initial_sequence_number()`. You can control Spicy's gap handling, including when to stop buffering data because you know nothing further will arrive anymore. Spicy can also notify you about unsuccessful reassembly through a series of built-in unit hooks. See *Sink* for a reference of the available functionality.

Contexts

Parsing may need to retain state beyond any specific unit's lifetime. For example, a UDP protocol may want to remember information across individual packets (and hence units), or a bi-directional protocol may need to correlate the request side with the response side. One option for implementing this in Spicy is managing such state manually in *global variables*, for example by maintaining a global map that ties a unique connection ID to the information that needs to be retained. However, doing so is clearly cumbersome and error prone. As an alternative, a unit can make use of a dedicated *context* value, which is an instance of a custom type that has its lifetime determined by the host application running the parser. For example, Zeek will tie the context to the underlying connection.

Any public unit can declare a context through a unit-level property called `%context`, which takes an arbitrary type as its argument. For example:

```
public type Foo = unit {
    %context = bytes;
    [...]
};
```

When used as a top-level entry point to parsing, the unit will then, by default, receive a unique context value of that type. That context value can be accessed through the `context()` method, which will return a reference to it:

```
module Test;

public type Foo = unit {
    %context = int64;

```

(continues on next page)

(continued from previous page)

```
    on %init { print self.context(); }
};
```

```
# spicy-driver foo.spicy </dev/null
0
```

By itself, this is not very useful. However, host applications can control how contexts are maintained, and they may assign the same context value to multiple units. For example, when parsing a protocol, the *Zeek plugin* always creates a single context value shared by all top-level units belonging to the same connection, enabling parsers to maintain bi-directional, per-connection state. The batch mode of *spicy-driver* does the same.

Note: A unit's context value gets set only when a host application uses it as the top-level starting point for parsing. If in the above example *Foo* wasn't the entry point, but used inside another unit further down during the parsing process, it's context would remain unset.

As an example, the following grammar—mimicking a request/reply-style protocol—maintains a queue of outstanding textual commands to then associate numerical result codes with them as the responses come in:

```
module Test;

# We wrap the state into a tuple to make it easy to add more attributes if needed␣
↪later.
type Pending = tuple<pending: vector<bytes>>;

public type Requests = unit {
    %context = Pending;

    : Request[] foreach { self.context().pending.push_back($$.cmd); }
};

public type Replies = unit {
    %context = Pending;

    : Reply[] foreach {
        if ( |self.context().pending| ) {
            print "%s -> %s" % (self.context().pending.back(), $$.response);
            self.context().pending.pop_back();
        }
        else
            print "<missing request> -> %s", $$.response;
      }
};

type Request = unit {
    cmd: /[A-Za-z]+/;
    : b"\n";
};

type Reply = unit {
    response: /[0-9]+/;
    : b"\n";
};
```

```
# spicy-driver -F input.dat context.spicy
msg -> 100
put -> 200
CAT -> 555
end -> 300
get -> 400
LST -> 666
```

The output is produced from `this input batch file`. This would work the same when used with the Zeek plugin on a corresponding packet trace.

Note that the units for the two sides of the connection need to declare the same `%context` type. Processing will abort at runtime with a type mismatch error if that's not the case.

Error Recovery

Real world input does not always look like what parsers expect: endpoints may not conform to the protocol's specification, a parser's grammar might not fully cover all of the protocol, or some input may be missing due to packet loss or stepping into the middle of a conversation. By default, if a Spicy parser encounters such situations, it will abort parsing altogether and issue an error message. Alternatively, however, Spicy allows grammar writers to specify heuristics to recover from errors. The main challenge here is finding a spot in the subsequent input where parsing can reliably resume.

Spicy employs a two-phase approach to such recovery: it first searches for a possible point in the input stream where it seems promising to attempt to resume parsing; and then it confirms that choice by trying to parse a few fields at that location according to the grammar grammar to see if that's successful. We say that during the first part of this process, the Spicy parser is in *synchronization mode*; d during the second, it is in *trial mode*.

Phase 1: Synchronization

To identity locations where parsing can attempt to pick up again after an error, a grammar can add `&synchronize` attributes to selected unit fields, marking them as a *synchronization points*. Whenever an error occurs during parsing, Spicy will determine the closest synchronization point in the grammar following the error's location, and then attempt to continue processing there by skipping ahead in the input data until it aligns with what that field is looking for.

A synchronization point may be any of the following:

- A field for which parsing begins with a constant literal (e.g., a specific sequence of bytes). To realign the input stream, the parser will search the input for the next occurrence of this literal, discarding any data in between. Example:

  ```
  type X = unit { ... }

  type Y = unit {
      a: b"begin-of-Y";
      b: bytes &size=10;
  };

  type Foo = unit {
      x: X;
      y: Y &synchronize;
  };
  ```

  If parse error occurs during `Foo::x`, Spicy will move ahead to `Foo::y`, switch into synchronization mode, and start search the input for the bytes `begin-of-Y`. If found, it'll continue with parsing `Foo::y` at that location in trial mode (see below).

Note: Behind the scenes, synchronization through literals uses the same machinery as *look-ahead parsing*, meaning that it works across sub-units, vector content, `switch` statements, etc.. No matter how complex the field, as long as there's one or more literals that always *must* be coming first when parsing it, the field may be used as a synchronization point.

- A field with a type which specifies *%synchronize-at* or *%synchronize-after*. The parser will search the input for the next occurrence of the given literal, discarding any data in between. If the search was successful, `%synchronize-at` will leave the input at the position of the search literal for later extraction while `%synchronize-after` will discard the search literal.

  If either of these unit properties is specified, it will always overrule any other potential synchronization points in the unit. Example:

```
type X = unit {
    ...
    : /END/;
};

type Y = unit {
    %synchronize-after = /END/;
    a: bytes &size=10;
};

type Foo = unit {
    x: X;
    y: Y &synchronize;
};
```

- A field that's located inside the input stream at a fixed offset relative to the field triggering the error. The parser will then be able to skip ahead to that offset. Example:

```
type X = unit { ... }
type Y = unit { ... }

type Foo = unit {}
    x: X &size=512;
    y: Y &synchronize;
};
```

  Here, when parsing `Foo:x` triggers an error, Spicy will know that it can continue with `Foo::y` at offset `<beginning of Foox:x> + 512`.

Todo: This synchronization strategy is not yet implemented.

- When *parsing a vector*, the inner elements may provide synchronization points as well. Example:

```
type X = unit {
    a: b"begin-of-X";
    b: bytes &size=10;
};

type Foo = unit {}
    xs: (X &synchronize)[];
};
```

If one element of the vector `Foo::xs` fails to parse, Spicy will attempt to find the beginning of the next `X` in the input stream and continue there. For this to work, the vector's elements must itself represent valid synchronization point (e.g., start with a literal). If the list is of fixed size, after successful synchronization, it will contain the expected number of entries, but some of them may remain (fully or partially) uninitialized if they encountered errors.

Phase 2: Trial parsing

Once input has been realigned with a synchronization point, parsing switches from synchronization mode into trial mode, in which the parser will attempt to confirm that it has indeed found a viable place to continue. It does so by proceeding to parse subsequent input from the synchronization point onwards, until one of the following occurs:

- A unit hook explicitly acknowledges that synchronization has been successful by executing Spicy's *confirm* statement. Typically, a grammar will do so once it has been able to correctly parse a few fields following the synchronization point–whatever it needs to sufficiently certain that it's indeed seeing the expected structure.
- A unit hook explicitly declines the synchronization by executing Spicy's *reject* statement. This will abandon the current synchronization attempt, and switch back into the original synchronization mode again to find another location to try.
- Parsing reaches the end of the grammar without either `confirm` or `reject` already called. In this case, the parser will abort with a fatal parse error.

Note that during trial mode, any fields between the synchronization point and the eventual `confirm`/`reject` location will already be processed as usual, including any hooks executing. This may leave the unit's state in a partially initialized state if trial parsing eventually fails. Trial mode will also consume any input along the way, with any further synchronization attempts proceeding only on subsequent, not yet seen, data.

Synchronisation Hooks

For customization, Spicy provides a set of hooks executing at different points during the synchronization process:

`on %synced { ...}` Executes when a synchronization point has been found and parsing resumes there, just before the parser begins processing the corresponding field in trial mode.

`on %confirmed { ...}` Executes when trial mode ends successfully with *confirm*.

`on %rejected { ...}` Executes when trial mode fails with *reject*.

Example Synchronization Process

As an example, let's consider a grammar consisting of two sections where each section is started with a section header literal (`SEC_A` and `SEC_B` here).

We want to allow for inputs which miss parts or all of the first section. For such inputs, we can still synchronize the input stream by looking for the start of the second section. (For simplicity, we just use a single unit, even though typically one would probably have separate units for the two sections.)

```
module Test;

public type Example = unit {
    start_a: /SEC_A/;
    a: uint8;

    # If we fail to find e.g., 'SEC_A' in the input, try to synchronize on this␣
↪literal.
```

(continues on next page)

(continued from previous page)

```
    start_b: /SEC_B/ &synchronize;
    b: bytes &eod;

    # In this example confirm unconditionally.
    on %synced {
        print "Synced: %s" % self;
        confirm;
    }

    # Perform logging for these %confirmed and %rejected.
    on %confirmed { print "Confirmed: %s" % self; }
    on %rejected { print "Rejected: %s" % self; }

    on %done { print "Done %s" % self; }
};
```

Let us consider that this parsers encounters the input `\xFFSEC_Babc` that missed the `SEC_A` section marker:

- `start_a` missing,
- `a=255`
- `start_b=SEC_B` as expected, and
- `b=abc`.

For such an input parsing will encounter an initial error when it sees `\xFF` where `SEC_A` would have been expected.

1. Since `start_b` is marked as a synchronization point, the parser enters synchronisation mode, and jumps over the field `a` to `start_b`, to now search for `SEC_B`.
2. At this point the input still contains the unexpected `\xFF` and remains `\xFFSEC_Babc` . While searching for `SEC_B` `\xFF` is skipped over, and then the expected token is found. The input is now `SEC_Babc`.
3. The parser has successfully synchronized and enters trial mode. All `%synced` hooks are invoked.
4. The unit's `%synced` hook executes `confirm` and the parser leaves trial mode. All `%confirmed` hooks are invoked.
5. Regular parsing continues at `start_b`. The input was `SEC_Babc` so `start_b` is set to `SEC_B` and `b` to `abc`.

Since parsing for `start_a` was unsuccessful and `a` was jumped over, their fields remain unset.

```
# printf '\xFFSEC_Babc' | spicy-driver foo.spicy
Synced: [$start_a=(not set), $a=(not set), $start_b=(not set), $b=(not set)]
Confirmed: [$start_a=(not set), $a=(not set), $start_b=(not set), $b=(not set)]
Done [$start_a=(not set), $a=(not set), $start_b=b"SEC_B", $b=b"abc"]
```

2.5.2 Language

Spicy provides a domain-specific language that consists of two main types of constructs: parsing elements that capture the layout of an input format; along with more standard constructs of typical imperative scripting languages, such as modules, types, declarations, expressions, etc.. While the *previous section* focuses on the former, we summarize the more "traditional" parts of Spicy's language in the following.

We assume some familiarity with other scripting languages. Generally, where not otherwise stated, think of Spicy as a "C-style scripting language" in terms of syntax & semantics, with corresponding rules for, e.g., block structure (`{`

`... }`), operator precedence, identifier naming, etc.. If you are familiar with Zeek's scripting language in particular, you should be able to get up to speed with Spicy pretty quickly.

Identifiers

Spicy distinguishes between different kinds of identifiers:

Declarations Identifiers used in declarations of variables, types, functions, etc., must start with a letter or underscore, and otherwise contain only alphanumerical characters and underscores. They must not start with two underscores, and cannot match any of *Spicy's built-in keywords*.

Attributes Identifiers augmenting other language elements with additional *attributes* always begin with &. They otherwise follow the same rules as identifiers for declarations, except that they also permit dashes. Note that you cannot define your own attributes; usage is limited to a set of predefined options.

Properties Identifiers defining *properties* of modules and types (as in, e.g., *Meta data*) always begin with %. They otherwise follow the same rules as identifiers for declarations, except that they also permit dashes. Note that you cannot define your own properties; usage is limited to a set of predefined options.

Identifiers are always case-sensitive in Spicy.

Modules

Spicy source code is structured around modules, which introduce namespaces around other elements defined inside (e.g., types, functions). Accordingly, all Spicy input files must start with `module NAME;`, where `NAME` is scope that's being created.

After that initial `module` statement, modules may contain arbitrary list of declarations (types, globals, functions), as well as code statements to execute. Any code defined at the global level will run once at the module's initialization time. That's what enables Spicy's minimal `hello-world` module to look like the following:

```
module Test;

print "Hello, world!";
```

```
# spicyc -j hello-world.spicy
Hello, world!
```

Importing

To make the contents of another module accessible, Spicy provides an `import NAME;` statement that pulls in all public identifiers of the specified external module. Spicy then searches for `name.spicy` (i.e., the lower-case version of the imported module `NAME` plus a `.spicy` extension) along it's module search path. By default, that's the current directory plus the location where Spicy's pre-built library modules are installed.

`spicy-config --libdirs` shows the default search path. The Spicy tools `spicy` && `spicy-driver` provide `--library-path` options to add further custom directories. They also allow to fully replace the built-in default search with a custom value by setting the environment variable `SPICY_PATH`.

There's a second version of the import statement that allows to import from relative locations inside the search path: `import NAME from X.Y.Z;` searches the module `NAME` (i.e., `NAME.spicy`) inside a sub-directory `X/Y/Z` along the search path.

Once Spicy code has imported a module, it can access identifiers by prefixing them with the module's namespace:

```
import MyModule;

print MyModule::my_global_variable;
```

Generally, only identifiers declared as `public` become accessible across module boundaries. The one exception are types, which are implicitly public.

Note: Spicy makes types implicitly public so that external *unit hooks* always have access to them. We may consider a more fine-grained model here in the future.

Spicy comes with a set of *library modules* that you may import in your code to gain access to their functionality.

Global Properties

A module may define the following global properties:

`%byte-order = ORDER;` Defaults the byte order for any parsing inside this module to <*expr*>, where `ORDER` must be of type is type *spicy::ByteOrder*.

`%spicy-version = "VERSION";` Specifies that the module requires a given minimum version of Spicy, where `VERSION` must be a string of the form `X.Y` or `X.Y.Z`.

`%skip = REGEXP;` Specifies a pattern which should be skipped when encountered in the input stream in between parsing of unit fields (including before/after the first/last field).

`%skip-pre = REGEXP;` Specifies a pattern which should be skipped when encountered in the input stream before parsing of a unit begins.

`%skip-post = REGEXP;` Specifies a pattern which should be skipped when encountered in the input stream after parsing of a unit has finished.

Functions

Spicy's language allows to define custom functions just like most other languages. The generic syntax for defining a function with is `N` parameters is:

```
[public] function NAME(NAME_1: TYPE_1, ..., NAME_N: TYPE_N) [: RETURN_TYPE ] {
    ... BODY ...
}
```

A `public` function will be *accessible from other modules* . If the return type is skipped, it's implicitly taken as `void`, i.e., the function will not return anything. If a function has return type other than void, all paths through the body must end in a *return* returning a corresponding value.

A parameter specification can be postfixed with a default value: `NAME: TYPE = DEFAULT`. Callers may then omit that parameter.

By default, by parameters are passed by constant reference and hence remain read-only inside the function's body. To make a parameter modifiable, with any changes becoming visible to the caller, a parameter can be prefixed with `inout`:

```
module Test;

global s = "1";

```

(continues on next page)

(continued from previous page)

```
function foo(inout x: string) {
    x = "2";
}

print s;
foo(s);
print s;
```

```
1
2
```

Spicy has couple more function-like constructs (*Unit Hooks* and *Unit Parameters*) that use the same conventions for parameter passing.

Variables and Constants

At the global module level, we declare variables with the `global` keyword:

```
[public] global NAME: TYPE [= DEFAULT];
```

This defines a global variable called `NAME` with type `TYPE`. If the variable is declared with `public` visibility other modules can reference it. If a default is given, Spicy initializes the global accordingly before any code executes. Otherwise, the global receives a type-specific default, typically the type's notion of a null value. As a result, globals are always initialized to a well-defined value.

As a shortcut, you can skip `:  TYPE` if the global comes with a default. Spicy then just applies the expression's type to the global.

We define global constants in a similar way, just replacing `global` with `const`:

```
const x: uint32 = 42;
const foo = "Foo";
```

Inside a function, local variables use the same syntax once more, just prefixed with `local` this time:

```
function f() {
    local x: bytes;
    local y = "Y";

}
```

Usual scoping rules apply to locals. Just like globals, locals are always initialized to a well-defined value: either their default if given, or the type's null value.

Types

Address

The address type stores both IPv4 and IPv6 addresses.

Type

- `addr`

Constants

- IPv4: `1.2.3.4`
- IPv6: `[2001:db8:85a3:8d3:1319:8a2e:370:7348]`, `[::1.2.3.4]`

This type supports the *pack/unpack operators*.

Methods

`family()` → `hilti::AddressFamily`
Returns the protocol family of the address, which can be IPv4 or IPv6.

Operators

`addr == addr` → `bool`
Compares two address values.

`addr != addr` → `bool`
Compares two address values.

Bitfield

Bitfields provide access to individual bitranges inside an unsigned integer. That can't be instantiated directly, but must be defined and parsed inside a unit.

Type

- `bitfield(N) { RANGE_1; ...; RANGE_N }`
- Each `RANGE` has one of the forms `LABEL: A` or `LABEL: A..B` where `A` and `B` are bit numbers.

Operators

`bitfield . <attribute>` → `<field type>`
Retrieves the value of a bitfield's attribute. This is the value of the corresponding bits inside the underlying integer value, shifted to the very right.

Bool

Boolean values can be `True` or `False`.

Type

- `bool`

Constants

- `True`, `False`

Operators

`bool == bool` → `bool`
: Compares two boolean values.

`bool != bool` → `bool`
: Compares two boolean values.

Bytes

Bytes instances store raw, opaque data. They provide iterators to traverse their content.

Types

- `bytes`
- `iterator<bytes>`

Constants

- `b"Spicy"`, `b""`

Methods

`at(i: uint<64>)` → `iterator<bytes>`
: Returns an iterator representing the offset *i* inside the bytes value.

`decode(charset: enum = hilti::Charset::UTF8, errors: enum = hilti::DecodeErrorStrategy::`
: Interprets the `bytes` as representing an binary string encoded with the given character set, and converts it into a UTF8 string. If data is encountered that *charset* or UTF* cannot represent, it's handled according to the *errors* strategy.

`find(needle: bytes)` → `tuple<bool, iterator<bytes>>`
: Searches *needle* in the value's content. Returns a tuple of a boolean and an iterator. If *needle* was found, the boolean will be true and the iterator will point to its first occurrence. If *needle* was not found, the boolean will be false and the iterator will point to the last position so that everything before it is guaranteed to not contain even a partial match of *needle*. Note that for a simple yes/no result, you should use the `in` operator instead of this method, as it's more efficient.

`join(inout parts: vector)` → `bytes`

Returns the concatenation of all elements in the *parts* list rendered as printable strings. The portions will be separated by the bytes value to which this method is invoked as a member.

`lower(charset: enum = hilti::Charset::UTF8, errors: enum = hilti::DecodeErrorStrategy::RE`

Returns a lower-case version of the bytes value, assuming it is encoded in character set *charset*. If data is encountered that *charset* cannot represent, it's handled according to the *errors* strategy.

`match(regex: regexp, [ group: uint<64> ])` → `result<bytes>`

Matches the `bytes` object against the regular expression *regex*. Returns the matching part or, if *group* is given, then the corresponding subgroup. The expression is considered anchored to the beginning of the data.

`split([ sep: bytes ])` → `vector<bytes>`

Splits the bytes value at each occurrence of *sep* and returns a vector containing the individual pieces, with all separators removed. If the separator is not found, the returned vector will have the whole bytes value as its single element. If the separator is not given, or empty, the split will take place at sequences of white spaces.

`split1([ sep: bytes ])` → `tuple<bytes, bytes>`

Splits the bytes value at the first occurrence of *sep* and returns the two parts as a 2-tuple, with the separator removed. If the separator is not found, the returned tuple will have the whole bytes value as its first element and an empty value as its second element. If the separator is not given, or empty, the split will take place at the first sequence of white spaces.

`starts_with(b: bytes)` → `bool`

Returns true if the bytes value starts with *b*.

`strip([ side: spicy::Side ], [ set: bytes ])` → `bytes`

Removes leading and/or trailing sequences of all characters in *set* from the bytes value. If *set* is not given, removes all white spaces. If *side* is given, it indicates which side of the value should be stripped; `Side::Both` is the default if not given.

`sub(begin: uint<64>, end: uint<64>)` → `bytes`

Returns the subsequence from offset *begin* to (but not including) offset *end*.

`sub(inout begin: iterator<bytes>, inout end: iterator<bytes>)` → `bytes`

Returns the subsequence from *begin* to (but not including) *end*.

`sub(inout end: iterator<bytes>)` → `bytes`

Returns the subsequence from the value's beginning to (but not including) *end*.

`to_int([ base: uint<64> ])` → `int<64>`

Interprets the data as representing an ASCII-encoded number and converts that into a signed integer, using a base of *base*. *base* must be between 2 and 36. If *base* is not given, the default is 10.

`to_int(byte_order: enum)` → `int<64>`

Interprets the `bytes` as representing an binary number encoded with the given byte order, and converts it into signed integer.

`to_time([ base: uint<64> ])` → `time`

Interprets the `bytes` as representing a number of seconds since the epoch in the form of an ASCII-encoded number, and converts it into a time value using a base of *base*. If *base* is not given, the default is 10.

`to_time(byte_order: enum)` → `time`

Interprets the `bytes` as representing as number of seconds since the epoch in the form of an binary number encoded with the given byte order, and converts it into a time value.

`to_uint([ base: uint<64> ])` → `uint<64>`

Interprets the data as representing an ASCII-encoded number and converts that into an unsigned integer, using a base of *base*. *base* must be between 2 and 36. If *base* is not given, the default is 10.

`to_uint(byte_order: enum)` → `uint<64>`

Interprets the `bytes` as representing an binary number encoded with the given byte order, and converts it into an unsigned integer.

`upper(charset: enum = hilti::Charset::UTF8, errors: enum = hilti::DecodeErrorStrategy::R`

Returns an upper-case version of the bytes value, assuming it is encoded in character set *charset*. If data is encountered that *charset* cannot represent, it's handled according to the *errors* strategy.

Operators

`begin(<container>)` → `<iterator>`

Returns an iterator to the beginning of the container's content.

`end(<container>)` → `<iterator>`

Returns an iterator to the end of the container's content.

`bytes == bytes` → `bool`

Compares two bytes values lexicographically.

`bytes > bytes` → `bool`

Compares two bytes values lexicographically.

`bytes >= bytes` → `bool`

Compares two bytes values lexicographically.

`bytes in bytes` → `bool`

Returns true if the right-hand-side value contains the left-hand-side value as a subsequence.

`bytes !in bytes` → `bool`

Performs the inverse of the corresponding `in` operation.

`bytes < bytes` → `bool`

Compares two bytes values lexicographically.

`bytes <= bytes` → `bool`

Compares two bytes values lexicographically.

`|bytes|` → `uint<64>`

Returns the number of bytes the value contains.

`bytes + bytes` → `bytes`

Returns the concatenation of two bytes values.

`bytes += bytes` → `bytes`

Appends one bytes value to another.

`bytes += uint<8>` → `bytes`

Appends a single byte to the data.

`bytes += view<stream>` → `bytes`

Appends a view of stream data to a bytes instance.

`bytes != bytes` → `bool`

Compares two bytes values lexicographically.

Iterator Operators

`*iterator<bytes>` → `uint<8>`
  Returns the character the iterator is pointing to.

`iterator<bytes> - iterator<bytes>` → `int<64>`
  Returns the number of bytes between the two iterators. The result will be negative if the second iterator points to a location before the first. The result is undefined if the iterators do not refer to the same bytes instance.

`iterator<bytes> == iterator<bytes>` → `bool`
  Compares the two positions. The result is undefined if they are not referring to the same bytes value.

`iterator<bytes> > iterator<bytes>` → `bool`
  Compares the two positions. The result is undefined if they are not referring to the same bytes value.

`iterator<bytes> >= iterator<bytes>` → `bool`
  Compares the two positions. The result is undefined if they are not referring to the same bytes value.

`iterator<bytes>++` → `iterator<bytes>`
  Advances the iterator by one byte, returning the previous position.

`++iterator<bytes>` → `iterator<bytes>`
  Advances the iterator by one byte, returning the new position.

`iterator<bytes> < iterator<bytes>` → `bool`
  Compares the two positions. The result is undefined if they are not referring to the same bytes value.

`iterator<bytes> <= iterator<bytes>` → `bool`
  Compares the two positions. The result is undefined if they are not referring to the same bytes value.

`iterator<bytes> + uint<64>` → `iterator<bytes>` (commutative)
  Returns an iterator which is pointing the given number of bytes beyond the one passed in.

`iterator<bytes> += uint<64>` → `iterator<bytes>`
  Advances the iterator by the given number of bytes.

`iterator<bytes> != iterator<bytes>` → `bool`
  Compares the two positions. The result is undefined if they are not referring to the same bytes value.

Enum

Enum types associate labels with numerical values.

Type

- `enum { LABEL_1, ..., LABEL_N }`
- Each label has the form `ID [= VALUE]`. If `VALUE` is skipped, one will be assigned automatically.
- Each enum type comes with an implicitly defined `Undef` label with a value distinct from all other ones. When coerced into a boolean, an enum will be true iff it's not `Undef`.

Note: An instance of an enum can assume a numerical value that does not map to any of its defined labels. If printed, it will then render into `<unknown-N>` in that case, with `N` being the decimal expression of its numeric value.

Constants

- The individual labels represent constants of the corresponding type (e.g., `MyEnum::MyFirstLabel` is a constant of type `MyEnum`).

Methods

`has_label()` → `bool`
Returns *true* if the value of *op1* corresponds to a known enum label (other than `Undef`), as defined by it's type.

Operators

`enum-type(int)` → `enum`
Instantiates an enum instance initialized from a signed integer value. The value does *not* need to correspond to any of the type's enumerator labels.

`enum-type(uint)` → `enum`
Instantiates an enum instance initialized from an unsigned integer value. The value does *not* need to correspond to any of the type's enumerator labels. It must not be larger than the maximum that a *signed* 64-bit integer value can represent.

`cast<int-type>(enum)` → `int`
Casts an enum value into a signed integer. If the enum value is `Undef`, this will return `-1`.

`cast<uint-type>(enum)` → `uint`
Casts an enum value into a unsigned integer. This will throw an exception if the enum value is `Undef`.

`enum == enum` → `bool`
Compares two enum values.

`enum != enum` → `bool`
Compares two enum values.

Exception

Todo: This isn't available in Spicy yet (#%s89).

Integer

Spicy distinguishes between signed and unsigned integers, and always requires specifying the bitwidth of a type.

Type

- `intN` for signed integers, where `N` can be one of 8, 16, 32, 64.
- `uintN` for signed integers, where `N` can be one of 8, 16, 32, 64.

Constants

- Unsigned integer: `1234`, `+1234`, `uint8(42)`, `uint16(42)`, `uint32(42)`, `uint64(42)`
- Signed integer: `-1234`, `int8(42)`, `int8(-42)`, `int16(42)`, `int32(42)`, `int64(42)`

This type supports the *pack/unpack operators*.

Operators

`uint & uint` → `uint`
    Computes the bit-wise 'and' of the two integers.

`uint | uint` → `uint`
    Computes the bit-wise 'or' of the two integers.

`uint ^ uint` → `uint`
    Computes the bit-wise 'xor' of the two integers.

`int16(int)` → `int<16>`
    Creates a 16-bit signed integer value.

`int16(uint)` → `int<16>`
    Creates a 16-bit signed integer value.

`int32(int)` → `int<32>`
    Creates a 32-bit signed integer value.

`int32(uint)` → `int<32>`
    Creates a 32-bit signed integer value.

`int64(int)` → `int<64>`
    Creates a 64-bit signed integer value.

`int64(uint)` → `int<64>`
    Creates a 64-bit signed integer value.

`int8(int)` → `int<8>`
    Creates a 8-bit signed integer value.

`int8(uint)` → `int<8>`
    Creates a 8-bit signed integer value.

`uint16(int)` → `uint<16>`
    Creates a 16-bit unsigned integer value.

`uint16(uint)` → `uint<16>`
    Creates a 16-bit unsigned integer value.

`uint32(int)` → `uint<32>`
    Creates a 32-bit unsigned integer value.

`uint32(uint)` → `uint<32>`
    Creates a 32-bit unsigned integer value.

`uint64(int)` → `uint<64>`
Creates a 64-bit unsigned integer value.

`uint64(uint)` → `uint<64>`
Creates a 64-bit unsigned integer value.

`uint8(int)` → `uint<8>`
Creates a 8-bit unsigned integer value.

`uint8(uint)` → `uint<8>`
Creates a 8-bit unsigned integer value.

`cast<bool-type>(int)` → `bool`
Converts the value to a boolean by comparing against zero

`cast<enum-type>(int)` → `enum`
Converts the value into an enum instance. The value does *not* need to correspond to any of the target type's enumerator labels.

`cast<enum-type>(uint)` → `enum`
Converts the value into an enum instance. The value does *not* need to correspond to any of the target type's enumerator labels. It must not be larger than the maximum that a *signed* 64-bit integer value can represent.

`cast<int-type>(int)` → `int`
Converts the value into another signed integer type, accepting any loss of information.

`cast<int-type>(uint)` → `int`
Converts the value into a signed integer type, accepting any loss of information.

`cast<interval-type>(int)` → `interval`
Interprets the value as number of seconds.

`cast<interval-type>(uint)` → `interval`
Interprets the value as number of seconds.

`cast<real-type>(int)` → `real`
Converts the value into a real, accepting any loss of information.

`cast<real-type>(uint)` → `real`
Converts the value into a real, accepting any loss of information.

`cast<time-type>(uint)` → `time`
Interprets the value as number of seconds since the UNIX epoch.

`cast<uint-type>(int)` → `uint`
Converts the value into an unsigned integer type, accepting any loss of information.

`cast<uint-type>(uint)` → `uint`
Converts the value into another unsigned integer type, accepting any loss of information.

`int--` → `int`
Decrements the value, returning the old value.

`uint--` → `uint`
Decrements the value, returning the old value.

`++int` → `int`
Increments the value, returning the new value.

`++uint` → `uint`
Increments the value, returning the new value.

`int - int` → `int`
Computes the difference between the two integers.

`uint - uint` → `uint`
Computes the difference between the two integers.

`int -= int` → `int`
Decrements the first value by the second, assigning the new value.

`uint -= uint` → `uint`
Decrements the first value by the second.

`int / int` → `int`
Divides the first integer by the second.

`uint / uint` → `uint`
Divides the first integer by the second.

`int /= int` → `int`
Divides the first value by the second, assigning the new value.

`uint /= uint` → `uint`
Divides the first value by the second, assigning the new value.

`int == int` → `bool`
Compares the two integers.

`uint == uint` → `bool`
Compares the two integers.

`int > int` → `bool`
Compares the two integers.

`uint > uint` → `bool`
Compares the two integers.

`int >= int` → `bool`
Compares the two integers.

`uint >= uint` → `bool`
Compares the two integers.

`int++` → `int`
Increments the value, returning the old value.

`uint++` → `uint`
Increments the value, returning the old value.

`++int` → `int`
Increments the value, returning the new value.

`++uint` → `uint`
Increments the value, returning the new value.

`int < int` → `bool`
Compares the two integers.

`uint < uint` → `bool`
Compares the two integers.

`int <= int` → `bool`
Compares the two integers.

`uint <= uint` → `bool`
Compares the two integers.

`int % int` → `int`
    Computes the modulus of the first integer divided by the second.

`uint % uint` → `uint`
    Computes the modulus of the first integer divided by the second.

`int * int` → `int`
    Multiplies the first integer by the second.

`uint * uint` → `uint`
    Multiplies the first integer by the second.

`int *= int` → `int`
    Multiplies the first value by the second, assigning the new value.

`uint *= uint` → `uint`
    Multiplies the first value by the second, assigning the new value.

`~uint` → `uint`
    Computes the bit-wise negation of the integer.

`int ** int` → `int`
    Computes the first integer raised to the power of the second.

`uint ** uint` → `uint`
    Computes the first integer raised to the power of the second.

`uint << uint` → `uint`
    Shifts the integer to the left by the given number of bits.

`uint >> uint` → `uint`
    Shifts the integer to the right by the given number of bits.

`-int` → `int`
    Inverts the sign of the integer.

`-uint` → `int`
    Inverts the sign of the integer.

`int + int` → `int`
    Computes the sum of the integers.

`uint + uint` → `uint`
    Computes the sum of the integers.

`int += int` → `int`
    Increments the first integer by the second.

`uint += uint` → `uint`
    Increments the first integer by the second.

`int != int` → `bool`
    Compares the two integers.

`uint != uint` → `bool`
    Compares the two integers.

Interval

Am interval value represents a period of time. Intervals are stored with nanosecond resolution, which is retained across all calculations.

Type

- `interval`

Constants

- `interval(SECS)` creates an interval from a signed integer or real value `SECS` specifying the period in seconds.
- `interval_ns(NSECS)` creates an interval from a signed integer value `NSECS` specifying the period in nanoseconds.

Methods

`nanoseconds()` → `uint<64>`
: Returns the time as an integer value representing nanoseconds since the UNIX epoch.

`seconds()` → `real`
: Returns the time as a real value representing seconds since the UNIX epoch.

Operators

`time(int)` → `time`
: Creates an time interpreting the argument as number of seconds.

`time(real)` → `time`
: Creates an time interpreting the argument as number of seconds.

`time(uint)` → `time`
: Creates an time interpreting the argument as number of seconds.

`time_ns(int)` → `time`
: Creates an time interpreting the argument as number of nanoseconds.

`time_ns(uint)` → `time`
: Creates an time interpreting the argument as number of nanoseconds.

`time - time` → `interval`
: Returns the difference of the times.

`time - interval` → `time`
: Subtracts the interval from the time.

`time == time` → `bool`
: Compares two time values.

`time > time` → `bool`
: Compares the times.

`time >= time` → `bool`
: Compares the times.

`time < time` → `bool`
Compares the times.

`time <= time` → `bool`
Compares the times.

`time + interval` → `time` (commutative)
Adds the interval to the time.

`time != time` → `bool`
Compares two time values.

List

Spicy uses lists only in a limited form as temporary values, usually for initializing other containers. That means you can only create list constants, but you cannot declare variables or unit fields to have a `list` type (use *vector* instead).

Constants

- `[E_1, E_2, ..., E_N]` creates a list of `N` elements. The values `E_I` must all have the same type. `[]` creates an empty list of unknown element type.
- `[EXPR for ID in ITERABLE]` creates a list by evaluating `EXPR` for all elements in `ITERABLE`, assembling the individual results into the final list value. The extended form `[EXPR for ID in SEQUENCE if COND]` includes only elements into the result for which `COND` evaluates to `True`. Both `EXPR` and `COND` can use `ID` to refer to the current element.
- `list(E_1, E_2, ..., E_N)` is the same as `[E_1, E_2, ..., E_N]`, and `list()` is the same as `[]`.
- `list<T>(E_1, E_2, ..., E_N)` creates a list of type `T`, initializing it with the `N` elements `E_I`. `list<T>()` creates an empty list.

Operators

`begin(<container>)` → `<iterator>`
Returns an iterator to the beginning of the container's content.

`end(<container>)` → `<iterator>`
Returns an iterator to the end of the container's content.

`list == list` → `bool`
Compares two lists element-wise.

`|list|` → `uint<64>`
Returns the number of elements a list contains.

`list != list` → `bool`
Compares two lists element-wise.

Map

Maps are containers holding key/value pairs of elements, with fast lookup for keys to retrieve the corresponding value. They provide iterators to traverse their content, with no particular ordering.

Types

- `map<K, V>` specifies a map with key type `K` and value type `V`.
- `iterator<map<K, V>>`

Constants

- `map(K_1: V_1, K_2: V_2, ..., K_N: V_N)` creates a map of `N` elements, initializing it with the given key/value pairs. The keys `K_I` must all have the same type, and the values `V_I` must likewise all have the same type. `map()` creates an empty map of unknown key/value types; this cannot be used directly but must be coerced into a fully-defined map type first.
- `map<K, V>(K_1: V_1, K_2: V_2, ..., K_N: V_N)` creates a map of type `map<K, V>`, initializing it with the given key/value pairs. `map<K, V>()` creates an empty map.

Methods

`clear()` → `void`
: Removes all elements from the map.

`get(key: <any>, [ default: <any> ])` → `<type of element>`
: Returns the map's element for the given key. If the key does not exist, returns the default value if provided; otherwise throws a runtime error.

Operators

`begin(<container>)` → `<iterator>`
: Returns an iterator to the beginning of the container's content.

`delete map[key]` → `void`
: Removes an element from the map.

`end(<container>)` → `<iterator>`
: Returns an iterator to the end of the container's content.

`map == map` → `bool`
: Compares two maps element-wise.

`<any> in map` → `bool`
: Returns true if an element is part of the map.

`<any> !in map` → `bool`
: Performs the inverse of the corresponding `in` operation.

`map[key]` → `<type of element>`
: Returns the map's element for the given key. The key must exist, otherwise the operation will throw a runtime error.

`map[key]=<any>` → `void`
: Updates the map value for a given key. If the key does not exist a new element is inserted.

`|map|` → `uint<64>`
Returns the number of elements a map contains.

`map != map` → `bool`
Compares two maps element-wise.

Iterator Operators

`*iterator<map>` → `<dereferenced type>`
Returns the map element that the iterator refers to.

`iterator<map> == iterator<map>` → `bool`
Returns true if two map iterators refer to the same location.

`iterator<map>++` → `iterator<map>`
Advances the iterator by one map element, returning the previous position.

`++iterator<map>` → `iterator<map>`
Advances the iterator by one map element, returning the new position.

`iterator<map> != iterator<map>` → `bool`
Returns true if two map iterators refer to different locations.

Optional

An optional value may hold a value of another type, or can alternatively remain unset. A common use case for optional is the return value of a function that may fail.

- `optional<TYPE>`

Constants

- `optional(EXPR)` creates an `optional<T>`, where `T` is the type of the expression `EXPR` and initializes it with the value of `EXPR`.

More commonly, however, optional values are initialized through assignment:

- Assigning an instance of `TYPE` to an `optional<TYPE>` sets it to the instance's value.
- Assigning `Null` to an `optional<TYPE>` unsets it.

Operators

`*optional` → `<dereferenced type>`
Returns the element stored, or throws an exception if none.

Port

Ports represent the combination of a numerical port number and an associated transport-layer protocol.

Type

- `port`

Constants

- `443/tcp`, `53/udp`
- `port(PORT, PROTOCOL)` creates a `port` where `PORT` is a port number and `PROTOCOL` a *spicy::Protocol*.

Methods

`protocol()` → `hilti::Protocol`
  Returns the protocol the port is using (such as UDP or TCP).

Operators

`port(uint<16>,enum)` → `port`
  Creates a port instance.

`port == port` → `bool`
  Compares two port values.

`port != port` → `bool`
  Compares two port values.

Real

“Real” values store floating points with double precision.

Type

- `real`

Constants

- `3.14`, `10e9`, `0x1.921fb78121fb8p+1`

This type supports the *pack/unpack operators*.

Operators

`cast<int-type>(real)` → `int`
: Converts the value to a signed integer type, accepting any loss of information.

`cast<interval-type>(real)` → `interval`
: Interprets the value as number of seconds.

`cast<time-type>(real)` → `time`
: Interprets the value as number of seconds since the UNIX epoch.

`cast<uint-type>(real)` → `uint`
: Converts the value to an unsigned integer type, accepting any loss of information.

`real - real` → `real`
: Returns the difference between the two values.

`real -= real` → `real`
: Subtracts the second value from the first, assigning the new value.

`real / real` → `real`
: Divides the first value by the second.

`real /= real` → `real`
: Divides the first value by the second, assigning the new value.

`real == real` → `bool`
: Compares the two reals.

`real > real` → `bool`
: Compares the two reals.

`real >= real` → `bool`
: Compares the two reals.

`real < real` → `bool`
: Compares the two reals.

`real <= real` → `bool`
: Compares the two reals.

`real % real` → `real`
: Computes the modulus of the first real divided by the second.

`real * real` → `real`
: Multiplies the first real by the second.

`real *= real` → `real`
: Multiplies the first value by the second, assigning the new value.

`real ** real` → `real`
: Computes the first real raised to the power of the second.

`-real` → `real`
: Inverts the sign of the real.

`real + real` → `real`
: Returns the sum of the reals.

`real += real` → `real`
: Adds the first real to the second, assigning the new value.

`real != real` → `bool`
: Compares the two reals.

Regular Expression

Spicy provides POSIX-style regular expressions.

Type

- `regexp`

Constants

- `/Foo*bar?/`, `/X(..)(..)(..)Y/`

Regular expressions use the extended POSIX syntax, with a few smaller differences and extensions:

- Supported character classes are: `[:lower:]`, `[:upper:]`, `[:digit:]`, `[:blank:]`.
- `\b` asserts a word-boundary, `\B` matches asserts no word boundary.
- `\xXX` matches a byte with the binary hex value `XX` (e.g., `\xff` matches a byte of decimal value 255).
- `{#<number>}` associates a numerical ID with a regular expression (useful for set matching).

Regular expression constants support two optional attributes:

`&anchor` Implicitly anchor the expression, meaning it must match at the beginning of the data.

`&nosub` Compile without support for capturing subexpressions, which makes matching more efficient.

Methods

`find(data: bytes) → tuple<int<32>, bytes>`

Searches the regular expression in *data* and returns the matching part. Different from `match`, this does not anchor the expression to the beginning of the data: it will find matches at arbitrary starting positions. Returns a 2-tuple with (1) an integer match indicator with the same semantics as that returned by `find`; and (2) if a match has been found, the data that matches the regular expression. (Note: Currently this function has a runtime that's quadratic in the size of *data*; consider using *match* if performance is an issue.)

`match(data: bytes) → int<32>`

Matches the regular expression against *data*. If it matches, returns an integer that's greater than zero. If multiple patterns have been compiled for parallel matching, that integer will be the ID of the matching pattern. Returns -1 if the regular expression does not match the data, but could still yield a match if more data were added. Returns 0 if the regular expression is not found and adding more data wouldn't change anything. The expression is considered anchored, as though it starts with an implicit ^ regexp operator, to the beginning of the data.

`match_groups(data: bytes) → vector<bytes>`

Matches the regular expression against *data*. If it matches, returns a vector with one entry for each capture group defined by the regular expression; starting at index 1. Each of these entries is a view locating the matching bytes. In addition, index 0 always contains the data that matches the full regular expression. Returns an empty vector if the expression is not found. The expression is considered anchored, as though it starts with an implicit ^ regexp operator, to the beginning of the data. This method is not compatible with pattern sets and will throw a runtime exception if used with a regular expression compiled from a set.

`token_matcher() → hilti::MatchState`

Initializes state for matching regular expression incrementally against chunks of future input. The expression is considered anchored, as though it starts with an implicit ^ regexp operator, to the beginning of the data.

Set

Sets are containers for unique elements with fast lookup. They provide iterators to traverse their content, with no particular ordering.

Types

- `set<T>` specifies a set with unique elements of type `T`.
- `iterator<set<T>>`

Constants

- `set(E_1, E_2, ..., E_N)` creates a set of `N` elements. The values `E_I` must all have the same type. `set()` creates an empty set of unknown element type; this cannot be used directly but must be coerced into a fully-defined set type first.
- `set<T>(E_1, E_2, ..., E_N)` creates a set of type `T`, initializing it with the elements `E_I`. `set<T>()` creates an empty set.

Methods

`clear()` → `void`
Removes all elements from the set.

Operators

`add set[element]` → `void`
Adds an element to the set.

`begin(<container>)` → `<iterator>`
Returns an iterator to the beginning of the container's content.

`delete set[element]` → `void`
Removes an element from the set.

`end(<container>)` → `<iterator>`
Returns an iterator to the end of the container's content.

`set == set` → `bool`
Compares two sets element-wise.

`<any> in set` → `bool`
Returns true if an element is part of the set.

`<any> !in set` → `bool`
Performs the inverse of the corresponding `in` operation.

`|set|` → `uint<64>`
Returns the number of elements a set contains.

`set != set` → `bool`
Compares two sets element-wise.

Iterator Operators

`*iterator<set>` → `<dereferenced type>`
Returns the set element that the iterator refers to.

`iterator<set> == iterator<set>` → `bool`
Returns true if two sets iterators refer to the same location.

`iterator<set>++` → `iterator<set>`
Advances the iterator by one set element, returning the previous position.

`++iterator<set>` → `iterator<set>`
Advances the iterator by one set element, returning the new position.

`iterator<set> != iterator<set>` → `bool`
Returns true if two sets iterators refer to different locations.

Sink

Sinks act as a connector between two units, facilitating feeding the output of one as input into the other. See *Sinks* for a full description.

Sinks are special in that they don't represent a type that's generally available for instantiation. Instead they need to be declared as the member of unit using the special `sink` keyword. You can, however, maintain references to sinks by assigning the unit member to a variable of type `Sink&`.

Methods

`close()` → `void`
Closes a sink by disconnecting all parsing units. Afterwards the sink's state is as if it had just been created (so new units can be connected). Note that a sink is automatically closed when the unit it is part of is done parsing. Also note that a previously connected parsing unit can *not* be reconnected; trying to do so will still throw a `UnitAlreadyConnected` exception.

`connect(u: strong_ref<unit>)` → `void`
Connects a parsing unit to a sink. All subsequent write operations to the sink will pass their data on to this parsing unit. Each unit can only be connected to a single sink. If the unit is already connected, a `UnitAlreadyConnected` exception is thrown. However, a sink can have more than one unit connected to it.

`connect_filter(filter: strong_ref<unit>)` → `void`
Connects a filter unit to the sink that will transform its input transparently before forwarding it for parsing to other connected units.

Multiple filters can be added to a sink, in which case they will be chained into a pipeline and the data will be passed through them in the order they have been added. The parsing will then be carried out on the output of the last filter in the chain.

Filters must be added before the first data chunk is written into the sink. If data has already been written when a filter is added, an error is triggered.

`connect_mime_type(mt: bytes)` → `void`
Connects parsing units to a sink for all parsers that support a given MIME type. All subsequent write operations to the sink will pass their data on to these parsing units. The MIME type may have wildcards for type or subtype, and the method will then connect units for all matching parsers.

`connect_mime_type(mt: string) → void`
Connects parsing units to a sink for all parsers that support a given MIME type. All subsequent write operations to the sink will pass their data on to these parsing units. The MIME type may have wildcards for type or subtype, and the method will then connect units for all matching parsers.

`gap(seq: uint<64>, len: uint<64>) → void`
Reports a gap in the input stream. *seq* is the sequence number of the first byte missing, *len* is the length of the gap.

`sequence_number() → uint<64>`
Returns the current sequence number of the sink's input stream, which is one beyond the index of the last byte that has been put in order and delivered so far.

`set_auto_trim(enable: bool) → void`
Enables or disables auto-trimming. If enabled (which is the default) sink input data is trimmed automatically once in-order and processed. See `trim()` for more information about trimming.

`set_initial_sequence_number(seq: uint<64>) → void`
Sets the sink's initial sequence number. All sequence numbers given to other methods are then assumed to be absolute numbers beyond that initial number. If the initial number is not set, the sink implicitly uses zero instead.

`set_policy(policy: enum) → void`
Sets a sink's reassembly policy for ambiguous input. As long as data hasn't been trimmed, a sink will detect overlapping chunks. This policy decides how to handle ambiguous overlaps. The default (and currently only) policy is `ReassemblerPolicy::First`, which resolves ambiguities by taking the data from the chunk that came first.

`skip(seq: uint<64>) → void`
Skips ahead in the input stream. *seq* is the sequence number where to continue parsing. If there's still data buffered before that position it will be ignored; if auto-skip is also active, it will be immediately deleted as well. If new data is passed in later that comes before *seq*, that will likewise be ignored. If the input stream is currently stuck inside a gap, and *seq* lies beyond that gap, the stream will resume processing at *seq*.

`trim(seq: uint<64>) → void`
Deletes all data that's still buffered internally up to *seq*. If processing the input stream hasn't reached *seq* yet, parsing will also skip ahead to *seq*.

Trimming the input stream releases the memory, but that means that the sink won't be able to detect any further data mismatches.

Note that by default, auto-trimming is enabled, which means all data is trimmed automatically once in-order and processed.

`write(data: bytes, [ seq: uint<64> ], [ len: uint<64> ]) → void`
Passes data on to all connected parsing units. Multiple *write* calls act like passing input in incrementally: The units will parse the pieces as if they were a single stream of data. If no sequence number *seq* is provided, the data is assumed to represent a chunk to be appended to the current end of the input stream. If a sequence number is provided, out-of-order data will be buffered and reassembled before being passed on. If *len* is provided, the data is assumed to represent that many bytes inside the sequence space; if not provided, *len* defaults to the length of *data*.

If no units are connected, the call does not have any effect. If multiple units are connected and one parsing unit throws an exception, parsing of subsequent units does not proceed. Note that the order in which the data is parsed to each unit is undefined.

Todo: The error semantics for multiple units aren't great.

Operators

`|sink|` → `uint<64>`

Returns the number of bytes written into the sink so far. If the sink has filters attached, this returns the value after filtering.

`|strong_ref<sink>|` → `uint<64>`

Returns the number of bytes written into the referenced sink so far. If the sink has filters attached, this returns the value after filtering.

Sinks provide a set of dedicated unit hooks as callbacks for the reassembly process. These must be implemented on the reader side, i.e., the unit that's connected to a sink.

`%on_gap(seq: uint64, len: uint64)`

`%on_overlap(seq: uint64, old: data, new: data)`

Triggered when reassembly encounters a 2nd version of data for sequence space already covered earlier. *seq* is the start of the overlap, and *old*/*new* the previous and the new data, respectively. This hook is just for informational purposes, the policy set with `set_policy()` determines how the reassembler handles the overlap.

`%on_skipped(seq: uint64)`

Any time `skip()` moves ahead in the input stream, this hook reports the new sequence number *seq*.

`%on_skipped(seq: uint64, data: bytes)`

If data still buffered is skipped over through `skip()`, it will be passed to this hook, before adjusting the current position. *seq* is the starting sequence number of the data, *data* is the data itself.

Stream

A `stream` is data structure that efficiently represents a potentially large, incrementally provided input stream of raw data. You can think of it as a *bytes* type that's optimized for (1) efficiently appending new chunks of data at the end, and (2) trimming data no longer needed at the beginning. Other than those two operation, stream data cannot be modified; there's no way to change the actual content of a stream once it has been added to it. Streams provide *iterators* for traversal, and *views* for limiting visibility to smaller windows into the total stream.

Streams are key to Spicy's parsing process, although most of that happens behind the scenes. You will most likely encounter them when using *Random access*. They may also be useful for buffering larger volumes of data during processing.

Types

- `stream`
- `iterator<stream>`
- `view<stream>`

Methods

`at(i: uint<64>) → iterator<stream>`
Returns an iterator representing the offset *i* inside the stream value.

`freeze() → void`
Freezes the stream value. Once frozen, one cannot append any more data to a frozen stream value (unless it gets unfrozen first). If the value is already frozen, the operation does not change anything.

`is_frozen() → bool`
Returns true if the stream value has been frozen.

`trim(inout i: iterator<stream>) → void`
Trims the stream value by removing all data from its beginning up to (but not including) the position *i*. The iterator *i* will remain valid afterwards and will still point to the same location, which will now be the beginning of the stream's value. All existing iterators pointing to *i* or beyond will remain valid and keep their offsets as well. The effect of this operation is undefined if *i* does not actually refer to a location inside the stream value. Trimming is permitted even on frozen values.

`unfreeze() → void`
Unfreezes the stream value. A unfrozen stream value can be further modified. If the value is already unfrozen (which is the default), the operation does not change anything.

Operators

`begin(<container>) → <iterator>`
Returns an iterator to the beginning of the container's content.

`stream(bytes) → stream`
Creates a stream instance preinitialized with the given data.

`end(<container>) → <iterator>`
Returns an iterator to the end of the container's content.

`|stream| → uint<64>`
Returns the number of stream the value contains.

`stream += bytes → stream`
Concatenates data to the stream.

`stream += view<stream> → stream`
Concatenates another stream's view to the target stream.

`stream != stream → bool`
Compares two stream values lexicographically.

Iterator Methods

`is_frozen() → bool`
Returns whether the stream value that the iterator refers to has been frozen.

`offset() → uint<64>`
Returns the offset of the byte that the iterator refers to relative to the beginning of the underlying stream value.

Iterator Operators

`*iterator<stream>` → `uint<64>`

Returns the character the iterator is pointing to.

`iterator<stream> - iterator<stream>` → `int<64>`

Returns the number of stream between the two iterators. The result will be negative if the second iterator points to a location before the first. The result is undefined if the iterators do not refer to the same stream instance.

`iterator<stream> == iterator<stream>` → `bool`

Compares the two positions. The result is undefined if they are not referring to the same stream value.

`iterator<stream> > iterator<stream>` → `bool`

Compares the two positions. The result is undefined if they are not referring to the same stream value.

`iterator<stream> >= iterator<stream>` → `bool`

Compares the two positions. The result is undefined if they are not referring to the same stream value.

`iterator<stream>++` → `iterator<stream>`

Advances the iterator by one byte, returning the previous position.

`++iterator<stream>` → `iterator<stream>`

Advances the iterator by one byte, returning the new position.

`iterator<stream> < iterator<stream>` → `bool`

Compares the two positions. The result is undefined if they are not referring to the same stream value.

`iterator<stream> <= iterator<stream>` → `bool`

Compares the two positions. The result is undefined if they are not referring to the same stream value.

`iterator<stream> + uint<64>` → `iterator<stream>` (commutative)

Advances the iterator by the given number of stream.

`iterator<stream> += uint<64>` → `iterator<stream>`

Advances the iterator by the given number of stream.

`iterator<stream> != iterator<stream>` → `bool`

Compares the two positions. The result is undefined if they are not referring to the same stream value.

View Methods

`advance(i: uint<64>)` → `view<stream>`

Advances the view's starting position by *i* stream, returning the new view.

`advance(inout i: iterator<stream>)` → `view<stream>`

Advances the view's starting position to a given iterator *i*, returning the new view. The iterator must be referring to the same stream values as the view, and it must be equal or ahead of the view's starting position.

`advance_to_next_data()` → `view<stream>`

Advances the view's starting position to the next non-gap position. This always advances the input by at least one byte.

`at(i: uint<64>)` → `iterator<stream>`

Returns an iterator representing the offset *i* inside the view.

`find(needle: bytes)` → `tuple<bool, iterator<stream>>`

Searches *needle* inside the view's content. Returns a tuple of a boolean and an iterator. If *needle* was found, the boolean will be true and the iterator will point to its first occurrence. If *needle* was not found, the boolean will be false and the iterator will point to the last position so that everything before that is guaranteed to not contain even a partial match of *needle* (in other words: one can trim until that position and then restart the search from there if more data gets appended to the underlying stream value). Note that for a simple yes/no result, you should use the `in` operator instead of this method, as it's more efficient.

`limit(i: uint<64>)` → `view<stream>`
Returns a new view that keeps the current start but cuts off the end *i* characters from that beginning. The returned view will not be able to expand any further.

`offset()` → `uint<64>`
Returns the offset of the view's starting position within the associated stream value.

`starts_with(b: bytes)` → `bool`
Returns true if the view starts with *b*.

`sub(begin: uint<64>, end: uint<64>)` → `view<stream>`
Returns a new view of the subsequence from offset *begin* to (but not including) offset *end*. The offsets are relative to the beginning of the view.

`sub(inout begin: iterator<stream>, inout end: iterator<stream>)` → `view<stream>`
Returns a new view of the subsequence from *begin* up to (but not including) *end*.

`sub(inout end: iterator<stream>)` → `view<stream>`
Returns a new view of the subsequence from the beginning of the stream up to (but not including) *end*.

View Operators

`view<stream> == bytes` → `bool` (commutative)
Compares a stream view and a bytes instance lexicographically.

`view<stream> == view<stream>` → `bool`
Compares the views lexicographically.

`bytes in view<stream>` → `bool`
Returns true if the right-hand-side bytes contains the left-hand-side view as a subsequence.

`view<stream> in bytes` → `bool`
Returns true if the right-hand-side view contains the left-hand-side bytes as a subsequence.

`bytes !in view<stream>` → `bool`
Performs the inverse of the corresponding `in` operation.

`view<stream> !in bytes` → `bool`
Performs the inverse of the corresponding `in` operation.

`|view<stream>|` → `uint<64>`
Returns the number of stream the view contains.

`view<stream> != bytes` → `bool` (commutative)
Compares a stream view and a bytes instance lexicographically.

`view<stream> != view<stream>` → `bool`
Compares two views lexicographically.

String

Strings store readable text that's associated with a given character set. Internally, Spicy stores them as UTF-8.

Type

- `string`

Constants

- `"Spicy"`, `""`
- When specifying string constants, Spicy assumes them to be in UTF-8.

Methods

`encode(charset: enum = hilti::Charset::UTF8)` → `bytes`
Converts the string into a binary representation encoded with the given character set.

Operators

`string == string` → `bool`
Compares two strings lexicographically.

`string % <any>` → `string`
Renders a printf-style format string.

`|string|` → `uint<64>`
Returns the number of characters the string contains.

`string + string` → `string`
Returns the concatenation of two strings.

`string != string` → `bool`
Compares two strings lexicographically.

Struct

A struct is a heterogeneous container of an ordered set of named values similar to a *Tuple*. In contrast to `tuple` elements, `struct` fields are mutable.

Type

- `struct { IDENTIFIER_1: TYPE_1; ...; IDENTIFIER_N: TYPE_N; }`

Constants

- Structs can be initialized with a `struct` initializer, `local my_struct:  MyStruct = [$FIELD_1 = X_1, ..., $FIELD_N = X_N]` where `FIELD_I` is the label of the corresponding field in `MyStruct`'s type.

Operators

`struct ?. <field>` → `bool`

Returns true if the struct's field has a value assigned (not counting any `&default`).

`struct . <field>` → `<field type>`

Retrieves the value of a struct's field. If the field does not have a value assigned, it returns its `&default` expression if that has been defined; otherwise it triggers an exception.

`struct .? <field>` → `<field type>`

Retrieves the value of a struct's field. If the field does not have a value assigned, it returns its `&default` expression if that has been defined; otherwise it signals a special non-error exception to the host application (which will normally still lead to aborting execution, similar to the standard dereference operator, unless the host application specifically handles this exception differently).

`unset struct.<field>` → `void`

Clears an optional field.

Time

A time value refers to a specific, absolute point of time, specified as the interval from January 1, 1970 UT (i.e., the Unix epoch). Times are stored with nanosecond resolution, which is retained across all calculations.

Type

- `time`

Constants

- `time(SECS)` creates a time from an unsigned integer or real value `SECS` specifying seconds since the epoch.
- `time_ns(NSECS)` creates a time from an unsigned integer value `NSECS` specifying nanoseconds since the epoch.

Methods

`nanoseconds()` → `uint<64>`

Returns the time as an integer value representing nanoseconds since the UNIX epoch.

`seconds()` → `real`

Returns the time as a real value representing seconds since the UNIX epoch.

Operators

`time(int)` → `time`
: Creates an time interpreting the argument as number of seconds.

`time(real)` → `time`
: Creates an time interpreting the argument as number of seconds.

`time(uint)` → `time`
: Creates an time interpreting the argument as number of seconds.

`time_ns(int)` → `time`
: Creates an time interpreting the argument as number of nanoseconds.

`time_ns(uint)` → `time`
: Creates an time interpreting the argument as number of nanoseconds.

`time - time` → `interval`
: Returns the difference of the times.

`time - interval` → `time`
: Subtracts the interval from the time.

`time == time` → `bool`
: Compares two time values.

`time > time` → `bool`
: Compares the times.

`time >= time` → `bool`
: Compares the times.

`time < time` → `bool`
: Compares the times.

`time <= time` → `bool`
: Compares the times.

`time + interval` → `time` $^{\text{(commutative)}}$
: Adds the interval to the time.

`time != time` → `bool`
: Compares two time values.

Tuple

Tuples are heterogeneous containers of a fixed, ordered set of types. Tuple elements may optionally be declared and addressed with custom identifier names. Tuple elements are immutable.

Type

- `tuple<[IDENTIFIER_1: ]TYPE_1, ...[IDENTIFIER_N: ]TYPE_N>`

Constants

- `(1, "string", True)`, `(1, )`, `()`
- `tuple(1, "string", True)`, `tuple(1)`, `tuple()`

Operators

`(x, ..., y)=<tuple>` → `<tuple>`
Assigns element-wise to the left-hand-side tuple

`tuple == tuple` → `bool`
Compares two tuples element-wise.

`tuple[uint<64>]` → `<type of element>`
Extracts the tuple element at the given index. The index must be a constant unsigned integer.

`tuple . <id>` → `<type of element>`
Extracts the tuple element corresponding to the given ID.

`tuple != tuple` → `bool`
Compares two tuples element-wise.

Unit

Type

- `unit { FIELD_1; ...; FIELD_N }`
- See *Parsing* for a full discussion of unit types.

Constants

- Spicy doesn't support unit constants, but you can initialize unit instances through coercion from a `struct` initializer, see *Struct*.

Todo: This initialization isn't actually available in Spicy yet (#%s1036).

Methods

`backtrack()` → `void`
Aborts parsing at the current position and returns back to the most recent `&try` attribute. Turns into a parse error if there's no `&try` in scope.

`connect_filter(filter: strong_ref<unit>)` → `void`
Connects a separate filter unit to transform the unit's input transparently before parsing. The filter unit will see the original input, and this unit will receive everything the filter passes on through `forward()`.

Filters can be connected only before a unit's parsing begins. The latest possible point is from inside the target unit's `%init` hook.

`context()` → `<context>&`
: Returns a reference to the `%context` instance associated with the unit.

`find(needle: bytes, [ dir: enum ], [ start: iterator<stream> ])` → `optional<iterator<stre`
: Searches a *needle* pattern inside the input region defined by where the unit began parsing and its current parsing position. If executed from inside a field hook, the current parasing position will represent the *first* byte that the field has been parsed from. By default, the search will start at the beginning of that region and scan forward. If the direction is `spicy::Direcction::Backward`, the search will start at the end of the region and scan backward. In either case, a starting position can also be explicitly given, but must lie inside the same region.

`forward(inout data: bytes)` → `void`
: If the unit is connected as a filter to another one, this method forwards transformed input over to that other one to parse. If the unit is not connected, this method will silently discard the data.

`forward_eod()` → `void`
: If the unit is connected as a filter to another one, this method signals that other one that end of its input has been reached. If the unit is not connected, this method will not do anything.

`input()` → `iterator<stream>`
: Returns an iterator referring to the input location where the current unit has begun parsing. If this method is called before the units parsing has begun, it will throw a runtime exception. Once available, the input position will remain accessible for the unit's entire life time.

`offset()` → `uint<64>`
: Returns the offset of the current location in the input stream relative to the unit's start. If executed from inside a field hook, the offset will represent the first byte that the field has been parsed from. If this method is called before the unit's parsing has begun, it will throw a runtime exception. Once parsing has started, the offset will remain available for the unit's entire life time.

`position()` → `iterator<stream>`
: Returns an iterator to the current position in the unit's input stream. If executed from inside a field hook, the position will represent the first byte that the field has been parsed from. If this method is called before the unit's parsing has begun, it will throw a runtime exception.

`set_input(i: iterator<stream>)` → `void`
: Moves the current parsing position to *i*. The iterator *i* must be into the input of the current unit, or the method will throw a runtime exception.

Operators

`unit ?. <field>` → `bool`
: Returns true if the unit's field has a value assigned (not counting any `&default`).

`unit . <field>` → `<field type>`
: Retrieves the value of a unit's field. If the field does not have a value assigned, it returns its `&default` expression if that has been defined; otherwise it triggers an exception.

`unit .? <field>` → `<field type>`
: Retrieves the value of a unit's field. If the field does not have a value assigned, it returns its `&default` expression if that has been defined; otherwise it signals a special non-error exception to the host application (which will normally still lead to aborting execution, similar to the standard dereference operator, unless the host application specifically handles this exception differently).

`unset unit.<field>` → `void`
: Clears an optional field.

Vector

Vectors are homogeneous containers, holding a set of elements of a given element type. They provide iterators to traverse their content.

Types

- `vector<T>` specifies a vector with elements of type `T`.
- `iterator<vector<T>>`

Constants

- `vector(E_1, E_2, ..., E_N)` creates a vector of `N` elements. The values `E_I` must all have the same type. `vector()` creates an empty vector of unknown element type; this cannot be used directly but must be coerced into a fully-defined vector type first.
- `vector<T>(E_1, E_2, ..., E_N)` creates a vector of type `T`, initializing it with the `N` elements `E_I`. `vector<T>()` creates an empty vector.
- Vectors can be initialized through coercion from a list value: `vector<string> I = ["A", "B", "C"]`.

Methods

`assign(i: uint<64>, x: <any>)` → `void`
: Assigns *x* to the i*th element of the vector. If the vector contains less than *i elements a sufficient number of default-initialized elements is added to carry out the assignment.

`at(i: uint<64>)` → `<iterator>`
: Returns an iterator referring to the element at vector index *i*.

`back()` → `<type of element>`
: Returns the last element of the vector. It throws an exception if the vector is empty.

`front()` → `<type of element>`
: Returns the first element of the vector. It throws an exception if the vector is empty.

`pop_back()` → `void`
: Removes the last element from the vector, which must be non-empty.

`push_back(x: <any>)` → `void`
: Appends *x* to the end of the vector.

`reserve(n: uint<64>)` → `void`
: Reserves space for at least *n* elements. This operation does not change the vector in any observable way but provides a hint about the size that will be needed.

`resize(n: uint<64>)` → `void`
: Resizes the vector to hold exactly *n* elements. If *n* is larger than the current size, the new slots are filled with default values. If *n* is smaller than the current size, the excessive elements are removed.

`sub(begin: uint<64>, end: uint<64>)` → `vector`
: Extracts a subsequence of vector elements spanning from index *begin* to (but not including) index *end*.

`sub(end: uint<64>)` → `vector`
: Extracts a subsequence of vector elements spanning from the beginning to (but not including) the index *end* as a new vector.

Operators

`begin(<container>)` → `<iterator>`
Returns an iterator to the beginning of the container's content.

`end(<container>)` → `<iterator>`
Returns an iterator to the end of the container's content.

`vector == vector` → `bool`
Compares two vectors element-wise.

`vector[uint<64>]` → `<type of element>`
Returns the vector element at the given index.

`|vector|` → `uint<64>`
Returns the number of elements a vector contains.

`vector + vector` → `vector`
Returns the concatenation of two vectors.

`vector += vector` → `vector`
Concatenates another vector to the vector.

`vector != vector` → `bool`
Compares two vectors element-wise.

Iterator Operators

`*iterator<vector>` → `<dereferenced type>`
Returns the vector element that the iterator refers to.

`iterator<vector> == iterator<vector>` → `bool`
Returns true if two vector iterators refer to the same location.

`iterator<vector>++` → `iterator<vector>`
Advances the iterator by one vector element, returning the previous position.

`++iterator<vector>` → `iterator<vector>`
Advances the iterator by one vector element, returning the new position.

`iterator<vector> != iterator<vector>` → `bool`
Returns true if two vector iterators refer to different locations.

Void

The void type is place holder for specifying "no type", such as when a function doesn't return anything.

Type

- `void`

Statements

Most of Spicy's statements are pretty standard stuff. We summarize them briefly in the following.

assert

```
assert EXPR;

assert EXPR : MSG;
```

Ensures at runtime that `EXPR` evaluates to a `True` value. If it doesn't, an exception gets thrown that will typically abort execution. `EXPR` must either be of boolean type to begin with, or support coercion into it. If `MSG` is specified, it must be a string and will be carried along with the exception.

break

```
break;
```

Inside a *for* or *while* loop, `break` aborts the loop's body, with execution then continuing right after the loop construct.

confirm

```
confirm;
```

If the parser is currently in trial mode, confirm that the unit is successfully synchronized to the input; the unit is then put into regular parsing mode again. If the unit is not in trial mode `confirm` has no effect.

See *reject* to reject the synchronization instead.

`confirm` can only be invoked from hooks.

for

```
for ( ID in ITERABLE )
    BLOCK
```

Loops over all the elements of an iterable value. `ID` is an identifier that will become local variable inside `BLOCK`, with the current loop element assigned on each round. `ITERABLE` is a value of any type that provides iterators.

Examples:

```
module Test;

for ( i in [1, 2, 3] )
    print i;

for ( i in b"abc" ) {
    print i;
}

local v = vector("a", "b", "c");
```

(continues on next page)

(continued from previous page)

```
for ( i in v )
    print i;
```

```
# spicyc -j for.spicy
1
2
3
97
98
99
a
b
c
```

if

```
if ( EXPR )
    BLOCK

if ( EXPR )
    BLOCK
else
    BLOCK
```

A classic `if`-statement branching based on a boolean expression `EXPR`.

import

```
import MODULE;
```

Makes the content of another module available, see *Modules* for more.

print

```
print EXPR;

print EXPR_1, ..., EXPR_N;
```

Prints one or more expressions to standard output. This is supported for expressions of any type, with each type knowing how to render its values into a readable representation. If multiple expressions are specified, commas will separate them in the output.

Note: A particular use-case combines `print` with string interpolation (i.e., *`string::Modulo`*):

```
module Test;

print "Hello, %s!" % "World";
print "%s=%d" % ("x", 1);
```

```
# spicyc -j print.spicy
Hello, World!
x=1
```

reject

```
reject;
```

If the parse is currently in trial mode, reject the synchronization; this immediately fails parsing of the unit and raises the parse error which caused the unit to be put into trial mode. If the unit is not in trial mode this triggers a generic parse error.

See *confirm* to confirm the synchronization instead.

`reject` can only be invoked from hooks.

return

```
return;

return EXPR;
```

Inside a function or hook, `return` yields control back to the caller. If it's a function with a non-void return value, the return must provide a corresponding `EXPR`.

stop

```
stop;
```

Inside a `foreach` container hook (see *here*), aborts the parsing loop without adding the current (final) value to the container.

switch

```
switch ( [local IDENT =] CTRL_EXPR ) {
    case EXPR [, ..., EXPR]:
        BLOCK;

    ...

    case EXPR [, ..., EXPR]:
        BLOCK;

   [default:
        BLOCK]
}
```

Branches across a set of alternatives based on the value of an control expression. `CTRL_EXPR` is compared against all the `case` expressions through the type's equality operator, coercing `CTRL_EXPR` accordingly first where necessary.

If `local IDENT` is specified, the blocks have access to a corresponding local variable that holds the value of the control expression. If no `default` is given, the runtime will throw an `UnhandledSwitchCase` exception if there's no matching case.

Note: Don't confuse the `switch` statement with the unit type's *switch parsing construct*. They look similar, but do different things.

`throw`

```
throw EXPR;
```

Triggers a parse error exception with the message indicated by `EXPR`. `EXPR` needs to be a *String*. `throw` aborts parsing.

`try/catch`

Todo: This isn't available in Spicy yet (#%s89).

```
try
    BLOCK

catch [(TYPE IDENT)]
    BLOCK

...

catch [(TYPE IDENT)]
    BLOCK
```

Catches any exception thrown in the `try` block that match one of the types in any of `catch` headers, which must be *Exception* types. A `catch` without a type matches any exception. If no `catch` matches an exception thrown in the `try` block, it'll be propagated further up the stack. A bare `throw` statement can be used inside a `catch` block to rethrow the current exception.

`while`

```
while ( COND )
    BLOCK

while ( local IDENT = EXPR; COND )
    BLOCK
```

`while` introduces a loop that executes `BLOCK` for as long as the boolean `COND` evaluates to true. The second form initializes a new local variable `IDENT` with `EXPR`, and makes it available inside both `COND` and `BLOCK`.

Packing / Unpacking Values

A few of Spicy's atomic types support low-level conversion from, or into, a binary representation through two operators:

- `pack(VALUE, ARGS)` turns a `VALUE` into a sequence of raw bytes representing the value in binary form. `ARGS` specify what encoding to use for the representation; they are type-specific (see below for a list). The `pack` operator returns a `bytes` instance containing the encoded data.

  As an example, `pack(uint16(513), spicy::ByteOrder::Network)` returns `\x02\x01`, which is 513 in network byte order.

- `unpack<TYPE>(DATA, ARGS)` parses a value of type `TYPE` from a binary representation `DATA`. `ARGS` specify what encoding to expect for `DATA`. The `unpack` operator returns a 2-tuple `(VALUE, REMAINDER)` where `VALUE` is the parsed value, and `REMAINDER` is any bytes left over from `DATA` that weren't used for parsing the value. If parsing fails, `unpack` throws an `InvalidValue` exception.

  As an example, `unpack<uint16>(b"\x02\x01XYZ", spicy::ByteOrder::Network)` returns the tuple `(513, b"XYZ")`.

The following table summarizes the types that currently support packing/unpacking, along with the encoding arguments that the operators expect for each:

Type	pack	unpack	Links
Address	`pack(VALUE, spicy::ByteOrder)`	`unpack<addr>(DATA, spicy::ByteOrder)`	*Address Family*, *Byte Order*
Integer	`pack(VALUE, spicy::ByteOrder)`	`unpack<uintX|intX>(DATA, spicy::ByteOrder)`	*Address Family*, *Byte Order*
Real	`pack(VALUE, spicy::RealType, spicy::ByteOrder)`	`unpack<uintX|intX>(DATA, spicy::RealType, spicy::ByteOrder)`	*Real Type* [1], *Byte Order*

Note: [1] Packing a `real` value as `IEEE754_Single` may loose information.

Error Handling

Todo: Spicy's error handling remains quite limited at this point, with more to come here in the future.

Exceptions

Exceptions provide Spicy's primary mechanism for reporting errors. Currently, various parts of the runtime system throw exceptions if they encounter unexpected situations. In particular, the generated parsers throw `ParsingError` exceptions if they find themselves unable to comprehend their input. However, the support for catching and handling exception is remains minimal at the moment. For now, only `ParsingError` exceptions can be caught indirectly through the *%on_error* unit hook, which internally is nothing else than an exception handler.

Todo: Support for catching other exception throughs *try/catch* needs to be added still (#%s89).

`result<T>` / `error`

Todo: Spicy doesn't have `result`/`error` yet (#%s90).

Error recovery

Support for resynchronizing parser with their input stream after parse errors is discussed in the section on *error recovery*.

Conditional Compilation

Spicy scripts offer a basic form of conditional compilation through `@if`/`@else`/`@endif` blocks, similar to a C preprocessor. For now, this supports only a couple types of conditions that are useful for feature and version testing. For example, the following `@if`/`@else` block branches to different code based on the Spicy version:

```
@if SPICY_VERSION < 10401
    <code for Spicy versions older than 1.4.1>
@else
    <code for Spicy versions equal or newer than 1.4.1>
@endif
```

`@if` directives can take one of the following forms:

`@if [!] IDENTIFIER OPERATOR VALUE` Compares the value of `IDENTIFIER` against `VALUE`. Supported comparison operators are `==`, `!=`, `<`, `<=`, `>`, `>=`. See below for valid identifiers. If an identifier is not defined, its value is assumed to be zero.

`@if [!] IDENTIFIER` This is a shortcut for `@if [!] IDENTIFIER != 0`.

By default, Spicy currently provides just one pre-defined identifier:

`SPICY_VERSION` The current Spicy version in numerical format (e.g., 10000 for version 1.0; see the output of `spicy-config --version-number`).

The Spicy plugin for Zeek defines a couple of *additional identifiers*.

Appendix

Reserved Keywords

The following is a list of keywords reserved by the Spicy language. They cannot be used as identifiers.

```
False
None
Null
True
__library_type
add
addr
any
assert
assert-exception
attribute
```

(continues on next page)

(continued from previous page)

```
begin
bitfield
bool
break
bytes
case
cast
catch
confirm
const
const_iterator
constant
continue
cregexp
cstring
default
delete
else
end
enum
exception
export
file
for
foreach
from
function
global
ident
if
import
in
inout
int16
int32
int64
int8
interval
interval_ns
iterator
list
local
map
mark
mod
module
net
new
object
on
optional
pack
port
print
priority
private
property
```

(continues on next page)

(continued from previous page)

```
public
real
regexp
reject
return
set
sink
stop
stream
string
struct
switch
throw
time
time_ns
timer
try
tuple
type
uint16
uint32
uint64
uint8
unit
unpack
unset
var
vector
view
void
while
```

2.5.3 Library

Module spicy

Types

spicy::AddressFamily

Specifies an address' IP family.

```
type AddressFamily = enum {
    IPv4, # IP4 address
    IPv6  # IPv6 address
};
```

spicy::Base64Stream

Captures the state of base64 encoding/decoding for the corresponding library functions.

spicy::BitOrder

Specifies the bit order for individual bit ranges inside a bitfield.

```
type BitOrder = enum {
    LSB0,     # bits are interpreted as lowest-significant-bit coming first
    MSB0      # bits are interpreted as most-significant-bit coming first
};
```

spicy::ByteOrder

Specifies byte order for data operations.

```
type ByteOrder = enum {
    Little,   # data is in little-endian byte order
    Big,      # data is in big-endian byte order
    Network,  # data is in network byte order (same a big endian)
    Host      # data is in byte order of the host we are executing on
};
```

spicy::Charset

Specifies the character set for bytes encoding/decoding.

```
type Charset = enum {
    ASCII,
    UTF8
};
```

spicy::DecodeErrorStrategy

Specifies how data is handled that’s not representable in a specified character set.

```
type DecodeErrorStrategy = enum {
    IGNORE,  # data is skipped but processing continues
    REPLACE, # data is replaced with a valid place-holder and processing continues
    STRICT   # runtime error is triggered
};
```

spicy::MatchState

Captures state for incremental regular expression matching.

spicy::Protocol

Specifies a transport-layer protocol.

```
type Protocol = enum {
    TCP,
    UDP,
    ICMP
};
```

spicy::RealType

Specifies the type of a real value.

```
type RealType = enum {
    IEEE754_Single, # single precision in IEEE754 format
    IEEE754_Double  # double precision in IEEE754 format
};
```

spicy::ReassemblerPolicy

Specifies the policy for a sink's reassembler when encountering overlapping data.

```
type ReassemblerPolicy = enum {
    First # take the original data & discard the new data
};
```

spicy::Side

Specifies a side an operation should operate on.

```
type Side = enum {
    Left,    # operate on left side
    Right,   # operate on right side
    Both     # operate on both sides
};
```

spicy::Direction

Specifies direction of a search.

```
type Direction = enum {
    Forward, # search forward
    Backward, # search backward
};
```

spicy::ZlibStream

Captures the state of gzip decompression for the corresponding library functions.

Functions

function spicy::zlib_init(window_bits: int64) : ZlibStream

Initializes a zlib stream for decompression.

`window_bits`: Same as the corresponding parameter for zlib's *inflateInit2* (see https://www.zlib.net/manual.html).

Will throw a *ZlibError* exception if initialization fails.

function spicy::zlib_decompress(inout stream_: ZlibStream, data: bytes) : bytes

Decompresses a chunk of data through the given zlib stream.

function spicy::zlib_finish(inout stream_: ZlibStream) : bytes

Finalizes a zlib stream used for decompression.

function spicy::base64_encode(inout stream_: Base64Stream, data: bytes) : bytes

Encodes a stream of data into base64.

function spicy::base64_decode(inout stream_: Base64Stream, data: bytes) : bytes

Decodes a stream of base64 data back into the clear.

function spicy::base64_finish(inout stream_: Base64Stream) : bytes

Finalizes a base64 stream used for decoding or encoding.

function spicy::crc32_init() : uint64

Returns the initialization value for CRC32 computation.

function spicy::crc32_add(crc: uint64, data: bytes) : uint64

Computes a running CRC32.

function spicy::current_time() : time

Returns the current wall clock time.

function spicy::mktime(y: uint64, m: uint64, d: uint64, H: uint64, M: uint64, S: uint64) : time

Constructs a time value from a tuple of broken-out elements specifying local time.

- *y*: year (1970-. . .)
- *m*: month (1-12)
- *d*: day (1-31)
- *H*: hour (0-23)
- *M*: minute (0-59)
- *S*: second (0-59)

function spicy::bytes_to_hexstring(value: bytes) : string

Returns a bytes value rendered as a hex string.

function spicy::getenv(name: string) : optional<string>

Returns the value of an environment variable, if set.

function spicy::strftime(format: string, timestamp: time) : string

Formats a time according to user-specified format string.

This function uses the currently active locale and timezone to format values. Formatted strings cannot exceed 128 bytes.

The format string can contain format specifiers supported by POSIX strftime, see https://pubs.opengroup.org/onlinepubs/009695399/functions/strftime.html.

This function can raise `InvalidArgument` if the timestamp could not be converted to local time or formatted.

function spicy::strptime(buf: string, format: string) : time

Parses time from a string.

This function uses the currently active locale and timezone to parse values.

The format string can contain format specifiers supported by POSIX strptime, see https://pubs.opengroup.org/onlinepubs/009695399/functions/strptime.html.

This function raises `InvalidArgument` if the string could not be parsed with the given format string, or `OutOfRange` if the parsed time value cannot be represented.

function spicy::parse_address(s: string) : addr

Parses an address from a string. The address can be in standard IPv4 or IPv6 ASCII represententation. The function raises `InvalidArgument` if the string could not be parsed.

function spicy::parse_address(b: bytes) : addr

Parses an address from a bytes instance. The address can be in standard IPv4 or IPv6 ASCII represententation. The function raises `InvalidArgument` if the string could not be parsed.

function spicy::accept_input()

Reports a confirmation to the host application indicating that the parser appears to be processing the expected input format. It's up to the host application how to use this information.

function spicy::decline_input(reason: string)

Reports a violation to the host application indicating that the parser appears not to be processing the expected input format. It's up to the host application how to use this information.

Note that this does not automatically abort processing. If that's desired, you need to trigger a parse error as well, e.g., by throwing an error, *throw "<error message>"*.

reason: user-presentable description of why the input seems wrong

Module `filter`

Types

spicy::Zlib

A filter that performs zlib decompression.

```
type Zlib = unit;
```

`spicy::Base64Decode`

A filter that performs Base64 decoding.

```
type Base64Decode = unit;
```

2.5.4 Examples

We collect some example Spicy parsers here that come with a growing collection of Spicy-based Zeek analyzers. Check out that package and its dependencies for more examples.

TFTP

A TFTP analyzer for Zeek, implementing the original RFC 1350 protocol (no extensions). It comes with a Zeek script producing a typical `tftp.log` log file.

This analyzer is a good introductory example because the Spicy side is pretty straight-forward. The Zeek-side logging is more tricky because of the data transfer happening over a separate network session.

- TFTP Spicy grammar
- TFTP Zeek analyzer definition (EVT)
- Zeek TFTP script for logging

HTTP

A nearly complete HTTP parser. This parser was used with the original Spicy prototype to compare output with Zeek's native handwritten HTTP parser. We observed only negligible differences.

- HTTP Spicy grammar
- Spicy code for HTTP analyzer Zeek integration
- HTTP Zeek analyzer definition (EVT)

DNS

A comprehensive DNS parser. This parser was used with the original Spicy prototype to compare output with Zeek's native handwritten DNS parser. We observed only negligible differences.

The DNS parser is a good example of using *random access*.

- DNS Spicy grammar
- Spicy code for DNS analyzer Zeek integration
- DNS Zeek analyzer definition (EVT)

DHCP

A nearly complete DHCP parser. This parser extracts most DHCP option messages understood by Zeek. The Zeek integration is almost direct and most of the work is in formulating the parser itself.

- DHCP Spicy grammar
- Spicy code for DHCP analyzer Zeek integration
- DHCP analyzer Zeek analyzer definition (EVT)

2.5.5 Debugging

It can be challenging to track down the specifics of what a parser is doing (or not doing) because often there's no directly observable effect. To make that easier, Spicy comes with debugging support that helps during parser development.

Generally, debugging support requires running `spicyc` or `spicy-driver` with option `-d`; that enables generating debug versions of the generated C++ code. In addition, the option `-X <tag>` may enable additional, more expensive debug instrumentation, as discussed below. Any use of `-X` implicitly turns on `-d`.

Debug Hooks

The simplest way to learn more about what's going on is to add hooks with `print` statements to your grammar. That's rather disruptive though, and hence there are also special `%debug` unit hooks which only get compiled into the resulting code if `spicy-driver` is run with debugging enabled (`-d`):

```
module Test;

public type test = unit {
    a: /1234/ %debug {
        print self.a;
    }

    b: /567890/;

    on b %debug { print self.b; }
};
```

```
# printf "1234567890" | spicy-driver -d debugging.spicy
1234
567890
```

```
# printf "1234567890" | spicy-driver debugging.spicy
```

Debug Streams

A second form of debugging support uses runtime *debug streams* that instrument the generated parsers to log activity as they are parsing their input. If you run `spicy-driver` with `-d`, you can set the environment variable `HILTI_DEBUG` to a set of debug stream names to select the desired information (see below for the list). Execution will then print debug information to standard error:

```
> echo "GET /index.html HTTP/1.0" | HILTI_DEBUG=spicy spicy-driver -d http-request.
↪spicy
[spicy] Request::RequestLine
[spicy]   method = GET
[spicy]   anon_2 =
[spicy]   uri = /index.html
[spicy]   anon_3 =
[spicy]   Request::Version
[spicy]     anon = HTTP/
[spicy]     number = 1.0
[spicy]   version = [$anon=b"HTTP/", $number=b"1.0"]
[spicy]   anon_4 = \n
GET, /index.html, 1.0
```

The available debug streams include:

spicy Logs unit fields and variables as they receive values. This is often the most helpful output as it shows rather concisely what the parser is doing, and in particular how far it gets in cases where it doesn't parse something correctly.

spicy-verbose Logs various internals about the parsing process, including the grammar rules currently being parsed, the current input, and lexer tokens.

This stream is primarily intended for debugging the Spicy compiler itself, but it can be helpful also in particular for understanding the data that remains to be parsed.

hilti-trace This is a HILTI-level debug stream that records every HILTI instruction being executed. To use this, you need to run `spicy-driver` with `-X trace`.

This stream is primarily intended for debugging the Spicy compiler itself.

hilti-flow This is a HILTI-level debug level recording flow information like function calls. To use this, you need to run `spicy-driver` with `-X flow`.

This stream is primarily intended for debugging the Spicy compiler itself, although it may also be helpful to understand the internal control flow when writing a grammar.

Multiple streams can be enabled by separating them with colons.

Exceptions

When encountering runtime errors, Spicy by default triggers C++ exceptions that bubble up back to the host application. If not handled there, execution will stop. For debugging, you can also let the Spicy runtime system `abort()` with a core dump, instead of throwing an exception, by running `spicy-driver` with `--abort-on-exceptions`. That especially helps inside a debugger.

If in addition you specify `--show-backtraces` as well, it will print a stack trace before aborting (assuming support for that is available on your platform).

Inspecting Generated Code

Using `spicyc` you can inspect the code that's being generated for a given Spicy grammar:

- `spicyc -p` outputs the intermediary HILTI code. The code tends to be pretty intuitively readable. Even if you don't know all the specifics of HILTI, much of the code is rather close to Spicy itself. (Per *above*, you can trace the generated HILTI code as it executes by activating the `hilti-trace` debug stream).
- `spicyc -c` outputs the final C++ code. If you add `-L`, the output will also include additional code generated by HILTI's linker (which enables cross-module functionality).
- When JITing a grammar with `spicyc -j`, running with `-D dump-code` will record all generated intermediary code (HILTI code, C++ code, object files) into files `dbg.*` inside the current directory.

Skipping validation

When working on the Spicy code, it can be helpful to disable internal validation of generated HILTI code with `-V`. That way, one can often still see the HILTI code even if it's malformed. Note, however, that Spicy may end up crashing if broken HILTI code gets passed into later stages.

2.5.6 Custom Extensions

As shown in the *library section*, Spicy's runtime library comes with a set of built-in functions that grammars can leverage for more complex tasks, such as Base64 decoding and `zlib` decompression. Behind the scenes, these functions are just small Spicy wrappers that call into corresponding C++ code. This section describes how you can provide your own built-in functions leveraging custom external C++ code.

Basic Example

Let's start with a simple example. Say we want to make a `rot13` function available to Spicy that rotates each letter of a string by 13 characters, using the following C++ implementation:

Listing 5: mylibrary.cc

```
#include <string>

namespace MyLibrary {

// Rotate each letter by 13 characters.
std::string rot13(const std::string& in) {
    std::string out;

    for ( auto c : in ) {
        char b = islower(c) ? 'a' : 'A';
        auto d = c - b + 13;

        if ( d >= 13 && d <= 38 )
            c = static_cast<char>(d % 26 + b);

        out.push_back(c);
    }

    return out;
}
```

(continues on next page)

(continued from previous page)

```
} // namespace MyLibrary
```

We can make this function available to Spicy by declaring it inside a custom module like this:

Listing 6: mylibrary.spicy

```
module MyLibrary;

## Rotate each letter by 13 characters.
public function rot13(s: string) : string &cxxname="MyLibrary::rot13";
```

Now we can use it in our Spicy code:

```
module Example;

import MyLibrary;

const data = "Hello, world!";
global encrypted = MyLibrary::rot13(data);
global decrypted = MyLibrary::rot13(encrypted);

print "'%s' -> '%s' -> '%s'" % (data, encrypted, decrypted);
```

To compile and execute this, we give `spicyc` all three files:

```
# spicyc -j rot13.spicy mylibrary.spicy mylibrary.cc
'Hello, world!' -> 'Uryyb, jbeyq!' -> 'Hello, world!'
```

Let's look more closely at what's going on here.

In `mylibrary.spicy`, the function attribute `&cxxname` is the marker for Spicy that we're not declaring a standard Spicy function but an external function that's implemented in C++. The value of that attribute is the C++-side name of the function, which Spicy will use to call it. In our case, the C++ name is the same as the fully qualified Spicy-side name, because we aligned C++ namespace and function ID accordingly. However, that doesn't need to be the case; see below for more.

Besides the naming, the key to interfacing Spicy with C++ lies in aligning the types for function parameters and results between the two sides. Internally, Spicy automatically creates a C++ function prototype for any function declaration coming with a `&cxxname` attribute. To do so, Spicy maps its own types to corresponding C++ types. We can see how that looks in our example by running `spicyc -gP` to print out the generated function prototype (plus a bit of boilerplate to produce a complete C++ `#include` header):

```
# spicyc -gP mylibrary.spicy
// Prototypes for module MyLibrary

#ifndef HILTI_PROTOTYPES_MYLIBRARY_H

#include <hilti/rt/libhilti.h>
#include <spicy/rt/libspicy.h>

namespace MyLibrary {
    extern auto rot13(const std::string& s) -> std::string;
}

namespace __hlt::MyLibrary {
```

(continues on next page)

(continued from previous page)

```
    extern void __register_module();
}

#endif
```

As you can see, Spicy maps `rot13`'s string argument and result into `std::string`, which happens to be exactly what we need in our simple example.

Todo: We should tweak `-P` so that it disables optimization automatically (because that just removes the supposedly unused function). (#%s1284)

Advanced Usage

In practice, it's often not quite as simple to provide new built-in functions as in our example because parameters or results might not directly align between Spicy and C++. In the following we walk through aspects that typically come up here, in particular when interfacing to already existing C++ code that doesn't know anything about Spicy.

Function naming

As said above, the name of the C++ function must be provided to Spicy through the `&cxxname` attribute. The name may be namespaced, but can also just be a global identifier, depending on what the C++ code expects. Spicy will simply use the name literally in any C++ code it generates.

Type mapping

For each Spicy type, the compiler picks a corresponding C++ type when generating an internal function prototype. The following table shows those mappings:

Spicy Type	C++ type
addr	hilti::rt::Address
bitfield(N)	uintN_t
bool	hilti::rt::Bool
bytes	hilti::rt::Bytes
enum	enum (a corresponding C++ enum type will be generated)
exception	hilti::rt::Exception
(u)int8/16/32/64	(u)int_8/16/32/64_t
interval	hilti::rt::Interval
list<T>	hilti::rt::Vector<T>
map<K,V>	hilti::rt::Map<K,V>
optional<T>	std::optional<T>
port	hilti::rt::Port
real	double
regexp	hilti::rt::RegExp
set<T>	hilti::rt::Set<T>
sink	spicy::rt::Sink
stream	spicy::rt::Stream
string	std::string
struct	struct (a corresponding C++ struct type will be generated)
time	hilti::rt::Time
tuple<Ts>	std::tuple<Ts>
unit	struct (a corresponding C++ struct type will be generated)
vector<T>	hilti::rt::Vector<T>

The C++ types that reside inside the hilti::rt or spicy::rt scopes, are defined in hilti/runtime/libhilti.h and spicy/runtime/libspicy.h, respectively.

If these type mappings match what your C++ code expects—as it did in our example—then there's nothing else to do. If they don't, you have three options:

1. You adapt the function's C++ declaration accordingly, assuming you can modify it.
2. If you are lucky, you may be able to get away with "slightly mismatching" C++ types as long as (a) they coerce into each other, and (b) Spicy is able to see the original prototype so that it can skip generating its own. However, it may be tricky to satisfy these two conditions, see the box below for more.
3. You provide an additional C++ wrapper function receiving the expected types and forwarding them to the actual function as appropriate.

Using external prototypes

Per option (2) above, it is possible to get away with "slightly mismatching" types in some cases. For example, if your C++ function expects just an unsigned int for one of its arguments, but Spicy passes a value as an uint8_t, that will compile just fine. But there's a caveat: to make that work, you need to prevent Spicy from generating its own prototype for the function, as the two would mismatch. There's a &have_prototype attribute for that: If you add that to the Spicy-side function declaration, Spicy will assume that it can rely on an already existing C++ prototype instead of creating its own.

However, there's a second challenge here now: Spicy's generated C++ code needs to actually find that existing prototype somewhere—-but unfortunately, there's currently no way of explicitly providing it. The only case where you can make this work right now is when Spicy's C++ runtime library happens to be already including a C++ header that comes with your desired prototype. That's unlikely for any non-standard functionality, but it may work if you're wrapping a standard system function, such as anything from stdlib.h for example (e.g., random()).

Todo: We should add a mechanism to provide an arbitrary custom C++ prototype directly. (#%s1286)

There's a similar trick for complex types, such as structs and enums: If your C++ function requires a type that Spicy doesn't know anything about, you can declare a Spicy-side dummy substitute like this:

```
public type MyType = __library_type("MyType");
```

Then you can use `MyType` as as type in Spicy-side declarations. The name given to `__library_type` works similar to function names provided to `&cxxname`: the Spicy compiler will take them literally to refer to the C++ type. However, this will work only in similar situations as &have_prototype`: the compiler must be able to find an existing declaration for that C++ type in any of its standard includes files. It's fine for that declaration to be just a forward declaration if that's sufficient for the C++ code to compile).

Linking to Libraries

In our example, we gave the custom C++ code directly to the Spicy compiler. That code will then simply be compiled along with everything else and be linked into the resulting binary code. Often, however, you may instead want to make functions available to Spicy that are implemented inside an external C++ library. In that case, Spicy will need to link the binary code to that library. To support that, `spicyc` provides an option `--cxx-link` that takes the full path to a (static or shared) library to link in. For example:

```
# spicyc -j --cxx-link /usr/local/lib/libz.a mycode.spicy
```

`--cxx-link` can be specified multiple times.

Include paths

If your C++ code requires additional include files outside of standard include paths, you can set the environment variable `HILTI_CXX_INCLUDE_DIRS` to a colon-separated list of additional directories for `spicyc` to use when compiling C++ code.

2.6 Toolchain

2.6.1 `spicy-build`

`spicy-build` is a shell frontend that compiles Spicy source code into a standalone executable by running *spicyc* to generate the necessary C++ code, then spawning the system compiler to compile and link that.

```
spicy-build [options] <input files>

    -d          Build a debug version.
    -o <file>   Destination name for the compiled executable; default is "a.out".
    -t          Do not delete tmp files (useful for inspecting, and use with debugger)
    -v          Verbose output, display command lines executing.
    -S          Do not compile the "spicy-driver" host application into executable.

Input files may be anything that spicyc can compile to C++.
```

2.6.2 spicy-config

`spicy-config` reports information about Spicy's build & installation options.

```
Usage: spicy-config [options]

Available options:

    --bindir                Prints the path to the directory where binaries are␣
↪installed.
    --build                 Prints "debug" or "release", depending on the build␣
↪configuration.
    --cmake-path            Prints the path to Spicy-provided CMake modules
    --cxx                   Print the path to the C++ compiler used to build Spicy
    --cxxflags              Print flags for C++ compiler when compiling generated␣
↪code statically
    --cxxflags-hlto         Print flags for C++ compiler when building precompiled␣
↪HLTO libraries
    --debug                 Output flags for working with debugging versions.
    --distbase              Print path of the Spicy source distribution.
    --dynamic-loading       Adjust --ldflags for host applications that dynamically␣
↪load precompiled modules
    --have-toolchain        Prints 'yes' if the Spicy toolchain was built, 'no'␣
↪otherwise.
    --have-zeek             Prints 'yes' if the Spicy was compiled with Zeek support,
↪'no' otherwise.
    --help                  Print this usage summary
    --include-dirs          Prints the Spicy runtime's C++ include directories
    --ldflags               Print flags for linker when compiling generated code␣
↪statically
    --ldflags-hlto          Print flags for linker linker when building precompiled␣
↪HLTO libraries
    --libdirs               Print standard Spicy library directories.
    --prefix                Print path of installation
    --spicy-build           Print the path to the spicy-build script.
    --spicyc                Print the path to the spicyc binary.
    --version               Print the Spicy version as a string.
    --version-number        Print the Spicy version as a numerical value.
    --zeek                  Print the path to the Zeek executable
    --zeek-include-dirs     Print the Spicy runtime's C++ include directories
    --zeek-module-path      Print the path of the directory the Zeek plugin searches␣
↪for *.hlto modules
    --zeek-plugin-path      Print the path to go into ZEEK_PLUGIN_PATH for enabling␣
↪the Zeek Spicy plugin
    --zeek-prefix           Print the path to the Zeek installation prefix
    --zeek-version          Print the Zeek version (empty if no Zeek available)
    --zeek-version-number   Print the Zeek version as a numerical value (zero if no␣
↪Zeek available)
```

2.6.3 spicyc

spicyc compiles Spicy code into C++ output, optionally also executing it directly through JIT.

```
Usage: spicyc [options] <inputs>

Options controlling code generation:

  -c | --output-c++                Print out all generated C++ code (including linker␣
↪glue by default).
  -d | --debug                     Include debug instrumentation into generated code.
  -e | --output-all-dependencies  Output list of dependencies for all compiled␣
↪modules.
  -g | --disable-optimizations    Disable HILTI-side optimizations of the generated␣
↪code.
  -j | --jit-code                  Fully compile all code, and then execute it unless -
↪-output-to gives a file to store it
  -l | --output-linker             Print out only generated HILTI linker glue code.
  -o | --output-to <path>          Path for saving output.
  -p | --output-hilti              Just output parsed HILTI code again.
  -v | --version                   Print version information.
  -A | --abort-on-exceptions       When executing compiled code, abort() instead of␣
↪throwing HILTI exceptions.
  -B | --show-backtraces           Include backtraces when reporting unhandled␣
↪exceptions.
  -C | --dump-code                 Dump all generated code to disk for debugging.
  -D | --compiler-debug <streams> Activate compile-time debugging output for given␣
↪debug streams (comma-separated; 'help' for list).
  -E | --output-code-dependencies Output list of dependencies for all compiled␣
↪modules that require separate compilation of their own.
  -L | --library-path <path>      Add path to list of directories to search when␣
↪importing modules.
  -O | --optimize                  Build optimized release version of generated code.
  -P | --output-prototypes         Output C++ header with prototypes for public␣
↪functionality.
  -R | --report-times              Report a break-down of compiler's execution time.
  -S | --skip-dependencies         Do not automatically compile dependencies during␣
↪JIT.
  -T | --keep-tmps                 Do not delete any temporary files created.
  -V | --skip-validation           Don't validate ASTs (for debugging only).
  -X | --debug-addl <addl>         Implies -d and adds selected additional␣
↪instrumentation (comma-separated; see 'help' for list).
       --cxx-link <lib>            Link specified static archive or shared library␣
↪during JIT or to produced HLTO file. Can be given multiple times.

  -Q | --include-offsets           Include stream offsets of parsed data in output.


Inputs can be .spicy, .hlt, .cc/.cxx, *.hlto.
```

spicyc also supports the following environment variables to control the compilation process:

SPICY_PATH Replaces the built-in search path for **.spicy* source files.

SPICY_CACHE Location for storing precompiled C++ headers. Default is ~/.cache/spicy/<VERSION>.

HILTI_CXX Specifies the path to the C++ compiler to use.

HILTI_CXX_COMPILER_LAUNCHER Specifies a command to prefix compiler invocations with during JIT. This can e.g., be used to use a compiler cache like ccache. If Spicy was configured with e.g., `--with-hilti-compiler-launcher=ccache` (the equivalent CMake option is `HILTI_COMPILER_LAUNCHER`) `ccache` would automatically be used during JIT. Setting this variable to an empty value disables use of `ccache` in that case.

HILTI_CXX_INCLUDE_DIRS Specified additional, colon-separated C++ include directory to search for header files.

HILTI_JIT_PARALLELISM Set to specify the maximum number of background compilation jobs to run during JIT. Defaults to number of cores.

HILTI_JIT_SEQUENTIAL Set to prevent spawning multiple concurrent C++ compiler instances. This overrides any value set for `HILTI_JIT_PARALLELISM` and effectively sets it to one.

HILTI_OPTIMIZER_PASSES Colon-separated list of optimizer passes to activate. If unset uses the default-enabled set.

HILTI_PATH Replaces the built-in search path for **.hlt* source files.

HILTI_PRINT_SETTINGS Set to see summary of compilation options.

2.6.4 `spicy-driver`

`spicy-driver` is a standalone Spicy host application that compiles and executes Spicy parsers on the fly, and then feeds them data for parsing from standard input.

```
Usage: cat <data> | spicy-driver [options] <inputs> ...

Options:

  -d | --debug                     Include debug instrumentation into generated code.
  -i | --increment <i>             Feed data incrementally in chunks of size n.
  -f | --file <path>               Read input from <path> instead of stdin.
  -l | --list-parsers              List available parsers and exit.
  -p | --parser <name>             Use parser <name> to process input. Only neeeded if␣
↪more than one parser is available.
  -v | --version                   Print version information.
  -A | --abort-on-exceptions       When executing compiled code, abort() instead of␣
↪throwing HILTI exceptions.
  -B | --show-backtraces           Include backtraces when reporting unhandled␣
↪exceptions.
  -D | --compiler-debug <streams> Activate compile-time debugging output for given␣
↪debug streams (comma-separated; 'help' for list).
  -F | --batch-file <path>         Read Spicy batch input from <path>; see docs for␣
↪description of format.
  -L | --library-path <path>       Add path to list of directories to search when␣
↪importing modules.
  -O | --optimize                  Build optimized release version of generated code.
  -R | --report-times              Report a break-down of compiler's execution time.
  -S | --skip-dependencies         Do not automatically compile dependencies during␣
↪JIT.
  -U | --report-resource-usage     Print summary of runtime resource usage.
  -X | --debug-addl <addl>         Implies -d and adds selected additional␣
↪instrumentation (comma-separated; see 'help' for list).

Environment variables:

  SPICY_PATH                       Colon-separated list of directories to search for␣
↪modules. In contrast to --library-paths using this flag overwrites builtin paths.
```

(continues on next page)

(continued from previous page)

```
Inputs can be .hlt, .spicy, .cc/.cxx, *.o, *.hlto.
```

spicy-driver supports the same environment variables as *spicyc*.

Specifying the parser to use

If there's only single public unit in the Spicy source code, spicy-driver will automatically use that for parsing its input. If there's more than one public unit, you need to tell spicy-driver which one to use through its --parser (or -p) option. To see the parsers that are available, use --list-parsers (or -l).

In addition to the names shown by --list-parsers, you can also specify a parser through a port or MIME type if the corresponding unit *defines them through properties*. For example, if a unit defines %port = 80/tcp, you can use spicy-driver -p 80/tcp to select it. To specify a direction, add either %orig or %resp (e.g., -p 80/tcp%resp); then only units with a port tagged with an &originator or &responder attribute, respectively, will be considered. If a unit defines %mime-type = application/test, you can select it through spicy-driver -p application/test. (Note that there must be exactly one unit with a matching property for this all to work, otherwise you'll get an error message.)

Batch input

spicy-driver provides a batch input mode for processing multiple interleaved input flows in parallel, mimicking how host applications like Zeek would be employing Spicy parsers for processing many sessions concurrently. The batch input must be prepared in a specific format (see below) that provides embedded meta information about the contained flows of input. The easiest way to generate such a batch is a Zeek script coming with Spicy. If you run Zeek with this script on a PCAP trace, it will record the contained TCP and UDP sessions into a Spicy batch file:

```
# zeek -b -r http/methods.trace record-spicy-batch.zeek
tracking [orig_h=128.2.6.136, orig_p=46562/tcp, resp_h=173.194.75.103, resp_p=80/tcp]
tracking [orig_h=128.2.6.136, orig_p=46563/tcp, resp_h=173.194.75.103, resp_p=80/tcp]
tracking [orig_h=128.2.6.136, orig_p=46564/tcp, resp_h=173.194.75.103, resp_p=80/tcp]
tracking [orig_h=128.2.6.136, orig_p=46565/tcp, resp_h=173.194.75.103, resp_p=80/tcp]
tracking [orig_h=128.2.6.136, orig_p=46566/tcp, resp_h=173.194.75.103, resp_p=80/tcp]
tracking [orig_h=128.2.6.136, orig_p=46567/tcp, resp_h=173.194.75.103, resp_p=80/tcp]
[...]
tracking [orig_h=128.2.6.136, orig_p=46608/tcp, resp_h=173.194.75.103, resp_p=80/tcp]
tracking [orig_h=128.2.6.136, orig_p=46609/tcp, resp_h=173.194.75.103, resp_p=80/tcp]
tracking [orig_h=128.2.6.136, orig_p=46610/tcp, resp_h=173.194.75.103, resp_p=80/tcp]
recorded 49 sessions total
output in batch.dat
```

You will now have a file batch.dat that you can use with spicy-driver -F batch.data ....

The batch created by the Zeek script will select parsers for the contained sessions through well-known ports. That means your units need to have a %port property matching the responder port of the sessions you want them to parse. So for the HTTP trace above, our Spicy source code would need to provide a public unit with property %port = 80/tcp;.

In case you want to create batches yourself, we document the batch format in the following. A batch needs to start with a line !spicy-batch v2<NL>, followed by lines with commands of the form @<tag> <arguments><NL>.

There are two types of input that the batch format can represent: (1) individual, uni-directional flows; and (2) bi-directional connections consisting in turn of one flow per side. The type is determined through an initial command:

`@begin-flow` starts a flow flow, and `@begin-conn` starts a connection. Either form introduces a unique, free-form ID that subsequent commands will then refer to. The following commands are supported:

`@begin-flow FID TYPE PARSER<NL>` Initializes a new input flow for parsing, associating the unique ID `FID` with it. `TYPE` must be either `stream` for stream-based parsing (think: TCP), or `block` for parsing each data block independent of others (think: UDP). `PARSER` is the name of the Spicy parser to use for parsing this input flow, given in the same form as with `spicy-driver`'s `--parser` option (i.e., either as a unit name, a `%port`, or a `%mime-type`).

`@begin-conn CID TYPE ORIG_FID ORIG_PARSER RESP_FID RESP_PARSER<NL>` Initializes a new input connection for parsing, associating the unique connection ID `CID` with it. `TYPE` must be either `stream` for stream-based parsing (think: TCP), or `block` for parsing each data block independent of others (think: UDP). `ORIG_FID` is separate unique ID for the originator-side flow, and `ORIG_PARSER` is the name of the Spicy parser to use for parsing that flow. `RESP_FID` and `RESP_PARSER` work accordingly for the responder-side flow. The parsers can be given in the same form as with `spicy-driver`'s `--parser` option (i.e., either as a unit name, a `%port`, or a `%mime-type`).

`@data FID SIZE<NL>` A block of data for the input flow `FID`. This command must be followed directly by binary data of length `SIZE`, plus a final newline character. The data represents the next chunk of input for the corresponding flow. `@data` can be used only inside corresponding `@begin-*` and `@end-*` commands bracketing the flow ID.

`@end-flow FID<NL>` Finalizes parsing of the input flow associated with `FID`, releasing all state. This must come only after a corresponding `@begin-flow` command, and every `@begin-flow` must eventually be followed by an `@end-flow`.

`@end-conn CID<NL>` Finalizes parsing the input connection associated with `CID`, releasing all state (including for its two flows). This must come only after a corresponding `@begin-conn` command, and every `@begin-conn` must eventually be followed by an `@end-end`.

2.6.5 `spicy-dump`

`spicy-dump` is a standalone Spicy host application that compiles and executes Spicy parsers on the fly, feeds them data for processing, and then at the end prints out the parsed information in either a readable, custom ASCII format, or as JSON (`--json` or `-J`). By default, `spicy-dump` disables showing the output of Spicy `print` statements, `--enable-print` or `-P` reenables that.

```
Usage: cat <data> | spicy-dump [options] <inputs> ...

Options:

  -d | --debug                    Include debug instrumentation into generated code.
  -f | --file <path>              Read input from <path> instead of stdin.
  -l | --list-parsers             List available parsers and exit.
  -p | --parser <name>            Use parser <name> to process input. Only neeeded if␣
↪more than one parser is available.
  -v | --version                  Print version information.
  -A | --abort-on-exceptions      When executing compiled code, abort() instead of␣
↪throwing HILTI exceptions.
  -B | --show-backtraces          Include backtraces when reporting unhandled␣
↪exceptions.
  -D | --compiler-debug <streams> Activate compile-time debugging output for given␣
↪debug streams (comma-separated; 'help' for list).
  -L | --library-path <path>      Add path to list of directories to search when␣
↪importing modules.
  -J | --json                     Print JSON output.
  -O | --optimize                 Build optimized release version of generated code.
```

(continues on next page)

(continued from previous page)

```
  -P | --enable-print              Show output of Spicy 'print' statements (default:␣
↪off).
  -Q | --include-offsets           Include stream offsets of parsed data in output.
  -R | --report-times              Report a break-down of compiler's execution time.
  -S | --skip-dependencies         Do not automatically compile dependencies during␣
↪JIT.
  -X | --debug-addl <addl>         Implies -d and adds selected additional␣
↪instrumentation (comma-separated; see 'help' for list).

Environment variables:

  SPICY_PATH                       Colon-separated list of directories to search for␣
↪modules. In contrast to --library-paths using this flag overwrites builtin paths.

Inputs can be .hlt, .spicy, *.spicy *.hlt *.hlto.
```

2.7 Zeek Integration

While Spicy itself remains application independent, transparent integration into Zeek has been a primary goal for its development. To facilitate adding new protocol and file analyzers to Zeek, there is a Zeek plugin that makes Spicy parsers accessible to Zeek's processing pipeline. In the following, we dig deeper into how to use all of this.

Note: There are a number of pieces involved in creating a full Zeek analyzer, in particular if you want to distribute it as a Zeek package. To help you get started with that, Zeek's package manager can create a skeleton Spicy package by running:

```
# zkg create --features=spicy-protocol-analyzer --packagedir <packagedir>
```

The generated files mark places that will need manual editing with `TODO`. See the *tutorial* for more on this.

2.7.1 Terminology

In Zeek, the term "analyzer" refers generally to a component that processes a particular protocol ("protocol analyzer"), file format ("file analyzer"), or low-level packet structure ("packet analyzer"). "Processing" here means more than just parsing content: An analyzer controls when it wants to be used (e.g., with connections on specific ports, or with files of a specific MIME type); what events to generate for Zeek's scripting layer; and how to handle any errors occurring during parsing. While Spicy itself focuses just on the parsing part, the Spicy plugin makes it possible to provide the remaining pieces to Zeek, turning a Spicy parser into a full Zeek analyzer. That's what we refer to as a "Spicy (protocol/file/packet) analyzer" for Zeek.

2.7.2 Installation

Since Zeek version 5.0, Spicy and Spicy plugin are by default bundled with Zeek. To confirm whether Spicy plugin is available you can inspect the list of installed plugin:

```
# zeek -N Zeek::Spicy
Zeek::Spicy - Support for Spicy parsers (*.hlto) (built-in)
```

If you see `Zeek::Spicy` listed, spicy-plugin is available, otherwise it needs to be installed manually. The recommended way to do so is through Zeek's package manager zkg. If you have not yet installed *zkg*, follow its instructions.

You will need to have Spicy and Zeek installed as well of course. Before proceeding, make sure `spicy-config` and `zeek-config` are in your `PATH`:

```
# which spicy-config
/opt/spicy/bin/spicy-config

# which zeek-config
/usr/local/zeek/bin/zeek-config
```

Package Installation

The easiest way to install the plugin is through Zeek's package manager:

```
# zkg install zeek/spicy-plugin
```

This will pull down the plugin's package, compile and test the plugin, and then install and activate it. That process may take a bit to complete. To check afterwards that the plugin has become available, run `zeek -N Zeek::Spicy`, it should show output like this:

```
# zeek -N Zeek::Spicy
Zeek::Spicy - Support for Spicy parsers (*.hlto) (dynamic, version x.y.z)
```

By default, *zkg* will install the most recent release version of the plugin. If you want to install the current development version, use `zkg install --version main zeek/spicy-plugin` instead.

If you want to develop your own Spicy analyzers for Zeek, you will need a tool that comes with the plugin's installation: `spicyz`. If you are using a recent version of *zkg* (>= 2.8.0), it's easy to make the tool show up in your `PATH`: Either run `zkg env` (see this configuration advice) or update your `PATH` manually:

```
# export PATH=$(zkg config bin_dir):$PATH
```

If you are using an older version of *zkg* (including the version coming with Zeek 4.0), it's a bit more difficult to find `spicyz`: it will be inside your *zkg* state directory at `<state_dir>/clones/package/spicy-plugin/build/bin/spicyz`. We recommend adding that directory to your `PATH`. (The state directory is usually either `<zeek-prefix>/var/lib/zkg` or `~/.zkg`, depending on how you have set up *zkg*.)

Manual Installation

If you prefer, you can also compile the Zeek plugin yourself, outside of the package manager by cloning the plugin's GitHub repository and building it through CMake. See the instructions in its README. This will install `spicyz` into `<prefix>/bin`.

Note: Developer's note: It works to point `ZEEK_PLUGIN_PATH` directly to the plugin's build directory, without installing it first. If you are building the plugin as part of the Spicy distribution, it will land in `<build-directory>/zeek/spicy-plugin`.

2.7.3 Interface Definitions ("evt files")

Per above, a Spicy analyzer for Zeek does more than just parsing data. Accordingly, we need to tell the Zeek plugin a couple of additional pieces about analyzers we want it to provide to Zeek:

Analyzer setup The plugin needs to know what type of analyzers we are creating, when we want Zeek to activate them, and what Spicy unit types to use as their parsing entry point.

Event definitions We need to tell the Spicy plugin what Zeek events to provide and when to trigger them.

We define all of these through custom interface definition files that Spicy's compiler for Zeek reads in. These files use an `*.evt` extension, and the following subsections discuss their content in more detail.

Generally, empty lines and comments starting with `#` are ignored in an `*.evt`.

Note: The syntax for `*.evt` files comes with some legacy pieces that aren't particularly pretty. We may clean that up at some point.

Analyzer Setup

You can define protocol analyzers, packet analyzers and file analyzers in an `*.evt` file, per the following.

Protocol Analyzer

To define a protocol analyzer, add a new section to an `*.evt` file that looks like this:

```
protocol analyzer ANALYZER_NAME over TRANSPORT_PROTOCOL:
    PROPERTY_1,
    PROPERTY_2,
    ...
    PROPERTY_N;
```

Here, `ANALYZER_NAME` is a name to identify your analyzer inside Zeek. You can choose names arbitrarily as long as they are unique. As a convention, however, we recommend name with a `spicy::*` prefix (e.g., `spicy::BitTorrent`).

On the Zeek-side, through some normalization, these names automatically turn into tags added to Zeek's `Analyzer::Tag` enum. For example, `spicy::BitTorrent` turns into `Analyzer::ANALYZER_SPICY_BITTORRENT`.

The analyzer's name is also what goes into Zeek signatures to activate an analyzer DPD-style. If the name is `spicy::BitTorrent`, you'd write `enable "spicy::BitTorrent"` into the signature.

Note: Once you have made your analyzers available to Zeek (which we will discuss below), running `zeek -NN Zeek::Spicy` will show you a summary of what's now available, including their Zeek-side names and tags.

`TRANSPORT_PROTOCOL` can be either `tcp` or `udp`, depending on the transport-layer protocol that your new analyzer wants to sit on top of.

Following that initial `protocol analyzer ...` line, a set of properties defines further specifics of your analyzer. The following properties are supported:

`parse [originator|responder] with SPICY_UNIT` Specifies the top-level Spicy unit(s) the analyzer uses for parsing payload, with `SPICY_UNIT` being a fully-qualified Spicy-side type name (e.g. `HTTP::Request`). The unit type must have been declared as `public` in Spicy.

If `originator` is given, the unit is used only for parsing the connection's originator-side payload; and if `responder` is given, only for responder-side payload. If neither is given, it's used for both sides. In other words, you can use different units per side by specifying two properties `parse originator with ...` and `parse responder with ...`.

`port PORT` or `ports { PORT_1, ..., PORT_M }` Specifies one or more well-known ports for which you want Zeek to automatically activate your analyzer with corresponding connections. Each port must be specified in Spicy's *syntax for port constants* (e.g., `80/tcp`), or as a port range `PORT_START-PORT_END` where start and end port are port constants forming a closed interval. The ports' transport protocol must match that of the analyzer.

Note: The plugin will also honor any `%port` *meta data property* that the responder-side `SPICY_UNIT` may define (as long as the attribute's direction is not `originator`).

`replaces ANALYZER_NAME` Disables an existing analyzer that Zeek already provides internally, allowing you to replace a built-in analyzer with a new Spicy version. `ANALYZER_NAME` is the Zeek-side name of the analyzer. To find that name, inspect the output of `zeek -NN` for available analyzers:

```
# zeek -NN | grep '\[Analyzer\]'
...
[Analyzer] SMTP (ANALYZER_SMTP, enabled)
...
```

Here, `SMTP` is the name you would write into `replaces` to disable the built-in SMTP analyzer.

As a full example, here's what a new HTTP analyzer could look like:

```
protocol analyzer spicy::HTTP over TCP:
    parse originator with HTTP::Requests,
    parse responder with HTTP::Replies,
    port 80/tcp,
    replaces HTTP;
```

Packet Analyzer

Defining packet analyzers works quite similar to protocol analyzers through `*.evt` sections like this:

```
packet analyzer ANALYZER_NAME:
    PROPERTY_1,
    PROPERTY_2,
    ...
    PROPERTY_N;
```

Here, `ANALYZER_NAME` is again a name to identify your analyzer inside Zeek. On the Zeek-side, the name will be added to Zeek's `PacketAnalyzer::Tag` enum.

Packet analyzers support the following properties:

`parse with SPICY_UNIT` Specifies the top-level Spicy unit the analyzer uses for parsing each packet, with `SPICY_UNIT` being a fully-qualified Spicy-side type name. The unit type must have been declared as `public` in Spicy.

`replaces ANALYZER_NAME` Disables an existing packet analyzer that Zeek already provides internally, allowing you to replace a built-in analyzer with a new Spicy version. `ANALYZER_NAME` is the Zeek-side name of the existing analyzer. To find that name, inspect the output of `zeek -NN` for available analyzers:

```
# zeek -NN | grep '\[Packet Analyzer\]'
...
[Packet Analyzer] Ethernet (ANALYZER_ETHERNET)
...
```

Here, `Ethernet` is the name you would give to `replaces` to disable the built-in Ethernet analyzer.

When replacing an existing packet analyzer, you still need to configure your new analyzer on the Zeek side through a *zeek_init()* event handler. See below for more.

Note: This feature requires Zeek >= 5.2.

As a full example, here's what a new packet analyzer could look like:

```
packet analyzer spicy::RawLayer:
    parse with RawLayer::Packet;
```

In addition to the Spicy-side configuration, packet analyzers also need to be registered with Zeek inside a `zeek_init` event handler; see the Zeek documentation for more. The `-NN` output shows your new analyzer's `ANALYZER_*` tag to use.

Note: Depending on relative loading order between your analyzer and Zeek's scripts, in some situations you may not have access to the `ANALYZER_*` tag referring to your new analyzer. If you run into this, use PacketAnalyzer::try_register_packet_analyzer_by_name to register your Spicy analyzer by name (as shown by `-NN`). Example:

```
event zeek_init()
    {
    if ( ! PacketAnalyzer::try_register_packet_analyzer_by_name("Ethernet", 0x88b5,
↪"spicy::RawLayer") )
            Reporter::error("unknown analyzer name used");
    }
```

File Analyzer

Defining file analyzers works quite similar to protocol analyzers, through `*.evt` sections like this:

```
file analyzer ANALYZER_NAME:
    PROPERTY_1,
    PROPERTY_2,
    ...
    PROPERTY_N;
```

Here, `ANALYZER_NAME` is again a name to identify your analyzer inside Zeek. On the Zeek-side, the name will be added to Zeek's `Files::Tag` enum.

File analyzers support the following properties:

`parse with SPICY_UNIT` Specifies the top-level Spicy unit the analyzer uses for parsing file content, with `SPICY_UNIT` being a fully-qualified Spicy-side type name. The unit type must have been declared as `public` in Spicy.

`mime-type MIME-TYPE` Specifies a MIME type for which you want Zeek to automatically activate your analyzer when it sees a corresponding file on the network. The type is specified in standard `type/subtype` notion, without quotes (e.g., `image/gif`).

> **Note:** The plugin will also honor any `%mime-type` *meta data property* that the `SPICY_UNIT` may define.

> **Note:** Keep in mind that Zeek identifies MIME types through "content sniffing" (i.e., similar to libmagic), and usually not by protocol-level headers (e.g., *not* through HTTP's `Content-Type` header). If in doubt, examine `files.log` for what it records as a file's type.

`replaces ANALYZER_NAME` Disables an existing file analyzer that Zeek already provides internally, allowing you to replace a built-in analyzer with a new Spicy version. `ANALYZER_NAME` is the Zeek-side name of the analyzer. To find that name, inspect the output of `zeek -NN` for available analyzers:

```
# zeek -NN | grep '\[File Analyzer\]'
...
[File Analyzer] PE (ANALYZER_PE, enabled)
...
```

Here, `PE` is the name you would write into `replaces` to disable the built-in PE analyzer.

> **Note:** This feature requires Zeek >= 4.1.

As a full example, here's what a new GIF analyzer could look like:

```
file analyzer spicy::GIF:
    parse with GIF::Image,
    mime-type image/gif;
```

Event Definitions

To define a Zeek event that you want the Spicy plugin to trigger, you add lines of the form:

```
on HOOK_ID -> event EVENT_NAME(ARG_1, ..., ARG_N);

on HOOK_ID if COND -> event EVENT_NAME(ARG_1, ..., ARG_N);
```

The Zeek plugin automatically derives from this everything it needs to register new events with Zeek, including a mapping of the arguments' Spicy types to corresponding Zeek types. More specifically, these are the pieces going into such an event definition:

`on HOOK_ID` A Spicy-side ID that defines when you want to trigger the event. This works just like an `on ...` *unit hook*, and you can indeed use anything here that Spicy supports for those as well (except *container hooks*). So, e.g., `on HTTP::Request::%done` triggers an event whenever a `HTTP::Request` unit has been fully parsed, and `on HTTP::Request::uri` leads to an event each time the `uri` field has been parsed. (In the former example you may skip the `%done`, actually: `on HTTP::Request` implicitly adds it.)

`EVENT_NAME` The Zeek-side name of the event you want to generate, preferably including a namespace (e.g., `http::request`).

`ARG_1, ..., ARG_N` Arguments to pass to the event, given as arbitrary Spicy expressions. Each expression will be evaluated within the context of the unit that the `on ...` triggers on, similar to code running inside the body of a corresponding *unit hook*. That means the expression has access to `self` for accessing the unit instance that's currently being parsed.

The Spicy type of the expression determines the Zeek-side type of the corresponding event argument. Most Spicy types translate over pretty naturally, the following summarizes the translation:

Table 1: Type Conversion from Spicy to Zeek

Spicy Type	Zeek Type	Notes
`addr`	`addr`	
`bool`	`bool`	
`enum { ... }`	`enum { ... }`	[1]
`int(8|16|32|64)`	`int`	
`interval`	`interval`	
`list<T>`	`vector of T`	
`map<V,K>`	`table[V] of K`	
`optional<T>`	`T`	[2]
`port`	`port`	
`real`	`double`	
`set<T>`	`set[T]`	
`string`	`string`	
`time`	`time`	
`tuple<T_1, ... ,T_N>`	`record { T1, ..., T_N }`	[3]
`uint(8|16|32|64)`	`count`	
`vector<T>`	`vector of T`	

Note:

[1] A corresponding Zeek-side `enum` type is automatically created. See *below* for more.

[2] The optional value must have a value, otherwise a runtime exception will be thrown.

[3] Must be mapped to a Zeek-side record type with matching fields.

If a tuple element is mapped to a record field with a `&default` or `&optional` attribute, a couple special cases are supported:

- If the expression evaluates to `Null`, the record field is left unset.
- If the element's expression uses the `.?` operator and that fails to produce a value, the record field is likewise left unset.

In addition to full Spicy expressions, there are three reserved IDs with specific meanings when used as arguments:

`$conn` Refers to the connection that's currently being processed by Zeek. On the Zeek-side this will turn into a parameter of Zeek type `connection`. This ID can be used only with protocol analyzers.

`$file` Refers to the file that's currently being processed by Zeek. On the Zeek-side this will turn into a parameter of Zeek type `fa_file`. This ID can be used only with file analyzers.

`$packet` Refers to the packet that's currently being processed by Zeek. On the Zeek-side this will turn into a parameter of Zeek type `raw_pkt_hdr`, with any fields filled in that have already been parsed by Zeek's built-in analyzers. This ID can be used only with packet analyzers. (Note that instantiation of `raw_pkt_hdr` values can be relatively expensive on the Zeek side, so best to limit usage of this ID to a small part of the overall traffic.)

`$is_orig` A boolean indicating if the data currently being processed is coming from the originator (`True`) or responder (`False`) of the underlying connection. This turns into a corresponding boolean value on the Zeek side. This ID can be used only with protocol analyzers.

Note: Some tips:

- If you want to force a specific type on the Zeek-side, you have a couple of options:
  1. Spicy may provide a `cast` operator from the actual type into the desired type (e.g., `cast<uint64>(..)`).
  2. Argument expressions have access to global functions defined in the Spicy source files, so you can write a conversion function taking an argument with its original type and returning it with the desired type.
- List comprehension can be convenient to fill Zeek vectors: `[some_func(i) for i in self.my_list]`.

`if COND` If given, events are only generated if the expression `COND` evaluates to true. Just like event arguments, the expression is evaluated in the context of the current unit instance and has access to `self`.

Exporting Types

As we *discuss above*, the type of each event argument maps over to a corresponding Zeek type. On the Zeek side, that corresponding type needs to be known by Zeek. That is always the case for built-in atomic types (per the table:ref:*the conversion table <zeek-event-arg-types>*), but it can turn out more challenging to achieve for custom types, such as `enum` and `record`, for which you would normally need to create matching type declarations in your Zeek scripts. While that's not necessarily hard, but it can become quite cumbersome.

Fortunately, the Spicy plugin can help: for most types, it can instantiate corresponding Zeek types automatically as it loads the corresponding analyzer. While you will never actually see the Zeek-side type declarations, they will be available inside your Zeek scripts as if you had typed them out yourself–similar to other types that are built into Zeek itself.

To have the plugin create a Zeek type for your analyzer automatically, you need to `export` the Spicy type in your EVT file. The syntax for that is:

```
export SPICY_ID [as ZEEK_ID];
```

Here, `SPICY_ID` is the fully-scoped type ID on the Spicy side, and `ZEEK_ID` is the fully-scoped type ID you want in Zeek. If you leave out the `as ...` part, the Zeek name will be the same as the Spicy name, including its namespace. For example, say you have a Spicy unit `TFTP::Request`. Adding `export TFTP::Request;` to your EVT file will make a `record` type of the same name, and with the same fields, available in Zeek. If you instead use `export TFTP::Request as TheOtherTFTP::Request`, it will be placed into a different namespace instead.

Exporting types generally works for most Spicy types as long as there's an ID associated with them in your Spicy code. However, exporting is most helpful with user-defined types, such as `enum` and `unit`, because it can save you quite a bit of typing there. We discuss the more common type conversions in more detail below.

To confirm the types made available to Zeek, you can see all exports in the output of `zeek -NN`. With our 2nd `TFTP::Request` example, that looks like this:

```
[...]
# zeek -NN Zeek::Spicy
Zeek::Spicy - Support for Spicy parsers (*.hlto)
    [Type] TheOtherTFTP::Request
[...]
```

Note: Most, but not all, types can be exported automatically. For example, self-recursive types are currently not supported. Generally, if you run into trouble exporting a type, you can always fall back to declaring a corresponding Zeek version yourself in your Zeek script. Consider the `export` mechanism as a convenience feature that helps avoid writing a lot of boiler plate code in common cases.

Enum Types

When you export a Spicy `enum` type, the Spicy plugin creates a corresponding Zeek `enum` type. For example, assume the following Spicy declaration:

```
module Test;

type MyEnum = enum {
    A = 83,
    B = 84,
    C = 85
};
```

Using `export Test::MyEnum;`, the plugin will create the equivalent of the following Zeek type for use in your scripts:

```
module Test;

export {

  type MyEnum: enum {
      MyEnum_A = 83,
      MyEnum_B = 84,
      MyEnum_A = 85,
      MyEnum_Undef = -1
```

(continues on next page)

(continued from previous page)

```
  };

}
```

(The odd naming is due to ID limitations on the Zeek side.)

Note: For backward compatibility, the `enum` type comes with an additional property: all *public* enum types are automatically exported, even without adding any `export` to your EVT file. This feature may go away at some point, and we suggest to not rely on it on new code; always use `export` for *enum* types as well.

Unit Types

When you export a Spicy `unit` type, the Spicy plugin creates a corresponding Zeek `record` type. For example, assume the following Spicy declaration:

```
module Test;

type MyRecord = unit {
    a: bytes &size=4;
    b: uint16;

    var c: bool;
};
```

Using `export Test::MyRecord;`, the plugin will then create the equivalent of the following Zeek type for use in your scripts:

```
module Test;

export {

  type MyRecord: record {
    a: string &optional;
    b: count &optional;
    c: bool;
  };

}
```

The individual fields map over just like event arguments do, following the *the table* above. For aggregate types, this works recursively: if, e.g., a field is itself of unit type *and that type has been exported as well*, the plugin will map it over accordingly to the corresponding `record` type. Note that such dependent types must be exported *first* in the EVT file for this to work. As a result, you cannot export self-recursive unit types.

As you can see in the example, unit fields are always declared as optional on the Zeek-side, as they may not have been set during parsing. Unit variables are non-optional by default, unless declared as `&optional` in Spicy.

Struct Types

A Spicy `struct` type maps over to Zeek in the same way as `unit` types do, treating each field like a unit variable. See *there* for more information.

Tuple Types

A Spicy `tuple` type maps over to a Zeek `record` similar to how `unit` types do, treating each tuple element like a unit variable. See *there* for more information.

Exporting works only for `tuple` types that declare names for all their elements.

Importing Spicy Modules

Code in an `*.evt` file may need access to additional Spicy modules, such as when expressions for event parameters call Spicy functions defined elsewhere. To make a Spicy module available, you can insert `import` statements into the `*.evt` file that work *just like in Spicy code*:

`import NAME` Imports Spicy module `NAME`.

`import NAME from X.Y.Z;` Searches for the module `NAME` (i.e., for the filename `NAME.spicy`) inside a sub-directory `X/Y/Z` along the search path, and then imports it.

Conditional Compilation

`*.evt` files offer the same basic form of *conditional compilation* through `@if`/`@else`/`@endif` blocks as Spicy scripts. The Zeek plugin makes two additional identifiers available for testing to both `*.evt` and `*.spicy` code:

`HAVE_ZEEK` Always set to 1 by the plugin. This can be used for feature testing from Spicy code to check if it's being compiled for Zeek.

`ZEEK_VERSION` The numerical Zeek version that's being compiled for (see `zeek -e 'print Version::number'`).

This is an example bracketing code by Zeek version in an EVT file:

```
@if ZEEK_VERSION < 30200
    <EVT code for Zeek versions older than 3.2>
@else
    <EVT code for Zeek version 3.2 or newer>
@endif
```

2.7.4 Compiling Analyzers

Once you have the `*.spicy` and `*.evt` source files for your new analyzer, you need to precompile them into an `*.hlto` object file containing their final executable code, which Zeek will then use. To do that, pass the relevant `*.spicy` and `*.evt` files to `spicyz`, then have Zeek load the output. To repeat the *example* from the *Getting Started* guide:

```
# spicyz -o my-http-analyzer.hlto my-http.spicy my-http.evt
# zeek -Cr request-line.pcap my-http-analyzer.hlto my-http.zeek
Zeek saw from 127.0.0.1: GET /index.html 1.0
```

Instead of providing the precompiled analyzer on the Zeek command line, you can also copy them into `${prefix}/lib/spicy/Zeek_Spicy/modules`. The Spicy plugin will automatically load any `*.hlto` object files it finds there. In addition, the plugin also scans Zeek's plugin directory for `*.hlto` files. Alternatively, you can override both of those locations by setting the environment variable `ZEEK_SPICY_MODULE_PATH` to a set of colon-separated directories to search instead. The plugin will then *only* look there. In all cases, the plugin searches any directories recursively, so it will find `*.hlto` also if they are nested in subfolders.

Run `spicyz -h` to see some additional options it provides, which are similar to *spicy-driver*.

2.7.5 Controlling Zeek from Spicy

Spicy grammars can import a provided library module `zeek` to gain access to Zeek-specific functions that call back into Zeek's processing:

`function zeek::confirm_protocol()`

[Deprecated] Triggers a DPD protocol confirmation for the current connection.

This function has been deprecated and will be removed. Use `spicy::accept_input` instead, which will have the same effect with Zeek.

`function zeek::reject_protocol(reason: string)`

[Deprecated] Triggers a DPD protocol violation for the current connection.

This function has been deprecated and will be removed. Use `spicy::decline_input` instead, which will have the same effect with Zeek.

`function zeek::weird(id: string, addl: string = "") : &cxxname="spicy::zeek::rt::weird"`

Reports a "weird" to Zeek. This should be used with similar semantics as in Zeek: something quite unexpected happening at the protocol level, which however does not prevent us from continuing to process the connection.

id: the name of the weird, which (just like in Zeek) should be a *static* string identifying the situation reported (e.g., `unexpected_command`).

addl: additional information to record along with the weird

`function zeek::is_orig() : bool`

Returns true if we're currently parsing the originator side of a connection.

`function zeek::uid() : string`

Returns the current connection's UID.

`function zeek::conn_id() : tuple<orig_h: addr, orig_p: port, resp_h: addr, resp_p: port>`

Returns the current connection's 4-tuple ID.

`function zeek::flip_roles()`

Instructs Zeek to flip the directionality of the current connection.

`function zeek::number_packets() : uint64`

Returns the number of packets seen so far on the current side of the current connection.

`function zeek::protocol_begin(analyzer: optional<string> = Null)`

Adds a Zeek-side child protocol analyzer to the current connection.

analyzer: type of analyzer to instantiate, specified through its Zeek-side name (similar to what Zeek's signature action *enable* takes); if not specified, Zeek will perform its usual dynamic protocol detection to figure out how to parse the data (the latter will work only for TCP protocols, though.)

`function zeek::protocol_data_in(is_orig: bool, data: bytes)`

Forwards protocol data to all previously instantiated Zeek-side child protocol analyzers.

is_orig: true to feed the data to the child's originator side, false for the responder data: chunk of data to forward to child analyzer

Note that due to Zeek limitations, any data passed through this function will always be forwarded to *all* currently active child analyzers.

`function zeek::protocol_gap(is_orig: bool, offset: uint64, len: uint64)`

Signals a gap in input data to all previously instantiated Zeek-side child protocol analyzers.

is_orig: true to signal gap to the child's originator side, false for the responder offset: start offset of gap in input stream len: size of gap

Note that due to Zeek limitations, any gaps signaled through this function will always be forwarded to *all* currently active child analyzers.

`function zeek::protocol_end()`

Signals end-of-data to all previously instantiated Zeek-side child protocol analyzers and removes them.

`function zeek::file_begin(mime_type: optional<string> = Null) : string`

Signals the beginning of a file to Zeek's file analysis, associating it with the current connection. Optionally, a mime type can be provided. It will be passed on to Zeek's file analysis framework. Returns the Zeek-side file ID of the new file.

`function zeek::fuid() : string`

Returns the current file's FUID.

`function zeek::terminate_session()`

Terminates the currently active Zeek-side session, flushing all state. Any subsequent activity will start a new session from scratch. This can only be called from inside a protocol analyzer.

`function zeek::file_set_size(size: uint64, fid: optional<string> = Null)`

Signals the expected size of a file to Zeek's file analysis.

size: expected size of file fid: Zeek-side ID of the file to operate on; if not given, the file started by the most recent file_begin() will be used

`function zeek::file_data_in(data: bytes, fid: optional<string> = Null)`

Passes file content on to Zeek's file analysis.

data: chunk of raw data to pass into analysis fid: Zeek-side ID of the file to operate on; if not given, the file started by the most recent file_begin() will be used

`function zeek::file_data_in_at_offset(data: bytes, offset: uint64, fid: optional<string> = Null)`

Passes file content at a specific offset on to Zeek's file analysis.

data: chunk of raw data to pass into analysis offset: position in file where data starts fid: Zeek-side ID of the file to operate on; if not given, the file started by the most recent file_begin() will be used

`function zeek::file_gap(offset: uint64, len: uint64, fid: optional<string> = Null)`

Signals a gap in a file to Zeek's file analysis.

offset: position in file where gap starts len: size of gap fid: Zeek-side ID of the file to operate on; if not given, the file started by the most recent file_begin() will be used

`function zeek::file_end(fid: optional<string> = Null)`

Signals the end of a file to Zeek's file analysis.

fid: Zeek-side ID of the file to operate on; if not given, the file started by the most recent file_begin() will be used

`function zeek::forward_packet(identifier: uint32)`

Inside a packet analyzer, forwards what data remains after parsing the top-level unit on to another analyzer. The index specifies the target, per the current dispatcher table.

`function zeek::network_time() : time`

Gets the network time from Zeek.

2.7.6 Dynamic Protocol Detection (DPD)

Spicy protocol analyzers support Zeek's *Dynamic Protocol Detection* (DPD), i.e., analysis independent of any well-known ports. To use that with your analyzer, add two pieces:

1. A Zeek signature to activate your analyzer based on payload patterns. Just like with any of Zeek's standard analyzers, a signature can activate a Spicy analyzer through the `enable "<name>"` keyword. The name of the analyzer comes out of the EVT file: it is the `ANALYZER_NAME` with the double colons replaced with an underscore (e.g., `spicy::HTTP` turns into `enable "spicy_HTTP"`.
2. You should call *spicy::accept_input()* from a hook inside your grammar at a point when the parser can be reasonably certain that it is processing the expected protocol. Optionally, you may also call *spicy::decline_input()* when you're sure the parser is *not* parsing the right protocol. However, the Zeek plugin will also trigger this automatically whenever your parser aborts with an error.

2.7.7 Configuration

Options

The Spicy plugin provides a set of script-level options to tune its behavior. These all live in the `Spicy::` namespace:

```
# Constant for testing if Spicy is available.
const available = T;

## Show output of Spicy print statements.
const enable_print = F &redef;

## abort() instead of throwing HILTI exceptions.
const abort_on_exceptions = F &redef;
```

(continues on next page)

(continued from previous page)

```
## Include backtraces when reporting unhandled exceptions.
const show_backtraces = F &redef;

## Maximum depth of recursive file analysis (Spicy analyzers only)
const max_file_depth: count = 5 &redef;
```

Functions

The Spicy plugin also adds the following new built-in functions to Zeek, which likewise live in the `Spicy::` namespace:

```
## Enable a specific Spicy protocol analyzer if not already active. If this
## analyzer replaces an standard analyzer, that one will automatically be
## disabled.
##
## tag: analyzer to toggle
##
## Returns: true if the operation succeeded
global enable_protocol_analyzer: function(tag: Analyzer::Tag) : bool;

## Disable a specific Spicy protocol analyzer if not already inactive. If
## this analyzer replaces an standard analyzer, that one will automatically
## be re-enabled.
##
## tag: analyzer to toggle
##
## Returns: true if the operation succeeded
global disable_protocol_analyzer: function(tag: Analyzer::Tag) : bool;


## Enable a specific Spicy file analyzer if not already active. If this
## analyzer replaces an standard analyzer, that one will automatically be
## disabled.
##
## tag: analyzer to toggle
##
## Returns: true if the operation succeeded
global enable_file_analyzer: function(tag: Files::Tag) : bool;

## Disable a specific Spicy file analyzer if not already inactive. If
## this analyzer replaces an standard analyzer, that one will automatically
## be re-enabled.
##
## tag: analyzer to toggle
##
## Returns: true if the operation succeeded
global disable_file_analyzer: function(tag: Files::Tag) : bool;
```

2.7.8 Debugging

If Zeek doesn't seem to be doing the right thing with your Spicy analyzer, there are several ways to debug what's going on. To facilitate that, compile your analyzer with `spicyz -d` and, if possible, use a debug version of Zeek (i.e., build Zeek with `./configure --enable-debug`).

If your analyzer doesn't seem to be active at all, first make sure Zeek actually knows about it: It should show up in the output of `zeek -NN Zeek::Spicy`. If it doesn't, you might not have been using the right `*.spicy` or `*.evt` files when precompiling, or Zeek is not loading the `*.hlto` file. Also check your `*.evt` if it defines your analyzer correctly.

If Zeek knows about your analyzer and just doesn't seem to activate it, double-check that ports or MIME types are correct in the `*.evt` file. If you're using a signature instead, try a port/MIME type first, just to make sure it's not a matter of signature mismatches.

If there's nothing obviously wrong with your source files, you can trace what the plugin is compiling by running `spicyz` with `-D zeek`. For example, reusing the *HTTP example* from the *Getting Started* guide:

```
# spicyz -D zeek my-http.spicy my-http.evt -o my-http.hlt
[debug/zeek] Loading Spicy file "/Users/robin/work/spicy/main/tests/spicy/doc/my-http.
↪spicy"
[debug/zeek] Loading EVT file "/Users/robin/work/spicy/main/doc/examples/my-http.evt"
[debug/zeek] Loading events from /Users/robin/work/spicy/main/doc/examples/my-http.evt
[debug/zeek]   Got protocol analyzer definition for spicy_MyHTTP
[debug/zeek]   Got event definition for MyHTTP::request_line
[debug/zeek] Running Spicy driver
[debug/zeek]   Got unit type 'MyHTTP::Version'
[debug/zeek]   Got unit type 'MyHTTP::RequestLine'
[debug/zeek] Adding protocol analyzer 'spicy_MyHTTP'
[debug/zeek] Adding Spicy hook 'MyHTTP::RequestLine::0x25_done' for event␣
↪MyHTTP::request_line
[debug/zeek] Done with Spicy driver
```

You can see the main pieces in there: The files being loaded, unit types provided by them, analyzers and events being created.

If that all looks as expected, it's time to turn to the Zeek side and see what it's doing at runtime. You'll need a debug version of Zeek for that, as well as a small trace with traffic that you expect your analyzer to process. Run Zeek with `-B dpd` (or `-B file_analysis` if you're debugging a file analyzer) on your trace to record the analyzer activity into `debug.log`. For example, with the same HTTP example, we get:

```
# zeek -B dpd -Cr request-line.pcap my-http.hlto
# cat debug.log
[dpd] Registering analyzer SPICY_MYHTTP for port 12345/1
[...[
[dpd] Available analyzers after zeek_init():
[...]
[dpd]     spicy_MyHTTP (enabled)
[...]
[dpd] Analyzers by port:
[dpd]     12345/tcp: SPICY_MYHTTP
[...]
[dpd] TCP[5] added child SPICY_MYHTTP[7]
[dpd] 127.0.0.1:59619 > 127.0.0.1:12345 activated SPICY_MYHTTP analyzer due to port␣
↪12345
[...]
[dpd] SPICY_MYHTTP[7] DeliverStream(25, T) [GET /index.html HTTP/1.0\x0a]
```

(continues on next page)

(continued from previous page)

```
[dpd] SPICY_MYHTTP[7] EndOfData(T)
[dpd] SPICY_MYHTTP[7] EndOfData(F)
```

The first few lines show that Zeek's analyzer system registers the analyzer as expected. The subsequent lines show that the analyzer gets activated for processing the connection in the trace, and that it then receives the data that we know indeed constitutes its payload, before it eventually gets shutdown.

To see this from the plugin's side, set the `zeek` debug stream through the `HILTI_DEBUG` environment variable:

```
# HILTI_DEBUG=zeek zeek -Cr request-line.pcap my-http.hlto
[zeek] Have Spicy protocol analyzer spicy_MyHTTP
[zeek] Registering Protocol::TCP protocol analyzer spicy_MyHTTP with Zeek
[zeek]   Scheduling analyzer for port 12345/tcp
[zeek] Done with post-script initialization
[zeek] [SPICY_MYHTTP/7/orig] initial chunk: |GET /index.html HTTP/1.0\\x0a|␣
↪(eod=false)
[zeek] [SPICY_MYHTTP/7/orig] -> event MyHTTP::request_line($conn, GET, /index.html, 1.
↪0)
[zeek] [SPICY_MYHTTP/7/orig] done with parsing
[zeek] [SPICY_MYHTTP/7/orig] parsing finished, skipping further originator payload
[zeek] [SPICY_MYHTTP/7/resp] no unit specified for parsing
[zeek] [SPICY_MYHTTP/7/orig] skipping end-of-data delivery
[zeek] [SPICY_MYHTTP/7/resp] no unit specified for parsing
[zeek] [SPICY_MYHTTP/7/orig] skipping end-of-data delivery
[zeek] [SPICY_MYHTTP/7/resp] no unit specified for parsing
```

After the initial initialization, you see the data arriving and the event being generated for Zeek. The plugin also reports that we didn't define a unit for the responder side—which we know in this case, but if that appears unexpectedly you probably found a problem.

So we know now that our analyzer is receiving the anticipated data to parse. At this point, we can switch to debugging the Spicy side *through the usual mechanisms*. In particular, setting `HILTI_DEBUG=spicy` tends to be helpful:

```
# HILTI_DEBUG=spicy zeek -Cr request-line.pcap my-http.hlto
[spicy] MyHTTP::RequestLine
[spicy]   method = GET
[spicy]   anon_2 =
[spicy]   uri = /index.html
[spicy]   anon_3 =
[spicy]   MyHTTP::Version
[spicy]     anon = HTTP/
[spicy]     number = 1.0
[spicy]   version = [$number=b"1.0"]
[spicy]   anon_4 = \n
```

If everything looks right with the parsing, and the right events are generated too, then the final part is to check out the events that arrive on the Zeek side. To get Zeek to see an event that the plugin raises, you need to have at least one handler implemented for it in one of your Zeek scripts. You can then load Zeek's `misc/dump-events` to see them as they are being received, including their full Zeek-side values:

```
# zeek -Cr request-line.pcap my-http.hlto misc/dump-events
[...]
1580991211.780489 MyHTTP::request_line
              [0] c: connection      = [id=[orig_h=127.0.0.1, orig_p=59619/tcp, ...] .
↪..]
              [1] method: string     = GET
```

(continues on next page)

(continued from previous page)

```
              [2] uri: string        = /index.html
              [3] version: string    = 1.0
[...]
```

2.8 Custom Host Applications

Spicy provides a C++ API for integrating its parsers into custom host applications. There are two different approaches to doing this:

1. If you want to integrate just one specific kind of parser, Spicy can generate C++ prototypes for it that facilitate feeding data and accessing parsing results.
2. If you want to write a generic host application that can support arbitrary parsers, Spicy provides a dynamic runtime introspection API for dynamically instantiating parsers and accessing results.

We discuss both approaches in the following.

Note: Internally, Spicy is a layer on top of an intermediary framework called HILTI. It is the HILTI runtime library that implements most of the functionality we'll look at in this section, so you'll see quite a bit of HILTI-side functionality. Spicy comes with a small additional runtime library of its own that adds anythings that's specific to the parsers it generates.

Note: The API for host applications is still in flux, and some parts aren't the prettiest yet. Specifics of this may change in future versions of HILTI/Spicy.

2.8.1 Integrating a Specific Parser

We'll use our simple HTTP example from the *Getting Started* section as a running example for a parser we want to leverage from a C++ application.

Listing 7: my-http.spicy

```
module MyHTTP;

const Token      = /[^ \t\r\n]+/;
const WhiteSpace = /[ \t]+/;
const NewLine    = /\r?\n/;

type Version = unit {
    :       /HTTP\//;
    number: /[0-9]+\.[0-9]+/;
};

public type RequestLine = unit {
    method:  Token;
    :        WhiteSpace;
    uri:     Token;
    :        WhiteSpace;
    version: Version;
```

(continues on next page)

(continued from previous page)

```
    :        NewLine;

    on %done {
        print self.method, self.uri, self.version.number;
        }
};
```

First, we'll use *spicyc* to generate a C++ parser from the Spicy source code:

```
# spicyc -c -g my-http.spicy -o my-http.cc
```

Option `-c` (aka `--output-c++`) tells `spicyc` that we want it to generate C++ code (rather than compiling everything down into executable code).

Option `-g` (aka `--disable-optimizations`) tells `spicyc` to not perform global optimizations. Optimizations are performed on all modules passed to a invocation of `spicyc` and can remove e.g., unused code. Since we generate output files with multiple invocations, optimizations could lead to incomplete code.

We also need `spicyc` to get generate some additional additional "linker" code implementing internal plumbing necessary for cross-module functionality. That's what `-l` (aka `--output-linker`) does:

```
# spicyc -l -g my-http.cc -o my-http-linker.cc
```

We'll compile this linker code along with the `my-http.cc`.

Next, `spicyc` can also generate C++ prototypes for us that declare (1) a set of parsing functions for feeding in data, and (2) a `struct` type providing access to the parsed fields:

```
# spicyc -P -g my-http.spicy -o my-http.h
```

The output of `-P` (aka `--output-prototypes`) is a bit convoluted because it (necessarily) also contains a bunch of Spicy internals. Stripped down to the interesting parts, it looks like this for our example:

```
[...]

namespace __hlt::MyHTTP {
    struct RequestLine : hilti::rt::trait::isStruct, hilti::rt::Controllable
↪<RequestLine> {
        std::optional<hilti::rt::Bytes> method{};
        std::optional<hilti::rt::Bytes> uri{};
        std::optional<hilti::rt::ValueReference<Version>> version{};
        [...]
    };

    struct Version : hilti::rt::trait::isStruct, hilti::rt::Controllable<Version> {
        std::optional<hilti::rt::Bytes> number{};
        [...]
    };

[...]
}

namespace hlt::MyHTTP::RequestLine {
    extern auto parse1(hilti::rt::ValueReference<hilti::rt::Stream>& data, const␣
↪std::optional<hilti::rt::stream::View>& cur) -> hilti::rt::Resumable;
    extern auto parse2(hilti::rt::ValueReference<__hlt::MyHTTP::RequestLine>& unit,␣
↪hilti::rt::ValueReference<hilti::rt::Stream>& data, const std::optional
↪<hilti::rt::stream::View>& cur) -> hilti::rt::Resumable;
```

(continues on next page)

(continued from previous page)

```
    extern auto parse3(spicy::rt::ParsedUnit& gunit, hilti::rt::ValueReference
↪<hilti::rt::Stream>& data, const std::optional<hilti::rt::stream::View>& cur) ->␣
↪hilti::rt::Resumable;
}

[...]
```

Todo: The `struct` declarations should move into the public namespace.

You can see the `struct` definitions corresponding to the two unit types, as well as a set of parsing functions with three different signatures:

`parse1` The simplest form of parsing function receives a stream of input data, along with an optional view into the stream to limit the region to parse if desired. `parse`` will internally instantiate an instance of the unit's `struct`, and then feed the unit's parser with the data stream. However, it won't provide access to what's being parsed as it doesn't pass back the `struct`.

`parse2` The second form takes a pre-instantiated instance of the unit's `struct` type, which parsing will fill out. Once parsing finishes, results can be accessed by inspecting the `struct` fields.

`parse3` The third form takes a pre-instantiated instance of a generic, type-erased unit type that the parsing will fill out. Accessing the data requires use of HILTI's reflection API, which we will discuss in *Supporting Arbitrary Parsers*.

Let's start by using `parse1()`:

Listing 8: my-http-host.cc

```
#include <iostream>

#include <hilti/rt/libhilti.h>

#include <spicy/rt/libspicy.h>

#include "my-http.h"

int main(int argc, char** argv) {
    assert(argc == 2);

    // Initialize runtime library.
    hilti::rt::init();

    // Create stream with $1 as data.
    auto stream = hilti::rt::reference::make_value<hilti::rt::Stream>(argv[1]);
    stream->freeze();

    // Feed data.
    hlt::MyHTTP::RequestLine::parse1(stream, {}, {});

    // Wrap up runtime library.
    hilti::rt::done();

    return 0;
}
```

This code first instantiates a stream from data giving on the command line. It freezes the stream to indicate that no further data will arrive later. Then it sends the stream into the `parse1()` function for processing.

We can now use the standard C++ compiler to build all this into an executable, leveraging `spicy-config` to add the necessary flags for finding includes and libraries:

```
# clang++ -o my-http my-http-host.cc my-http.cc my-http-linker.cc $(spicy-config --
↪cxxflags --ldflags)
# ./my-http $'GET index.html HTTP/1.0\n'
GET, /index.html, 1.0
```

The output comes from the execution of the `print` statement inside the Spicy grammar, demonstrating that the parsing proceeded as expected.

When using `parse1()` we don't get access to the parsed information. If we want that, we can use `parse2()` instead and provide it with a `struct` to fill in:

Listing 9: my-http-host.cc

```
#include <iostream>

#include <hilti/rt/libhilti.h>
#include <spicy/rt/libspicy.h>

#include "my-http.h"

int main(int argc, char** argv) {
    assert(argc == 2);

    // Initialize runtime libraries.
    hilti::rt::init();
    spicy::rt::init();

    // Create stream with $1 as data.
    auto stream = hilti::rt::reference::make_value<hilti::rt::Stream>(argv[1]);
    stream->freeze();

    // Instantiate unit.
    auto request = hilti::rt::reference::make_value<__hlt::MyHTTP::RequestLine>();

    // Feed data.
    hlt::MyHTTP::RequestLine::parse2(request, stream, {}, {});

    // Access fields.
    std::cout << "method : " << *request->method << std::endl;
    std::cout << "uri    : " << *request->uri << std::endl;
    std::cout << "version: " << *(*request->version)->number << std::endl;

    // Wrap up runtime libraries.
    spicy::rt::done();
    hilti::rt::done();

    return 0;
}
```

```
# clang++ -o my-http my-http-host.cc my-http-host.cc $(spicy-config --cxxflags --
↪ldflags)
# ./my-http $'GET index.html HTTP/1.0\n'
GET, /index.html, 1.0
```

(continues on next page)

(continued from previous page)

```
method : GET
uri    : /index.html
version: 1.0
```

Another approach to retrieving field values goes through Spicy hooks calling back into the host application. That's how the Zeek plugin operates. Let's say we want to execute a custom C++ function every time a `RequestList` has been parsed. By adding the following code to `my-http.spicy`, we (1) declare that function on the Spicy-side, and (2) implement a Spicy hook that calls it:

Listing 10: my-http.spicy

```
public function got_request_line(method: bytes, uri: bytes, version_number: bytes) :␣
↪void &cxxname="got_request_line";

on RequestLine::%done {
    got_request_line(self.method, self.uri, self.version.number);
}
```

The `&cxxname` attribute for `got_request_line` indicates to Spicy that this is a function implemented externally inside custom C++ code, accessible through the given name. Now we need to implement that function:

Listing 11: my-http-callback.cc

```
#include <iostream>

#include <hilti/rt/libhilti.h>

#include <spicy/rt/libspicy.h>

void got_request_line(const hilti::rt::Bytes& method, const hilti::rt::Bytes& uri,␣
↪const hilti::rt::Bytes& version_number) {
    std::cout << "In C++ land: " << method << ", " << uri << ", " << version_number <
↪< std::endl;
}
```

Finally, we compile it altogether:

```
# spicyc -c -g my-http.spicy -o my-http.cc
# spicyc -l -g my-http.cc -o my-http-linker.cc
# spicyc -P -g my-http.spicy -o my-http.h
# clang++ -o my-http my-http.cc my-http-linker.cc my-http-callback.cc my-http-host.cc
↪$(spicy-config --cxxflags --ldflags)
# ./my-http $'GET index.html HTTP/1.0\n'
In C++ land: GET, index.html, 1.0
GET, index.html, 1.0
```

Note that the C++ function signature needs to match what Spicy expects, based on the Spicy-side prototype. If you are unsure how Spicy arguments translate into C++ arguments, look at the C++ prototype that's included for the callback function in the output of `-P`.

A couple more notes on the compilation process for integrating Spicy-generated code into custom host applications:

- Above we used `spicyc -l` to link our Spicy code from just a single Spicy source file. If you have more than one source file, you need to link them altogether in a single step. For example, if we had `A.spicy`, `B.spicy` and `C.spicy`, we'd do:

```
# spicyc -c -g A.spicy -o A.cc
# spicyc -c -g B.spicy -o B.cc
# spicyc -c -g C.spicy -o C.cc
# spicyc -l -g A.cc B.cc C.cc -o linker.cc
# clang++ A.cc B.cc C.cc linker.cc -o a.out ...
```

- If your Spicy code is importing any library modules (e.g., the standard `filter` module), you'll need to compile those as well in the same fashion.

2.8.2 Supporting Arbitrary Parsers

This approach is more complex, and we'll just briefly describe the main pieces here. All of the tools coming with Spicy support arbitrary parsers and can serve as further examples (e.g., *spicy-driver*, *spicy-dump*, the *Zeek plugin*). Indeed, they all build on the same C++ library class `spicy::rt::Driver` that provides a higher-level API to working with Spicy's parsers in a generic fashion. We'll do the same in the following.

Retrieving Available Parsers

The first challenge for a generic host application is that it cannot know what parsers are even available. Spicy's runtime library provides an API to get a list of all parsers that are compiled into the current process. Continuing to use the `my-http.spicy` example, this code prints out our one available parser:

Listing 12: my-http-host.cc

```
#include <iostream>

#include <hilti/rt/libhilti.h>

int main(int argc, char** argv) {
    assert(argc == 2);

    // Initialize runtime libraries.
    hilti::rt::init();
    spicy::rt::init();

    // Instantiate driver providing higher level parsing API.
    spicy::rt::Driver driver;

    // Print out available parsers.
    print(unit->value());

    // Wrap up runtime libraries.
    spicy::rt::done();
    hilti::rt::done();

    return 0;
}
```

```
# clang++ -o my-http my-http-host.cc my-http.cc my-http-linker.cc $(spicy-config --
↪cxxflags --ldflags)
# ./my-http
Available parsers:

    MyHTTP::RequestLine
```

Using the name of the parser (`MyHTTP::RequestLine`) we can instantiate it from C++, and then feed it data:

```
    // Retrieve meta object describing parser.
    auto parser = driver.lookupParser("MyHTTP::RequestLine");
    assert(parser);

    // Fill string stream with $1 as data to parse.
    std::stringstream data(argv[1]);

    // Feed data.
    auto unit = driver.processInput(**parser, data);
```

```
# clang++ -o my-http my-http-host.cc my-http.cc my-http-linker.cc $(spicy-config --
↪cxxflags --ldflags)
# ./my-http $'GET index.html HTTP/1.0\n'
GET, /index.html, 1.0
```

That's the output of the `print` statement once more.

`unit` is of type `spicy::rt::ParsedUnit`, which is a type-erased class holding, in this case, an instance of `_hlt::MyHTTP::RequestLine`. Internally, that instance went through the `parse3()` function that we have encountered in the previous section. To access the parsed fields, there's a visitor API to iterate generically over HILTI types like this unit:

```
void print(const hilti::rt::type_info::Value& v) {
    const auto& type = v.type();
    switch ( type.tag ) {
        case hilti::rt::TypeInfo::Bytes: std::cout << type.bytes->get(v); break;
        case hilti::rt::TypeInfo::ValueReference: print(type.value_reference->
↪value(v)); break;
        case hilti::rt::TypeInfo::Struct:
            for ( const auto& [f, y] : type.struct_->iterate(v) ) {
                std::cout << f.name << ": ";
                print(y);
                std::cout << std::endl;
            }
            break;
        default: assert(false);
    }
}
```

Adding `print(unit->value()` after the call to `processInput()` then gives us this output:

```
# clang++ -o my-http my-http-host.cc my-http.cc my-http-linker.cc $(spicy-config --
↪cxxflags --ldflags)
# ./my-http $'GET index.html HTTP/1.0\n'
GET, /index.html, 1.0
method: GET
uri: /index.html
version: number: 1.0
```

Our visitor code implements just what we need for our example. The source code of `spicy-dump` shows a full implementation covering all available types.

So far we have compiled the Spicy parsers statically into the generated executable. The runtime API supports loading them dynamically as well from pre-compiled `HLTO` files through the class `hilti::rt::Library`. Here's the full example leveraging that, taking the file to load from the command line:

Listing 13: my-driver

```
int main(int argc, char** argv) {
    // Usage now: "my-driver <hlto> <name-of-parser> <data>"
    assert(argc == 4);

    // Load pre-compiled parser. This must come before initializing the
    // runtime libraries.
    auto library = hilti::rt::Library(argv[1]);
    auto rc = library.open();
    assert(rc);

    // Initialize runtime libraries.
    hilti::rt::init();
    spicy::rt::init();

    // Instantiate driver providing higher level parsing API.
    spicy::rt::Driver driver;

    // Print out available parsers.
    driver.listParsers(std::cout);

    // Retrieve meta object describing parser.
    auto parser = driver.lookupParser(argv[2]);
    assert(parser);

    // Fill string stream with $1 as data to parse.
    std::stringstream data(argv[3]);

    // Feed data.
    auto unit = driver.processInput(**parser, data);
    assert(unit);

    // Print out conntent of parsed unit.
    print(unit->value());

    // Wrap up runtime libraries.
    spicy::rt::done();
    hilti::rt::done();

    return 0;
}
```

```
# $(spicy-config --cxx) -o my-driver my-driver.cc $(spicy-config --cxxflags --ldflags --dynamic-loading)
# spicyc -j my-http.spicy >my-http.hlto
# ./my-driver my-http.hlto "$(cat data)"
Available parsers:

    MyHTTP::RequestLine

GET, /index.html, 1.0
method: GET
uri: /index.html
version: number: 1.0
```

Note: Note the addition of `--dynamic-loading` to the `hilti-config` command line. That's needed when the resulting binary will dynamically load precompiled Spicy parsers because linker flags need to be slightly adjusted in that case.

2.8.3 API Documentation

We won't go further into details of the HILTI/Spicy runtime API here. Please see *C++ API documentation* for more on that, the namespaces `hilti::rt` and `spicy::rt` cover what's available to host applications.

Our examples always passed the full input at once. You don't need to do that, Spicy's parsers can process input incrementally as it comes in, and return back to the caller to retrieve more. See the source of spicy::Driver::processInput() for an example of how to implement that.

2.9 Release Notes

This following summarizes the most important changes in recent Spicy releases. For an exhaustive list of all changes, see the #%sCHANGES file coming with the distribution.

2.9.1 Version 1.7

New Functionality

- Support Zeek-style documentation strings in Spicy source code.
- Provide ability for host applications to initiate runtime's module-pre-init phase manually.
- Add DPD-style `spicy::accept_input()` and `spicy::decline_input()`.
- Add driver option to output full set of generated C++ files.
- GH-1123: Support arbitrary expression as argument to type constructors, such as `interval(...)`.

Changed Functionality

- Search `HILTI_CXX_INCLUDE_DIRS` paths before default include paths.
- Search user module paths before system paths.
- Streamline runtime exception hierarchy.
- Fix bug in cast from `real` to `interval`.
- GH-1326: Generate proper runtime types for enums.
- GH-1330: Reject uses of imported module IDs as expression.

Bug fixes

- GH-1310: Fix ASAN false positive with GCC.
- GH-1345: Improve runtime performance of stream iteration.
- GH-1367: Use unique filename for all object files generated during JIT.
- Remove potential race during JIT when using `HILTI_CXX_COMPILER_LAUNCHER`.
- GH-1349: Fix incremental regexp matching for potentially empty results.

Documentation

2.9.2 Version 1.6

New Functionality

- GH-1249: Allow combining `&eod` with `&until` or `&until-including`.
- GH-1251: When decoding bytes into a string using a given character set, allow caller to control error handling.

  All methods taking a charset parameters now take an additional enum selecting 1 of 3 possible error handling strategies in case a character can't be decoded/represented: `STRICT` throws an error, `IGNORE` skips the problematic character and proceeds with the next, and `REPLACE` replaces the problematic character with a safe substitute. `REPLACE` is the default everywhere now, so that by default no errors are triggered.

  This comes with an additional functional change for the ASCII encoding: we now consistently sanitize characters that ASCII can't represent when in `REPLACE`/`IGNORE` modes (and, hence, by default), and trigger errors in `STRICT` mode. Previously, we'd sometimes let them through, and never triggered any errors. This also fixes a bug with the ASCII encoding sometimes turning a non-printable character into multiple repeated substitutes.
- GH-1294: Add library function to parse an address from string or bytes.
- HLTO files now perform a version check when loaded.

  We previously would potentially allow building a HLTO file against one version of the Spicy runtime, and then load it with a different version. If exposed symbols matched loading might have succeeded, but could still have lead to sublte bugs at runtime.

  We now embed a runtime version string in HLTO files and reject loading HLTO files into a different runtime version. We require an exact version match.
- New `pack` and `unpack` operators.

  These provide low-level primitives for transforming a value into, or out of, a binary representations, see *the docs* for details.

Changed Functionality

- GH-1236: Add support for adding link dependencies via `--cxx-link`.
- GH-1285: C++ identifiers referenced in `&cxxname` are now automatically interpreted to be in the global namespace.
- Synchronization-related debug messages are now logged to the `spicy-verbose` stream. We added logging of successful synchronization.
- Downgrade required Flex version. We previously required at least flex-2.6.0; we can now build against flex-2.5.37.

- Improve C++ caching during JIT.

  We improved caching behavior via `HILTI_CXX_COMPILER_LAUNCHER` if the configuration of `spicyc` was changed without changing the C++ file produced during JIT.

- `hilti::rt::isDebugVersion` has been removed.
- The `-O | --optimize` flag has been removed from command line tools.

  This was already a no-op without observable side-effects.

- GH-1311: Reject use of `context()` unit method if unit does not declare a context with `%context`.
- GH-1319: Unsupported unit variable attributes are now rejected.
- GH-1299: Add validator for bitfield field ranges.
- We now reject uses of `self` as an ID.
- GH-1233: Reject key types for maps that can't be sorted.
- Fix validator for field `&default` expression types for constness.

  When checking types of field `&default` expressions we previously would also consider their constness. This breaks e.g., cases where the used expression is not a LHS like the field the `&default` is defined for,

  ```
  type X = unit {
      var x: bytes = b"" + a;
  };
  ```

  We now do not consider constness in the type check anymore. Since fields are never const this allows us to set a `&default` with constant expressions as well.

Bug fixes

- GH-1231: Add special handling for potential `advance` failure in trial mode.
- GH-1115, GH-1196: Explicitly type temporary value used by `&max_size` logic.
- GH-1143, GH-1220: Add coercion on assignment for optionals that only differ in constness of their inner types.
- GH-1230: Add coercion to default argument of `map::get`.
- GH-1234, GH-1238: Fix assertions with anonymous struct constructor.
- GH-1248: Fix `stop` for unbounded loop.
- GH-1250: Fix internal errors when seeing unsupported character classes in regular expression.
- GH-1170: Fix contexts not allowing being passed `inout`.
- GH-1266: Fix wrong type for Spicy-side `self` expression.
- GH-1261: Fix inability to access unit fields through `self` in `&convert` expressions.
- GH-1267: Install only needed headers from bundled SafeInt library.
- GH-1227: Fix code generation when a module's file could be imported through different means.
- GH-1273: Remove bundled code licensed under CPOL license.
- GH-1303: Fix potentially late synchronization when jumping over gaps during synchronization.
- Do not force gold linker with user-provided linker flags or when built as a CMake subproject.
- Improve efficiency of `startsWith` for long inputs.

Documentation

- The documentation now reflects Zeek package manager Spicy feature templates.
- The documentation for bitfields was clarified.
- Documentation for casts from integers to boolean was added.
- We added documentation for how to expose custom C++ code in Spicy.
- Update doc link to commits mailing list.
- Clarify that `%context` can only be used in top-level units.
- Clarify that `&until` consumes the delimiter.
- GH-1240: Clarify docs on `SPICY_VERSION`.
- Add FAQ item on source locations.
- Add example for use of `?.`.

2.9.3 Version 1.5

New Functionality

- GH-1179: Cap parallelism use for JIT background jobs.

  During JIT, we would previously launch all compilation jobs in parallel. For projects using many modules this could have lead to resource contention which often forced users to use sequential compilation with `HILTI_JIT_SEQUENTIAL`. We now by default cap the number of parallel background jobs at the number of logical cores. This can be parameterized with the environment variable `HILTI_JIT_PARALLELISM` which for `HILTI_JIT_PARALLELISM=1` reproduces `HILTI_JIT_SEQUENTIAL`.

- GH-1134: Add support for `synchronize-at` and `synchronize-after` properties.

  These unit properties allow specifying a literal which should be searched for during error recovery. If the respective unit is used as a synchronize point during error recovery, i.e., it is used as a field which is marked `&synchronize`, input resynchronization during error recovery will seek to the next position of this pattern in the input stream.

- GH-1209: Provide error message to `%error` handler.

  > We now allow to optionally provide a string parameter with `%error` that will receive the associated error message:

  ```
  on %error(msg: string) { print msg; }
  ```

Changed Functionality

- GH-1184: Allow more cache hits if only a few modules are changed in multi-module compilation.
- GH-1208: Incremental performance tweaks for JIT.
- GH-1197: Make handling of sanitizer workarounds more granular.

Bug fixes

- GH-1150: Preserve additional permissions from umask when generating HLTO files.
- GH-1154: Add stringificaton of `Map::value_type`.
- GH-1080: Reject constant declarations at non-global scope.
- GH-1164: Make compiler plugin initialization explicit.
- GH-1050: Update location when entering most parser methods.
- GH-1187: Fix support for having multiple source modules of the same name.
- GH-1197: Prevent too early integer overflow in pow.
- GH-1201: Adjust removal of symlinks on install for `DESTDIR`.
- GH-1203: Allow changing `DESTDIR` between configure and install time.
- GH-1204: Remove potential use-after-move.
- GH-1210: Prevent unnecessarily executable stack with GNU toolchain.
- GH-1206: Fix detection of recursive dependencies.
- GH-1217: Produce `hilti::rt::Bool` when casting to boolean.
- GH-1224: Fix import segfault.

Documentation

- GH-44: Update docs for spicy-plugin rename `_Zeek::Spicy` -> `Zeek::Spicy`.
- GH-1183: Update docs for Discourse migration [skip CI].
- GH-1205: Update Spicy docs for now being built into Zeek.

2.9.4 Version 1.4

New Functionality

- Add support for recovery from parse errors or incomplete input

  This release adds support for recovering from parse errors or incomplete input (e.g., gaps or partial connections). Grammars can denote unit synchronization points with a `&synchronize` attribute. If an error is encountered while extracting a previous fields, parsing will attempt to resynchronize the input at that point. The synchronization result needs to be checked and confirmed or rejected explicitly; a number of hooks are provided for that. See *the docs* for details.

- Remove restriction that units used as sinks need to be `public`
- Uses `ccache` for C++ compilation during JIT if Spicy itself was configured to use `ccache`

  Spicy spends a considerable amount of JIT time compiling generated C++ code. This work can be cached if neither inputs nor any of the used flags have changed so that subsequent JIT runs can complete much faster.

  We now automatically cache many C++ compilation artifacts with `ccache` if Spicy itself was configured with e.g., `--with-hilti-compiler-launcher=ccache`. This behavior can be controlled or disabled via the `HILTI_CXX_COMPILER_LAUNCHER` environment variable.

- GH-842: Add Spicy support for struct initialization.

- GH-1036: Support unit initialization through a struct constructor expression.

Changed Functionality

- GH-1074: `%random-access` is now derived automatically from uses and declaring it explicitly has been deprecated.
- GH-1072: Disallow enum declarations with non-unique values.

  It is unclear what code should be generated when requested to convert an integer value to the following enum:

  ```
  type E = enum {
      A = 1,
      B = 2,
      C = 1,
  };
  ```

  For `1` we could produce either `E::A` or `E::C` here.

  Instead of allowing this ambiguity we now disallow enums with non-unique values.

Bug fixes

- Prevent exception if cache directory is not readable.
- Propagate failure from `cmake` up to `./configure`.
- GH-1030: Make sure types required for globals are declared before being used.
- Fix potentially use-after-free in stringification of `stream::View`.
- GH-1087: Make `offset` return correct value even before parsing of field.

Documentation

2.9.5 Version 1.3

New Functionality

- Add optimizer removing unused `%random-access` or `%filter` functionality

  If a unit has e.g., a `%random-access` attribute Spicy emits additional code to track and update offsets. If the `%random-access` functionality is not used this leads to unneeded code being emitted which causes unneeded overhead, both during JIT and during execution.

  We now emit such feature-dependent code under a feature flag (effectively a global boolean constant) which is by default *on*. Additionally, we added an optimizer pass which detects whether a feature is used and can disable unused feature functionality (switching the feature flag to *off*), and can then remove unreachable code behind such disabled feature flags by performing basic constant folding.
- Add optimizer pass removing unused sink functionality

  By default any unit declared `public` can be used as a sink. To support sink behavior additional code is emitted and invoked at runtime, regardless of whether the unit is used as a sink or not.

  We now detect unused sink functionality and avoid emitting it.
- GH-934: Allow `$$` in place of `self` in unit convert attributes.

Changed Functionality

- GH-941: Allow use of units with all defaulted parameters as entry points.
- We added precompilation support for `libspicy.h`.
- Drop support for end-of-life Fedora 32, and add support for Fedora 34.

Bug fixes

- Correctly handle lookups for NULL library symbols.
- Use safe integers for `size` functions in the runtime library.
- Make it possible to build on ARM64.
- Fix building with gcc-11.

Documentation

2.9.6 Version 1.2

New Functionality

- GH-913: Add support for switch-level `&parse-at` and `&parse-from` attributes inside a unit.
- Add optimizer pass removing unimplemented functions and methods.

  This introduces a global pass triggered after all individual input ASTs have been finalized, but before we generate any C++ code. We then strip out any unimplemented member functions (typically Spicy hooks), both their definitions as well as their uses.

  In order to correctly handle previously generated C++ files which might have been generated with different optimization settings, we disallow optimizations if we detect that a C++ input file was generated by us.

Changed Functionality

- Add validation of unit switch attributes. We previously silently ignored unsupported attributes; now errors are raised.
- Remove configure option `--build-zeek-plugin`. Spicy no longer supports building the Zeek plugin/analyzers in-tree. This used to be available primarily for development purposes, but became challenging to maintain.
- Add environment variable `HILTI_CXX_INCLUDE_DIRS` to specify additional C++ include directories when compiling generated code.
- GH-940: Add runtime check for parsing progress during loops.

Bug fixes

- Fix computation of unset locations.
- Fix accidental truncating conversion in integer code.

Documentation

2.9.7 Version 1.1

New Functionality

- GH-844: Add support for `&size` attribute to unit `switch` statement.
- GH-26: Add `%skip`, `%skip-pre` and `%skip-post` properties for skipping input matching a regular expression before any further input processing takes place.
- Extend library functionality provided by the `spicy` module:
  - `crc32_init()/crc32_add()` compute CRC32 checksums.
  - `mktime()` creates a `time` value from individual components.
  - `zlib_init()` initializes a `ZlibStream` with a given window bits argument.
  - `Zlib` now accepts a window bits parameter.
- Add a new `find()` method to units for that searches for a `bytes` sequence inside their input data, forward or backward from a given starting position.
- Add support for `&chunked` when parsing bytes data with `&until` or `&until_including`.
- Add `encode()` method to `string` for conversion to `bytes`.
- Extend parsing of `void` fields:
  - Add support for `&eod` to skip all data until the end of the current input is encountered.
  - Add support for `&until` to skip all data until a deliminator is encountered. The deliminator will be extracted from the stream before continuing.
- Port Spicy to Apple silicon.
- Add Dockerfile for OpenSUSE 15.2.

Changed Functionality

- Reject `void` fields with names.
- Lower minimum required Python version to 3.2.
- GH-882: Lower minimum required Bison version to 3.0.

Bug fixes

- GH-872: Fix missing normalization of enum label IDs.
- GH-878: Fix casting integers to enums.
- GH-889: Fix hook handling for anonymous void fields.
- GH-901: Fix type resolution bug in `&convert`.
- Fix handling of `&size` attribute for anonymous void fields.
- Fix missing update to input position before running `%done` hook.
- Add validation rejecting `$$` in hooks not supporting it.
- Make sure container sizes are runtime integers.
- Fix missing operator<< for enums when generating debug code.
- GH-917: Default-initialize forwarding fields without type arguments.

Documentation

- GH-37: Add documentation on how to skip data with `void` fields.

2.9.8 Migrating from the old prototype

Below we summarize language changes in Spicy compared to the original research prototype. Note that some of the prototype's more advanced functionality has not yet been ported to the new code base; see the corresponding list on GitHub for what's still missing.

Changes:

- Renamed `export` linkage to `public`.
- Renamed `%byteorder` property to `%byte-order`.
- Renamed `&byteorder` attribute to `&byte-order`.
- Renamed `&bitorder` attribute to `&bit-order`.
- All unit-level properties now need to conclude with a semicolon (e.g., `%filter;`).
- Renamed `&length` attribute to `&size`.
- Renamed `&until_including` attribute to `&until-including`.
- Replaced `&parse` with separate `&parse-from` (taking a "bytes" instance) and `&parse-at` (taking a stream iterator) attributes.
- Attributes no longer accept their arguments in parentheses, it now must `<attr>=expr`. (Before, both versions were accepted.)
- `uint<N>` and `int<N>` are no longer accepted, use `uintN/intN` instead (which worked before already as well)
- `list<T>` is no longer supported, use `vector<T>` instead.
- New syntax for parsing sequences: Use `x: int8[5]` instead of `x: vector<int8> &length=5`. For lists of unknown size, use `x: int8[]`. When parsing sequences sub-units, use: `x: Item[]`; or, if further arguments/attributes are required, `x: (Item(1,2,3))[]`. (The latter syntax isn't great, but the old syntax was ambiguous.)

- New syntax for functions: `function f(<params>) [: <result>]` instead of `<result> f(<params>)`
- Renamed runtime support module from `Spicy` to `spicy` (so use `import spicy`)
- In units, variables are now initialized to default values by default. Previously, that was (inconsistently) happening only for variables of type sink. To revert to the old behaviour, add "&optional" to the variable.
- Renamed type `double` to `real`.
- Generally, types don't coerce implicitly to bool anymore except in specific language contexts, such as in statements with boolean conditions.
- Filters can now be implemented in Spicy itself. The pre-built `filter::Base64Decode` and `filter::Zlib` provide the base64 and zlib functionality of the previously built-in filters.
- `{unit,sink}::add_filter` are renamed to `{unit,sink}::connect_filter`.
- Enums don't coerce to bool anymore, need to manually compare to `Undef`.
- Coercion to bool now happens only in certain contexts, like `if`-conditions (similar to C++).
- The sink method `sequence` has been renamed to `sequence_number`.
- The effect of the sink method `set_initial_sequence_number` no longer persists when reconnecting a different unit to a sink.
- `&transient` is no longer a supported unit field attribute. The same effect can now be achieved through an anonymous field (also see next point).
- `$$` can now be generally used in hooks to refer to the just parsed value. That's particularly useful inside hooks for anonymous fields, including fields that previously were `&transient` (see above). Previously, "$$" worked only for container elements in `foreach` hooks (which still operates the same way).
- Fields of type `real` are parsed with `&type` attribute (e.g., `&type=Spicy::RealType::IEEE754_Double`). They used to `&precision` attributes with a different enum type.
- Assigning to unit fields and variables no longer triggers any hooks. That also means that hooks are generally no longer supported for variables (This is tricky to implement, not clear it's worth the effort.)
- When importing modules, module names are now case-sensitive.
- When parsing vectors/lists of integers of a given length, use `&count` instead of `&length`.
- Zeek plugin:
  - `Bro::dpd_confirm()` has been renamed to `zeek::confirm_protocol()`. There's also a corresponding `zeek::reject_protocol()`.
  - To auto-export enums to Zeek, they need to be declared public.

2.10 Developer's Manual

2.10.1 Architecture

Components & Data Flow

Runtime Libraries

HILTI and Spicy each come with their own runtime libraries providing functionality that the execution of compiled code requires. The bulk of the functionality here resides with the HILTI side, with the Spicy runtime adding pieces that are specific to its use case (i.e., parsing).

Conceptually, there are a few different categories of functionality going into these runtime libraries, per the following summary.

Categories of Functionality

Category 1 Public library functionality that Spicy programs can `import` (e.g., functions like `spicy::current_time()` inside the `spicy` module; filters like `filter::Zlib` inside the `filter` module). This functionality is declared in `spicy/lib/*.spicy` and implemented in C++ in `libspicy-rt.a`.

Category 2 Public library functionality that HILTI programs can `import` (e.g., the `hilti::print()` function inside the `hilti` module). This functionality is declared in `hilti/lib/hilti.hlt` and implemented in C++ in `libhilti-rt.a`.

> **Note:** "Public functionality" here means being available to any *HILTI* program. This functionality is *not* exposed inside Spicy, and hence usually not visible to users unless they happen to start writing HILTI programs (e.g., when adding test cases to the code base).

Category 3 Public library functionality for C++ host applications to `#include` for interacting with the generated C++ code (e.g., to retrieve the list of available Spicy parsers, start parsing, and gain access to parsed values). This is declared inside the `hilti:rt` C++ namespace by `hilti/include/rt/libhilti.h` for HILTI-side functionality; and inside the `spicy::rt` namespace by `spicy/include/rt/libspicy.h` for purely Spicy-side functionality. This functionality is implemented in `libhilti-rt.a` and `libspicy-rt.a`, respectively.

> **Note:** Everything in the sub-namespaces `{hilti,spicy}::rt::detail` remains private and is covered by categories 4 and 5.

Category 4 Private Spicy-side library functionality that the HILTI code coming out of Spicy compilation can `import` (e.g., functions to access parsing input, such as `spicy_rt::waitForInput()`; HILTI-side type definitions for Spicy-specific types, such as for a `sink`). This functionality is declared in `spicy/lib/spicy_rt.spicy` and implemented in C++ in `libspicy-rt.a`.

Category 5 Private HILTI-side library functionality for use by C++ code generated from HILTI code. This is declared by `hilti/include/rt/libhilti.h` inside the `hilti::rt::detail` namespace. The functionality is implemented in `libhilti-rt.a`. (The generated C++ code uses public `hilti::rt` functionality from Category 3 as well.)

> **Note:** This category does not have any Spicy-side logic (by definition, because Spicy does not generate C++ code directly). Everything in `libspicy-rt.a`, and `spicy::rt::detail` is covered by one of the other categories.

What goes where?

Think of Category 1 as the "Spicy standard library": functionality for user-side Spicy code to leverage.

Category 2 is the same for HILTI, except that the universe of HILTI users remains extremely small right now (it's just Spicy and people writing tests).

Category 3 is our client-side C++ API for host applications to drive Spicy parsers and retrieve results.

When adding new functionality, one needs to decide between the HILTI and Spicy sides. Rules of thumb:

1. If it's "standard library"-type stuff that's meant for Spicy users to `import`, make it part of Category 1.
2. If it's something that's specific to parsing, add it to the Spicy side, either Category 3 for public functionality meant to be used by host applications; or Category 4 if it's something needed just by the generated HILTI code doing the parsing.
3. If it's something that's generic enough to be used by other HILTI applications (once we get them), add it to the HILTI side, either Category 2 or 5. Think, e.g., a Zeek script compiler.

2.10.2 Testing

Spicy's testing & CI setup includes several pieces that we discuss in the following.

TLDR; If you make changes, make sure that `make check` runs through. You need the right `clang-format` (see *Formatting*) and `clang-tidy` (see *Static analysis*) versions for that (from Clang >=10). If you don't have them (or want to save time), run at least `make test`. If that still takes too long for you, run `make test-core`.

BTest

Most tests are end-to-end tests that work from Spicy (or HILTI) source code and check that everything compiles and produces the expected output. We use BTest to drive these, very similar to Zeek. `make test` from the top-level directory will execute these tests. You get the same effect by changing into `tests/` and running `btest -j` there (`-j` parallelizes test execution).

The most important BTest options are:

- `-d` prints debugging output for failing tests to the console
- `-f diag.log` records the same debugging output into `diag.log`
- `-u` updates baselines when output changes in expected ways (don't forget to commit the updates)

There are some alternatives to running just all tests, per the following:

Running tests using installation after `make install`

By default, btests are running completely out of the source & build directories. If you run `btest -a installation`, BTest will instead switch to pulling everything from their installation locations. If you have already deleted the build directory, you also need to have the environment variable `SPICY_INSTALLATION_DIRECTORY` point to your installation prefix, as otherwise BTest has no way of knowing where to find Spicy.

Unit tests

There's a growing set of units test. These are using `doctest` and are executed through `btest` as well, so just running tests per above will have these included.

Alternatively, the test binaries in the build directory can be executed to exercise the tests, or one can use the `check` build target to execute all unit tests.

Sanitizers

To build tools and libraries with support for Clang's address/leak sanitizer, configure with `--enable-sanitizer`. If Clang's `asan` libraries aren't in a standard runtime library path, you'll also need to set `LD_LIBRARY_PATH` (Linux) or `DYLD_LIBRARY_PATH` (macOS) to point there (e.g., `LD_LIBRARY_PATH=/opt/clang9/lib/clang/9.0.1/lib/linux`).

When using the Spicy plugin for Zeek and Zeek hasn't been compiled with sanitizer support, you'll also need to set `LD_PRELOAD` (Linux) or `DYLD_INSERT_LIBRARIES` (macOS) to the shared `asan` library to use (e.g., `LD_PRELOAD=/data/clang9/lib/clang/9.0.1/lib/linux/libclang_rt.asan-x86_64.so`). Because you probably don't want to set that permanently, the test suite pays attention to a variable `ZEEK_LD_PRELOAD`: If you set that before running `btest` to the path you want in `LD_PRELOAD`, the relevant tests will copy the value for running Zeek.

To make the sanitizer symbolize its output you need to set the `ASAN_SYMBOLIZER_PATH` environment variable to point to the `llvm-symbolizer` binary, or make sure `llvm-symbolizer` is in your `PATH`.

Note: As we are running one of the CI build with sanitizers, it's ok not to run this locally on a regular basis during development.

Code Quality

Our CI runs the *Formatting* and *Static analysis* checks, and will fail if any of that doesn't pass. To execute these locally, run the make target `format` and `tidy`, respectively. Don't forget to set `CLANG_FORMAT` and `CLANG_TIDY` to the right version of the binary if they aren't in your `PATH`.

CI also runs pre-commit with a configuration pre-configured in `.pre-commit-config.yaml`. To run that locally on every commit, install pre-commit and then put its git hook in place through executing `pre-commit install`; see the installation instructions for more details.

Docker Builds

We are shipping a number of Docker files in `docker/`; see *Using Docker* for more information. As part of our CI, we make sure these build OK and pass `btest -a installation`. If you have Docker available, you can run these individually yourself through `make test-<platform>` in `docker/`. However, usually it's fine to leave this to CI.

How Test Your Branch

If you run `make check` in the top-level directory you get the combination of all the btests, formatting, and linting. That's the best check to do to make sure your branch is in good shape, in particular before filing a pull request.

2.10.3 Debugging

The user manual's *debugging section* serves as a good starting point for development-side debugging as well—it's often the same mechanisms that help understand why something's not working as expected. In particular, looking at the generated HILTI & C++ code often shows quickly what's going on.

That section describes only runtime debugging output. The Spicy toolchain also has a set of compile-time debug output streams that shine light on various parts of the compiler's operation. To activate that output, both `spicyc` and `spicy-driver` (and `hiltic` as well) take a `-D` option accepting a comma-separated list of stream tags. The following choices are available:

`ast-dump-iterations` The compiler internally rewrites ASTs in multiple rounds until they stabilize. Activating this stream will print the ASTs into files `dbg.*` on disk after each round. This is pretty noisy, and maybe most helpful as a last resort when it's otherwise hard to understand some aspects of AST processing without seeing really *all* the changes.

`ast-final` Prints out all the final ASTs, with all transformations, ID & operator resolving, etc fully applied (and just *before* final validation).

`ast-orig` Prints out all the original ASTs, before any changes are applied.

`ast-pre-transformed` Prints out ASTs just before the AST transformation passes kick in. Note that "transformation" here refers to a specific pass in the pipeline that's primarily used for Spicy-to-HILTI AST rewriting.

`ast-resolved` Prints out ASTs just after the pass that resolves IDs and operators has concluded. Note that this happens once per round, with progressively more nodes being resolved.

`ast-scopes` Prints out ASTs just after scopes have been built for all nodes, with the output including the scopes. Note that this happens once per round, with progressively more nodes being resolved.

`ast-transformed` Prints out ASTs just after the AST transformation passes kick in. Note that "transformation" here refers to a specific pass in the pipeline that's primarily used for Spicy-to-HILTI AST rewriting.

`ast-codegen` Prints out the ASTs used for C++ code generation. These are the final ASTs with possibly additional global optimizations applied to them.

`compiler` Prints out a various progress updates about the compiler's internal workings. Note that `driver` is often a better high-level starting point.

`driver` Prints out the main high-level steps while going from source code to final compiler output. This stream provides a good high-level overview what's going on, with others going into more detail on specific parts.

`grammar` Prints out the parsing grammars that Spicy's parser generator creates before code generation.

`jit` Prints out details about the JIT process.

`parser` Prints out details about flex/bison processing.

`resolver` Prints out a detailed record of how, and why, IDs and operators are resolved (or not) during AST rewriting.

2.10.4 Benchmarking

End-to-end Parsers

We have a Benchmarking script that builds the HTTP and DNS parsers, runs them on traces both with and without Zeek, and then reports total execution times. The script also compares times against Zeek's standard analyzers. The following summarizes how to use that script.

Preparation

1. You need to build both Spicy and Zeek in release mode (which is the default for both).
2. You need sufficiently large traces of HTTP and DNS traffic and preprocess them into the right format. We normally use Zeek's M57 testsuite traces for this, and have prepared a prebuilt archive of the processed data that you can just download and extract: spicy-benchmark-m57.tar.xz (careful, it's large!).

   To preprocess some other trace `trace.pcap`, do the following:

   - Extract HTTP and DNS sub-traces into `spicy-http.pcap` and `spicy-dns.pcap`, respectively (do not change the file names):

     ```
     # tcpdump -r trace.pcap -w spicy-http.pcap tcp port 80
     # tcpdump -r trace.pcap -w spicy-dns.pcap udp port 53
     ```

   - Run Zeek on these traces with the `record-spicy-batch.zeek` script that comes with Spicy:

     ```
     # zeek -br spicy-http.pcap zeek/scripts/record-spicy-batch.zeek␣
     ↪SpicyBatch::filename=spicy-http.dat
     # zeek -br spicy-dns.pcap  zeek/scripts/record-spicy-batch.zeek␣
     ↪SpicyBatch::filename=spicy-dns.dat
     ```

   - Move traces and resulting data files into a separate directory:

     ```
     # mkdir my-benchmark-data
     # mv spicy-{http,dns}.pcap spicy-{http,dns}.data my-benchmark-data/
     ```

   - Now you can use that `my-benchmark-data/` directory with the Benchmarking script, as shown below.

Execution

1. Use `scripts/run-benchmark` script to build/recompile the parsers. It's easiest to run out of the Spicy build directory (see its usage message for setting paths otherwise). Watch for warnings about accidentally using debug versions of Spicy or Zeek:

   ```
   # cd build
   # ../scripts/run-benchmark build
   ```

   This will put all precompiled code into `./benchmark`.

2. Run the benchmark script with a directory containing your preprocessed data. If you're using the provided M57 archive:

```
# ../scripts/run-benchmark -t /path/to/spicy-benchmark-m57/long run

    http-static  1.58        1.54        1.56
      http-hlto  1.74        1.75        1.75
```

(continues on next page)

(continued from previous page)

```
http-zeek-spicy  4.97        4.87        5.02          conn.log=2752  http.
↪log=4833
  http-zeek-std  3.69        3.59        3.74          conn.log=2752  http.
↪log=4906

     dns-static  0.01        0.01        0.01
       dns-hlto  0.01        0.01        0.01
 dns-zeek-spicy  0.97        0.94        0.97          conn.log=3458   dns.
↪log=3458
   dns-zeek-std  0.80        0.76        0.76          conn.log=3464   dns.
↪log=3829
```

Each line is three executions of the same command. Values are user time in seconds.

The `run-benchmark` script leaves its precompiled code in a subdirectory `./benchmark`. In particular, you will find static binaries there that you can profile. For example, with `perf` on Linux:

```
# perf record  --call-graph dwarf -g ./benchmark/http-opt -U -F spicy-benchmark-m57/
↪long/spicy-http.dat
# perf report -G
```

Microbenchmarks

Todo: Add fiber benchmark.

2.10.5 Style

Todo: This is very preliminary. We'll extend it over time. It's also not consistently applied everywhere yet. Working on that.

Tooling

Spicy ships with a set of linter configurations to enforce some of the style guidelines documented below. We use pre-commit to run linters on each commit.

After cloning the repository, one can install the commit hooks by running the following command from the root of the checkout:

```
$ pre-commit install && pre-commit install --hook-type commit-msg
pre-commit installed at .git/hooks/pre-commit
pre-commit installed at .git/hooks/commit-msg
```

With installed hooks the configured linters check the code after each commit. To run linters standalone one can use the following:

```
$ pre-commit run -a
```

See the pre-commit CLI documentation for more information on how pre-commit can be used.

Note: Some linters might require that a full build was performed or additional external tooling, see e.g., *Formatting*.

Commit Messages

- Provide meaningful commit messages. Start the commit message with a one line summary and then explain at a high-level what's going on, including in particular any new functionality and changes to existing semantics. Include short examples of functionality if possibles. (Expect people to read your commit messages. :)
- If the commit refers to ticket or PR, include the number into the commit message.
- Aim to make commits self-containing chunks of functionality. Rebase and squash before pushing upstream.
- Formatting aspects of commit messages are linted with gitlint via pre-commit hooks, see *Tooling*. In particular we enforce that summary lines start with a capital letter and end in a period, and length limits for both summary and body lines.

Formatting

Spicy comes with a `clang-format` configuration that enforces a canonical style. We require `clang-format` version >= 10 because we need a style option that wasn't available earlier. Formatting is checked by a `clang-format` linter which automatically pulls in a suitable `clang-format` binary, see the *Tooling* section.

Spicy's CI runs `pre-commit run clang-format` as part of its code checks and will abort if there's anything not formatted as expected.

Static analysis

Spicy also comes with a `clang-tidy` configuration, as well as `Makefile` targets similar to the ones for formatting: `make tidy` will check the code, and `make tidy-fixit` will apply fixes automatically where `clang-tidy` is able to. Note that the latter can sometimes make things worse: Double-check `git diff` before committing anything.

You can set the environment variable `CLANG_TIDY` to the full path of the `clang-tidy` to ensure the right version is found (which, similar to `clang-format`, needs to be from Clang >= 10).

Spicy's CI runs `make tidy` as part of its code checks and will abort if there's anything not formatted as expected.

Code Conventions

Identifiers

- Class methods: `lowerCamelCase()` for public and protected methods; `_lowerCamelCase()` for private methods.
- Class member constants & variables: `lower_case` for public members, and `_lower_case_with_leading_underscore` for private members.
- Global function: `lowerCamelCase()`

Comments

- In header files:
  - Public namespace (i.e., anything *not* in `*::detail::*`)
    - Add Doxygen comments to all namespace elements.
    - Add Doxygen comments to all `public` and `protected` members of classes. (Exceptions: Default constructors; destructors; `default` operators; "obvious" operators, such as basic constructors and straight-forward comparisons; obvious getters/setters).
  - Private namespace (i.e., anything in `*::detail::*`)
    - Add a brief sentence or two to all namespace elements that aren't obvious.
    - Add a brief sentence or two to all class members that aren't obvious.
- In implementation files
  - For elements that aren't declared in a separate header file, follow the rules for headers defining elements of the private namespace.
  - Inside methods and functions, comment liberally but not needlessly. Briefly explain the main reasoning behind non-obvious logic, and introduce separate parts inside larger chunks of code.

Doxygen style

- Always start with a brief one-sentence summary in active voice ("Changes X to Y.")
- For functions and methods, include `@param` and `@return` tags even if it seems obvious what's going on. Add `@throws` if the function/method raises an exception in a way that's considered part of its specific semantics.

2.10.6 C++ API documentation

The generated C++ API documentation can be found here.

CHAPTER THREE

INDEX

- genindex
- modindex
- search

INDEX