## Contents

1 Getting in Touch

2 Documentation
   2.1 Installation ......................................................... 5
   2.2 Getting Started .......................................................... 10
   2.3 Frequently Asked Questions .............................................. 16
   2.4 Tutorial: A Real Analyzer .................................................. 17
   2.5 Programming in Spicy ...................................................... 31
   2.6 Toolchain ............................................................... 97
   2.7 Zeek Integration .......................................................... 102
   2.8 Custom Host Applications ............................................... 114
   2.9 Release Notes ............................................................. 122
   2.10 Developer’s Manual ...................................................... 167

3 Index

Index ................................................................. 179
Overview  Spicy is a C++ parser generator that makes it easy to create robust 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 define 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 the system’s C++ compiler), or ahead-of-time either by creating pre-compiled shared libraries or simply by giving you C++ code that you can link into your application.

Spicy comes with a Zeek plugin that enables adding new protocols 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.

See our growing collection of example grammars to get a sense of how Spicy works.

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.
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, in GitHub’s Discussions.
- Subscribe to the Spicy mailing list.
- To follow development, subscribe to the commits mailing list (it can be noisy).
2.1 Installation

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

2.1.1 Pre-built binaries

Linux

We provide pre-built binaries for some platforms from the Git master branch. Artifacts are distributed as TAR archives which can be unpacked to any location:

```
# Unpack archive to e.g., /opt/spicy overwriting any previous installation.
# Writing to /opt/spicy likely requires superuser privileges.
# The path /opt/spicy can be replaced with a custom path.
rm -rf /opt/spicy && mkdir /opt/spicy
    tar xf spicy-linux.tar.gz -C /opt/spicy --strip-components=1
```

The binaries might require installation of additional dependencies; see the Dockerfile for the respective platform.

- alpine-3.12 master, Dockerfile
- centos-7 master, Dockerfile
- centos-8 master, Dockerfile
- debian-9 master, Dockerfile
- debian-10 master, Dockerfile
- fedora-32 master, Dockerfile
- fedora-33 master, Dockerfile
- ubuntu-16 master, Dockerfile
Spicy, Release 0.4.0-branch (afcaa664)

ubuntu-18 master, Dockerfile
ubuntu-20 master, Dockerfile

macOS

Homebrew

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

`brew tap zeek/zeek`

To install Spicy execute:

`brew install --HEAD spicy`

Pre-built binaries

We additionally provide pre-build nightly binaries for MacOS Catalina and MacOS Big Sur. To prevent MacOS from quarantining the files, you should download and unpack via the command line, e.g.:

```
tar xf spicy-darwin.tar.gz
```

For JIT support these binaries require an Xcode installation.

These binaries require Zeek if you want to use the Spicy Zeek plugin:

`brew install zeek # for Zeek integration`

2.1.2 Using Docker

The Spicy distribution comes with a set of Docker files that create images for selected Linux distributions. We walk through how to use these in the following. We also welcome contributions to support more Linux distributions. If you create a new Docker file, please file a pull request.

Pre-requisites

You first need to install Docker on your host system if you haven’t yet.

Linux

All major Linux distributions provide Docker. Install it using your package manager. Alternatively, follow the official instructions.
macOS

Install Docker Desktop for Mac following the official instructions.

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 or Zeek and will cause an error that looks something like this:

```bash
-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.

Using Pre-built Docker Images

We provide a Docker image with Spicy built from trunk.

To run the image execute the following command:

```bash
docker run -it zeekurity/spicy:latest
```

Spicy is installed in `/opt/spicy`.

Build Your Own Spicy Container

You can build your Spicy container from one of the Docker files coming with Spicy: Go into Spicy’s `docker` directory and run `make` to see the container platforms available:

```
# cd spicy
# make
```

Run "make build-<platform>", then "make run-<platform>".

Available platforms:

- alpine-3.12
- centos-8
- ubuntu-18

To build a Spicy container image based on, for example, Ubuntu 18, run:

```bash
# make build-ubuntu-18
```

Once the container build has finished, you can double-check that the container image is now available in your local Docker registry:

```
# docker images | grep -e spicy -e REPO
REPOSITORY TAG IMAGE ID CREATED SIZE
```

(continues on next page)
Great, let’s fire it up!

```
# make run-ubuntu-18
root@bc93113300bc:~# spycy --version
0.2.0-dev
```

## 2.1.3 Building from source

### Prerequisites

Spicy currently supports the following platforms:

- Linux (x86_64)
- MacOS 10.15 / Catalina

Other platforms are unlikely to work at the moment.

**Note:** Earlier versions of macOS aren’t easily supported because of their older C++ standard libraries.

To build Spicy, you will need:

- For compiling the toolchain:
  - A C++ compiler that supports C++17 (known to work are Clang >= 9 and GCC >= 9)
  - CMake >= 3.13
  - Bison >= 3.4
  - Flex >= 2.6
  - Python >= 3.4
  - Zlib (no particular version)

- For integration with Zeek:
  - Zeek >= 3.0

- 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. Additionally, we provide Docker files for building on selected Linux distributions, see Using Docker.

**macOS**

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

If you are using MacPorts:

```
• # port install flex bison cmake ninja python38 py38-pip py38-sphinx
    py38-sphinx_rtd_theme
• # pip install btest
```

If you are using Homebrew:

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

In either case, install Zeek >= 3.0 from source, per the instructions.

**Linux**

On Ubuntu 18 (Bionic):

```
• See the Ubuntu 18 Docker file.
```

On Alpine 3.12:

```
• See the Alpine 3.11 Docker file.
```

On CentOS 8 / RedHat 8:

```
• See the CentOS 8 Docker file.
```

**Installing the Spicy Toolchain**

Get the code:

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

The short version to install Spicy is the standard `./configure && make && make install`. However, you’ll likely need to customize the build a bit, so we’ll walk through some of the options in the following.

Normally, Spicy’s build system will just pick up the system’s standard C++ compiler. If you want to point it to a different compiler, the `configure` script provides an option `--with-cxx-compiler` to do so.

Spicy by default installs into `/usr/local`. You can change that by giving `configure` a `--prefix`:

```
# ./configure --prefix=/opt/spicy
```

If Zeek is installed but not in its standard location (i.e., `/usr/local/zeek`), you can tell `configure` the prefix where to look for it:

```
# ./configure --with-zeek=/opt/zeek
```

The final `configure` output will summarize your build’s configuration. To ensure that Zeek support is enabled, verify the presence of the following line:
Zeek plugin enabled:  yes

Note: configure has a few more flags that may be helpful, see its --help output. For developers, the following may be particular useful:

- --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
- --enable-ccache: use the ccache compiler cache to speed up compilation

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 it, the binaries land in `build/bin`.

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

### 2.1.4 Development setup

In order to speed up precompilation 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>` in 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’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:

```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:
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 (e.g., `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 straightforward: 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 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 spycyc, 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:

```bash
# 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:

```bash
# 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 spycyc -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 spycyc 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:

```bash
# spycyc -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 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:

```bash
# 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
  }
}
```

```bash
# echo "GET /index.html HTTP/1.0" | ./a.out
GET, /index.html, 1.0
```

If you want to see the actual parsing code that Spicy generates, use spycyc again: spycyc -c my-http.spicy will show the C++ code, and spycyc -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 first need to prepare some input, and get Zeek to load the required Spicy plugin. 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 tell Zeek to load a Spicy plugin. If your Spicy build has found Zeek during its configure run, it will have already compiled and installed the plugin into Zeek’s system-wide plugin directory. You can confirm that with `zeek -N`:

```
# zeek -N
<...>
_Zeek::Spicy - Support for Spicy parsers (*.spicy, *.evt) (dynamic, version 0.3.0)
```

As you can see, Zeek now reports the Spicy plugin as available among all the other plugins that it has already built-in. If you don’t see the Spicy plugin in there, the installation might not have had permission to write into the Zeek plugin directory. See Installation for how to point Zeek to the right location manually.

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, ...
6  →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, $\text{conn}$, is a “magic” keyword that lets the Spicy plugin pass the Zeek-side connection ID ($\text{conn_id}$) 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$\text{id}$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

```cpp
#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 Zeek

**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.

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).

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

- Creating a Spicy Grammar
  - Parsing One Packet Type
  - Generalizing to More Packet Types
  - Using Enums
  - Using Unit Parameters
  - Complete Grammar
- Zeek Integration
  - Compiling the Analyzer
  - Activating the Analyzer
  - Defining Events
  - Detour: Zeek vs. TFTP
  - Zeek Script
- Next Steps

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:

<table>
<thead>
<tr>
<th>2 bytes</th>
<th>string</th>
<th>1 byte</th>
<th>string</th>
<th>1 byte</th>
<th>(from RFC 1350)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>Filename</td>
<td>0</td>
<td>Mode</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

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:

```spicy
module TFTP;

public type ReadRequest = unit {
    opcode: uint16;
    filename: bytes &until=b"\x00";
    mode: bytes &until=b"\x00";

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

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`:
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 `printf` 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.txt\000octet\000" [\|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:

```spicy
module TFTP;

public type Packet = unit {
  opcode: uint16;

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

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

```spicy
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:

```spicy
module TFTP;

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

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

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:

```spicy
module TFTP;

public type Packet = unit {
  opcode: uint16;

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

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

```plaintext
# 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:

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

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

    on %done { print self; }
};

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

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):

```plaintext
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:

```plaintext
# 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:

```plaintext
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:

```plaintext
# 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:

```
# # Trivial File Transfer Protocol
# # Specs from https://tools.ietf.org/html/rfc1350
#
module TFTP;
#
# Common header for all messages:
#
# | 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;
  }
  on %done { print self; }
};

# 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)

# Figure 5-1: RRQ/WRQ packet
#
# | Opcode | Filename | 0 | Mode | 0 |
# ----------------------------------------
```
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.

Compiling the Analyzer

While the Spicy plugin for Zeek can compile Spicy code on the fly, it is usually more convenient to compile an analyzer once upfront. Spicy comes with a tool `spicyz` for that. The following command line produces a binary object file `tftp.hlto` containing the executable analyzer code:

```
2.4. Tutorial: A Real Analyzer 25
```
## 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:

```
protocol analyzer spicy::TFTP over UDP:
   parse with TFTP::Packet,
   port 69/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 on the Zeek side.

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

```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:

```zeek
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:

```zeek
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:

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

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:

```
2.4. Tutorial: A Real Analyzer
```
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 extend our Spicy TFTP grammar to call two helper functions that the Spicy plugin makes available: `zeek::confirm_protocol` once we have successfully parsed a request, and `zeek::reject_protocol` in case we encounter a parsing error. While we could put this code right into `tftp.spicy`, we prefer to store it inside separate Spicy file (`zeek_tftp.spicy`) because this is Zeek-specific logic:

```spicy
module Zeek_TFTP;
import zeek;      # Library module provided by the Spicy plugin for Zeek.
import TFTP;

on TFTP::Request::%done {
    zeek::confirm_protocol();
}

on TFTP::Request::%error {
    zeek::reject_protocol("error while parsing TFTP request");
}
```

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:

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

(continues on next page)
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 CH3xFz3U1nYIlDp1Dk 192.168.0.253 50618 192.168.0.10 69 udp ->spicy::tftp - - 0S0 -0 0 D 1 48 0 0 -
1367411052.077243 CfwsLw2TaTie6q9E9g 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);
}
```

2.4. Tutorial: A Real Analyzer
{  
  print "TFTP read request", c$id, filename, mode;  
schedule_tftp_analyzer(c$id);  
}

event tftp::write_request(c: connection, is_orig: bool, filename: string, mode: ...)  
{  
  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 zeek_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\x0a\x0a\x0aN\x0aN\x0aN\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 zeek_tftp.spicy tftp.evt  
# zeek -r tftp_rrq.pcap tftp.hlto tftp.zeek  
# cat tftp.log

(continues on next 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 - - - 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 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:

```spicy
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:

```spicy
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:

```spicy
module Test;

type Foo = unit {
    version: uint32 &default=42;
};
global f: Foo;
print f;
print "version is \%s\n" % 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.

```spicy
module Test;

class Test;

public type Foo = unit {
    version: uint32;
};
```
Let's use `spicy-driver` to parse 4 bytes of input through this unit:

```plaintext
# 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:

```plaintext
module Test;
import spicy;
public type Foo = unit {
    version: uint32 &byte-order=spicy::ByteOrder::Little;
    on %done {
        print "0x%x" % self.version;
    }
};
```

```plaintext
# 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:

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

```spicy
module Test;

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

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.

### Limiting Input 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:

```spicy
module Test;

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

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.
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).

```spicy
module Test;

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

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:

```spicy
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]
```
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:

```spicy
module Test;
import spicy;
public type Foo = unit {
    x: int8 &requires=($$ < 5);
    on %done { print self; }
};
# printf '\001' | spicy-driver foo.spicy
[$x=1]
```

```spicy
# printf '\010' | spicy-driver foo.spicy
[error] terminating with uncaught exception of type spicy::rt::ParseError:
  --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:

```spicy
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]
```

```spicy
# printf '\010' | spicy-driver foo.spicy
[error] terminating with uncaught exception of type spicy::rt::ParseError:
  --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 { ... } Executes when a parse error has been encountered, just before the parser either aborts processing.

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 $$$ identifier. It will also have been assigned to the field already, potentially with any relevant type conversion applied (see On-the-fly Type Conversion with &convert).

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; we discuss these in the corresponding section.

There are three locations where hooks can be implemented:

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

```spicy
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:

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

```spicy
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 instances 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; }
    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.
Unit Parameters

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

```spicy
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);
};
```

```bash
# 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, though the syntax is a bit peculiar:

```spicy
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))[];  # Element constructor must be in "(...)"
};
```

```bash
# 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 s: string)`).

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

```spicy
type Foo = unit {
  ...
  b: Bar(self);
  ...
}(continues on next page)
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;
  ```

- **&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.

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:

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

```plaintext
# printf '1234567890123456' | spicy-driver foo.spicy
```

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 5 as f.x3, respectively:

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

(continues on next page)
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; }
};
```

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.

---

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

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.

Note: &chunked can currently not be combined with &until or &until-including.

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:

```plaintext
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.

```spicy
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:

```spicy
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  Parses elements until one with the value EXPR is encountered. EXPR must be of the same type as the vector’s elements. Once the specified element is encountered, vector parsing stops without including the matching one into the field’s vector value.

&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[]). The one exception that requires special syntax are units with parameters. In that case, one needs to wrap the ELEM_TYPE in additional parentheses, and then add the parameters to it (e.g., x: (MyUnit("arg1"))[]).

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; }
};
```

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 parsing anything. While that’s not very useful for normal fields, it allows branches in switch constructs to forego any parsing.

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.

```spicy
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; }
};
```

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:

```spicy
module Test;

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

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

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:

```plaintext
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:

```plaintext
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; }
};
```

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(); }`).

---

2.5. Programming in Spicy 47
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:

```spicy
module Test;
public type Foo = unit {
    switch {
        -> a: b"A";
        -> b: b"B";
        -> c: b"C";
    }
    on %done { print self; }
};
```

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

```spicy
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; }
};
```

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:

```spicy
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;
};
```

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:
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 spicy::method:unit::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. To enable this functionality, the unit needs to be declared with the %random-access property. 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 {
  %random-access;

  on %init() { self.start = self/input(); }
  
a: A { self.set_input(self.start); }
  b: B;

  on %done() { print \self; }

  var start: iterator<stream>;
};
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;
    
    : 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 to unit instances, with 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:

```spicy
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>]
```

Note: Sinks must be declared public currently. That’s a restriction that we may eventually remove.

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.

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 connected and all will receive the same data).

“MIME type” can mean actual MIME types, such as `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).

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:

```spicy
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; }
};
```

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 is maintaining the state in *global variables*. However, doing so can be cumbersome because a global will often not match semantics well. 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 suitably by the host application running the parser. The host application could, for example, tie the context to the underlying connection.

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

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

By default, each instance of such a unit will then receive a unique context value of that type. The context value can be accessed through the `context()` method, which will return a reference to it:

```
module Test;
public type Foo = unit {
   %context = int64;
   on %init { print self.context(); }
};
```

By itself, this is not very useful. However, host applications may change how contexts are maintained, and they may assign the same context value to multiple units. For example, when parsing a protocol, a host application could create a single context value shared by all top-level units belonging to the same connection.

That’s indeed what both the *Zeek plugin* and the batch mode of *spicy-driver* do, enabling parsers to maintain bi-directional, per-connection state. 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 {
   (continues on next page)
if ( |self.context().pending| ) {
    print "%s -> %s" % (self.context().pending.back(), $$\cdot$.response);
    self.context().pending.pop_back();
} else {
    print "<missing request> -> %s", $$\cdot$.response;
}
}

**Type Request**

```plaintext
{ cmd: /[A-Za-z]+/;
  : b"\n";
}
```

**Type Reply**

```plaintext
{ response: /[0-9]+/;
  : b"\n";
}
```

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.

### 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!";
```

```
# spyc -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 its 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` and `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 typei 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.
```

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";

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

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

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:

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

2.5. Programming in Spicy
This defines a global variable called NAME with type TYPE. If a default is giving, Spicy initialized the global accordingly before any code executes. Otherwise, the global received 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]

**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) → string
Interprets the bytes as representing an binary string encoded with the given character set, and converts it into a UTF8 string.

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) → bytes
Returns a lower-case version of the bytes value, assuming it is encoded in character set charset.

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) → bytes
Returns an upper-case version of the bytes value, assuming it is encoded in character set charset.

**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 < 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 → const 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(uint)` → `enum`

Instantiates an enum instance initialized from an integer value. The value does *not* need to correspond to any of the type’s enumerator labels.

`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 an 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 (#89).

### 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)

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.
```

```
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
```

```
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
Decrement the value, returning the old value.
```

```
uint-- → uint
Decrement the value, returning the old value.
```

```
++int → int
Increment the value, returning the new value.
```

```
++uint → uint
Increment 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.

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

An 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 - 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

• \([E_1, E_2, \ldots, E_N]\) creates a list of \(N\) elements. The values \(E_I\) must all have the same type. \([\,]\) creates an empty list of unknown element type.

• \([\text{EXPR} \text{ for } ID \in \text{ITERABLE}]\) creates a list by evaluating \(\text{EXPR}\) for all elements in \(\text{ITERABLE}\), assembling the individual results into the final list value. The extended form \([\text{EXPR} \text{ for } ID \in \text{SEQUENCE} \text{ if } \text{COND}]\) includes only elements into the result for which \(\text{COND}\) evaluates to \(\text{True}\). Both \(\text{EXPR}\) and \(\text{COND}\) can use \(ID\) to refer to the current element.

• \(\text{list}(E_1, E_2, \ldots, E_N)\) is the same as \([E_1, E_2, \ldots, E_N]\), and \(\text{list}()\) is the same as \([\,]\).

• \(\text{list}<T>(E_1, E_2, \ldots, E_N)\) creates a list of type \(T\), initializing it with the \(N\) elements \(E_I\).

Operators

\begin{verbatim}
begin(<container>) \rightarrow <iterator>
\end{verbatim}
Returns an iterator to the beginning of the container’s content.

\begin{verbatim}
end(<container>) \rightarrow <iterator>
\end{verbatim}
Returns an iterator to the end of the container’s content.

\begin{verbatim}
list == list \rightarrow bool
\end{verbatim}
Compares two lists element-wise.

\begin{verbatim}
|list| \rightarrow \text{uint}<64>
\end{verbatim}
Returns the number of elements a list contains.

\begin{verbatim}
list !\neq list \rightarrow bool
\end{verbatim}
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

• \(\text{map}<K, V>\) specifies a map with key type \(K\) and value type \(V\).

• \(\text{iterator}<\text{map}<K, V>>\)

Constants

• \(\text{map}(K_1: V_1, K_2: V_2, \ldots, K_N: V_N)\) 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. \(\text{map}()\) 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.

• \(\text{map}<K, V>(K_1: V_1, K_2: V_2, \ldots, K_N: V_N)\) creates a map of type \(\text{map}<K, V>\), initializing it with the given key/value pairs. \(\text{map}<K, V>()\) 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[element] → void
   Removes an element from the map.

del(map) → void
   Removes the map.

del(map, key) → void
   Removes the element with the given key from the map.

del(map, key, default) → <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.

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.

map[<any>] → <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| → 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 == 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

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 \(\rightarrow\) real
  Inverts the sign of the real.
real + real \(\rightarrow\) real
  Returns the sum of the reals.
real += real \(\rightarrow\) real
  Adds the first real to the second, assigning the new value.
real != real \(\rightarrow\) bool
  Compares the two reals.

**Regular Expression**

Spicy provides POSIX-style regular expressions.

**Type**

- regexp

**Constants**

- `/Foo*ba?r/, /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) \(\rightarrow\) tuple<int32>, 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) \(\rightarrow\) int32**
  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) \rightarrow 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() \rightarrow 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() \rightarrow void`

Removes all elements from the set.

Operators

`add set[element] \rightarrow void`

Adds an element to the set.

`begin(<container>) \rightarrow <iterator>`

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

`delete set[element] \rightarrow void`

Removes an element from the set.

`end(<container>) \rightarrow <iterator>`

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

`set == set \rightarrow bool`

Compares two sets element-wise.
<any> in set → bool
    Returns true if an element is part of the set.

|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(inout 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(inout 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

\texttt{at(i: \text{uint}<64>) \rightarrow \text{iterator<stream>}}

Returns an iterator representing the offset \( i \) inside the stream value.

\texttt{freeze() \rightarrow \text{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.

\texttt{is\_frozen() \rightarrow \text{bool}}

Returns true if the stream value has been frozen.

\texttt{trim(inout i: \text{iterator<stream>}) \rightarrow \text{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.

\texttt{unfreeze() \rightarrow \text{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

\texttt{begin(<container>) \rightarrow \text{iterator}}

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

\texttt{end(<container>) \rightarrow \text{iterator}}

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

\texttt{|stream| \rightarrow \text{uint}<64>}

Returns the number of stream the value contains.

\texttt{stream += \text{bytes} \rightarrow \text{stream}}

Concatenates data to the stream.

\texttt{stream += \text{view<stream>} \rightarrow \text{stream}}

Concatenates another stream’s view to the target stream.

\texttt{stream != \text{stream} \rightarrow \text{bool}}

Compares two stream values lexicographically.

Iterator Methods

\texttt{is\_frozen() \rightarrow \text{bool}}

Returns whether the stream value that the iterator refers to has been frozen.

\texttt{offset() \rightarrow \text{uint}<64>}

Returns the offset of the byte that the iterator refers to relative to the beginning of the underlying stream value.

Iterator Operators

\texttt{*\text{iterator<stream>} \rightarrow \text{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.

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 occurance. 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.

|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.

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.

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 - 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 \leq time \rightarrow \text{bool} \\
\text{Compares the times.}

time + interval \rightarrow time^{\text{(commutative)}} \\
\text{Adds the interval to the time.}

time \neq time \rightarrow \text{bool} \\
\text{Compares two time values.}

**Tuple**

Tuples are heterogeneous containers of a fixed, ordered set of types.

**Type**

- \text{tuple<TYPE}_1\ldots\text{TYPE}_N

**Constants**

- (1, "string", True), (1, ), ()
- \text{tuple}(1, "string", True), \text{tuple}(1), \text{tuple}()

**Operators**

- \text{tuple} == \text{tuple} \rightarrow \text{bool} \\
\text{Compares two tuples element-wise.}
- \text{tuple}[\text{uint<64>}] \rightarrow \text{<type of element>} \\
\text{Extracts the tuple element at the given index. The index must be a constant unsigned integer.}
- \text{tuple} . \text{<id>} \rightarrow \text{<type of element>} \\
\text{Extracts the tuple element corresponding to the given ID.}
- \text{tuple} \neq \text{tuple} \rightarrow \text{bool} \\
\text{Compares two tuples element-wise.}

**Unit**

**Type**

- \text{unit} \{ FIELD_1; \ldots; FIELD_N \}
- \textbf{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 list expression: \text{my_unit} = \{\$FIELD_1 = X_1, \$FIELD_N = X_N, \ldots\} where \$FIELD_I is the label of a corresponding field in \text{my_unit}’s type.
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.

custom() → <context>&
Returns a reference to the %context instance associated with the unit.

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.

Usage of this method requires the unit to be declared with the %random-access property.

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.

Usage of this method requires the unit to be declared with the %random-access property.

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.

Usage of this method requires the unit to be declared with the %random-access property.

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.

Usage of this method requires the unit to be declared with the %random-access property.

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 yet 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 yet have a value assigned, it returns its \&default expression if that has been defined. Otherwise it triggers an exception, unless used in a context that specifically allows for that situation (such as, inside the Zeek plugin’s evt files).

unset unit.<field> → void
Resets a field back to its original uninitialized state.

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.

dend(<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.

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");

for ( i in v )
    print i;

# spycyc -j for.spicy
1
2

(continues on next page)
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::Module`):

```
module Test;

print "Hello, %s!" % "World";
print "%s=%d" % ("x", 1);

# spycyc -j print.spicy
Hello, World!
x=1
```
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 a 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

Todo: This isn’t available in Spicy yet (#89).

throw EXPR;

throw; # only inside "catch"
Throws an exception, letting execution stop at the current location and propagating \texttt{EXPR} up the call stack to the nearest matching \texttt{try/catch}. If not match is found, the exception eventually bubbles up to the host application. If that doesn’t catch it either, the runtime will abort execution. \texttt{EXPR} must be of type \texttt{Exception}. \texttt{EXPR} can be skipped to rethrow the current exception inside a \texttt{catch} block.

\textbf{try/catch}

\begin{verbatim}
try
  BLOCK
catch [(TYPE IDENT)]
  BLOCK
...
catch [(TYPE IDENT)]
  BLOCK
\end{verbatim}

Catches any exception thrown in the \texttt{try} block that match one of the types in any of \texttt{catch} headers, which must be \texttt{Exception} types. A \texttt{catch} without a type matches any exception. If no \texttt{catch} matches an exception thrown in the \texttt{try} block, it’ll be propagated further up the stack. A bare \texttt{throw} statement can be used inside a \texttt{catch} block to rethrow the current exception.

\textbf{while}

\begin{verbatim}
while ( COND )
  BLOCK
while ( local IDENT = EXPR; COND )
  BLOCK
\end{verbatim}

\texttt{while} introduces a loop that executes \texttt{BLOCK} for as long as the boolean \texttt{COND} evaluates to true. The second form initializes a new local variable \texttt{IDENT} with \texttt{EXPR}, and makes it available inside both \texttt{COND} and \texttt{BLOCK}.

\textbf{Error Handling}

\begin{verbatim}
Todo:  This isn’t available in Spicy yet (#89).
\end{verbatim}

\textbf{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 \texttt{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 \texttt{ParsingError} exceptions can be caught indirectly through the \texttt{%on_error} unit hook, which internally is nothing else than an exception handler.

\begin{verbatim}
Exception
\end{verbatim}
Todo: Support for catching other exception throughs `try/catch` needs to be added still (#89).

```result<T> / error
```

Todo: Spicy doesn’t have `result/error` yet (#90).

Error recovery

Todo: The earlier Spicy prototype had support for resynchronizing parsers with their input stream after parse error. It’s on the list to bring that back (#23).

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 |
| begin |
| bitfield |
| bool |
| break |
| bytes |
| case |
| cast |
| catch |
| const |
| const_iterator |
| constant |
| continue |
| cregexp |
| cstring |
| default |
| delete |
| else |
| end |
| enum |

(continues on next page)
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
port
print
priority
private
property
public
real
regexp
return
set
sink
stop
stream
string
struct
switch
time
time_ns
timer
try
tuple
type
uint16
uint32
uint64
uint8
2.5.3 Library

Module spicy

Types

spicy::AddressFamily

Specifies an address’ IP family.

type AddressFamily = {
    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 = {
    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 = {
    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::MatchState

Captures state for incremental regular expression matching.

spicy::Protocol

Specifies a transport-layer protocol.

```cpp
type Protocol = {
    TCP,
    UDP,
    ICMP
};
```

spicy::RealType

Specifies the type of a real value.

```cpp
type RealType = {
    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.

```cpp
type ReassemblerPolicy = {
    First # take the original data & discard the new data
};
```

spicy::Side

Specifies a side an operation should operate on.

```cpp
type Side = {
    Left,   # operate on left side
    Right,  # operate on right side
    Both    # operate on both sides
};
```

spicy::ZlibStream

Captures the state of gzip decompression the corresponding library functions.

Functions
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::current_time() : time

Returns the current wall clock time.

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.

2.5. Programming in Spicy
function spicy::strptime(buf: string, format: string) : time

Parse time from string.
This function uses the currently active locale and timezone to parse values.
The format string can contain format specifiers supported by POSIX strftime, 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.

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 the Spicy distribution.

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.
  • Spicy grammar
  • Spicy code for Zeek integration
  • 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.
  • Spicy grammar
• Spicy code for Zeek integration
• 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.

• Spicy grammar
• Spicy code for Zeek integration
• 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.

• Spicy grammar
• Spicy code for Zeek integration
• 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):

```spicy
module Test;

public type test = unit {  
a: /1234/ %debug {  
    print self.a;
  }  

b: /567890/;  

  on b %debug { print self.b; }
};
```

2.5. Programming in Spicy
```
# 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
    Request::RequestLine
    Request::method = GET
    Request::anon_2 =
    Request::uri = /index.html
    Request::anon_3 =
    Request::Request::Version
    Request::anon = HTTP/
    Request::number = 1.0
    Request::version = "$anon=b"HTTP/", $number=b"1.0"
    Request::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.

### 2.6 Toolchain

**2.6.1 spacy-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 spacy-config**

spicy-config reports information about Spicy’s build & installation options.

```
Usage: spacy-config [options]
```

Available options:

- `--build` Prints "debug" or "release", depending on the build configuration.

(continues on next page)
## 2.6.3 spycy

spycy compiles Spicy code into C++ output, optionally also executing it directly through JIT.

Usage: spycy [options] <inputs>

Options controlling code generation:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-c</td>
<td>--output-c++</td>
</tr>
<tr>
<td>-d</td>
<td>--debug</td>
</tr>
<tr>
<td>-e</td>
<td>--output-all-dependencies</td>
</tr>
<tr>
<td>-j</td>
<td>--jit-code</td>
</tr>
<tr>
<td>-1</td>
<td>--output-linker</td>
</tr>
<tr>
<td>-o</td>
<td>--output-to &lt;path&gt;</td>
</tr>
<tr>
<td>-p</td>
<td>--output-hilti</td>
</tr>
<tr>
<td>-v</td>
<td>--version</td>
</tr>
<tr>
<td>-A</td>
<td>--abort-on-exceptions</td>
</tr>
<tr>
<td>-B</td>
<td>--show-backtraces</td>
</tr>
</tbody>
</table>

(continues on next page)
-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 | --version Print version information.
-X | --debug-addl <addl> Implies -d and adds selected additional
→instrumentation (comma-separated; see 'help' for list).
-Q | --include-offsets Include stream offsets of parsed data in output.

Inputs can be .hlt, .spicy, .cc/.cxx, *.hlto.

## 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 needed 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:

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPICY_PATH</td>
<td>Colon-separated list of directories to search for modules. In contrast to --library-paths using this flag overwrites built-in paths.</td>
</tr>
</tbody>
</table>

Inputs can be .hlt, .spicy, .cc/.cxx, *.o, *.hlto.

**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
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
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
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
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
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 needed 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.

(continues on next page)
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, Spicy ships with a Zeek plugin that makes its parsers accessible to Zeek’s processing pipeline. In the following, we dig deeper into how to use all of this.

2.7.1 Terminology

In Zeek, the term “analyzer” refers generally to a component that processes a particular protocol (“protocol analyzer”) or file format (“file analyzer”). “Processing” here is more than just parsing content: An analyzer controls when it wants to be used (e.g., with connection 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 provides the remaining pieces to Zeek, turning a Spicy parser into a full Zeek analyzer. That’s what we refer to as “Spicy (protocol or file) analyzer” for Zeek.

2.7.2 Installation

To use the Spicy plugin with Zeek, the main preparation task is making sure Zeek knows where to find it. If you performed a standard Spicy installation through `make install`, that will normally have linked the plugin right into Zeek’s system-wide plugin directory so that it becomes directly available. You can check the output of `zeek --N` to confirm:

```
# zeek --N _Zeek::Spicy
_Zeek::Spicy - Support for Spicy parsers (*.spicy, *.evt, *.hlto) (dynamic, version 0.3.0)
```

If the symlink didn’t get installed (e.g., because you’re lacking permission to write into Zeek’s plugin’s directory), you can alternatively set Zeek’s `ZEEK_PLUGIN_PATH` manually to the directory where the installation placed the plugin.

```
spicy-config --zeek-plugin-path knows where that is:
```

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` `*.hlto`.
If you want to move the plugin to other systems, you can find a binary distribution inside your build directory at `zeek/plugin/Zeek_Spicy.tgz`. On the target system, the plugin will not require a Spicy installation to operate as long as you are not using just-in-time compilation.

**Note:** Developer’s note: You can also point `ZEEK_PLUGIN_PATH` to the Spicy build directory, without installing it first. The plugin get built into `<build-directory>/zeek/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 the Spicy plugin 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 both protocol 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:

```plaintext
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`.

### 2.7. Zeek Integration
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`). 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 just one property currently:

   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.

As a full example, here's what a new analyzer could look like:

   packet analyzer spicy::RawLayer: parse with Raw Layer::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. You will need to use the PacketAnalyzer::try_register_packet_analyzer_by_name for registering Spicy analyzers (not register_packet Analyzer), with the name of the new Spicy analyzer being ANALYZER_NAME. zeek -NV shows the names of existing analyzers. For example:

```cpp
event zeek_init()
{
  if ( ! PacketAnalyzer::try_register_packet_analyzer_by_name("Ethernet", 0x88b5, ...
                   "spicy::RawLayer") )
    Reporter::error("cannot register Spicy analyzer");
}
```

**File Analyzer**

Defining file analyzers works quite similar to protocol analyzers, through *.evt sections like this:

```cpp
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 a 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.
As a full example, here’s what a new GIF analyzer could look like:

```text
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:

```text
on HOOK_ID -> event EVENT_NAME(ARG1, ..., ARG_N);
```

```text
on HOOK_ID if COND -> event EVENT_NAME(ARG1, ..., 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 a `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 event you want to generate, preferably including a namespace (e.g., `http::request`).

- **ARG_I** An argument to pass to the event, given as an arbitrary Spicy expression. The 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 expressions 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 parameters. Most Spicy types translate over pretty naturally, the following summarizes the translation:
Table 1: Type Conversion from Spicy to Zeek

<table>
<thead>
<tr>
<th>Spicy Type</th>
<th>Zeek Type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>addr</td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td>bool</td>
<td></td>
</tr>
<tr>
<td>enum { ... }</td>
<td>enum { ... }</td>
<td>[1]</td>
</tr>
<tr>
<td>int(8</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>list&lt;T&gt;</td>
<td>vector of T</td>
<td></td>
</tr>
<tr>
<td>map&lt;V,K&gt;</td>
<td>table[V] of K</td>
<td></td>
</tr>
<tr>
<td>optional&lt;T&gt;</td>
<td>T</td>
<td>[2]</td>
</tr>
<tr>
<td>port</td>
<td>port</td>
<td></td>
</tr>
<tr>
<td>real</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>set&lt;T&gt;</td>
<td>set[T]</td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>tuple&lt;T_1, ..., T_N&gt;</td>
<td>record { T1, ..., T_N }</td>
<td>[3]</td>
</tr>
<tr>
<td>uint(8</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>vector&lt;T&gt;</td>
<td>vector of T</td>
<td></td>
</tr>
</tbody>
</table>

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.

$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.

Enum Types

The Zeek plugin automatically makes Spicy enum types available on the Zeek-side if you declare them public. For example, assume the following Spicy declaration:

```spicy
module Test;
public type MyEnum = enum {
    A = 83,
    B = 84,
    C = 85
};
```

The plugin will then create the equivalent of the following Zeek type for use in your scripts:

```zeek
module Test;
export {
    type MyEnum: enum {
        MyEnum_A = 83,
        MyEnum_B = 84,
        MyEnum_A = 85,
        MyEnum_Undef = -1
    };
}
```

(The odd naming is due to ID limitations on the Zeek side.)

You can also see the type in the output of `zeek -NN`:

```
[...]
_Zeek::Spicy - Support for Spicy parsers
    [Type] Test::MyEnum
[...]
```

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 a basic form of conditional compilation through @if COND/@else/@endif blocks, similar to a C preprocessor. For now, this supports only a single type of condition that allows comparing against the Zeek version the plugin is being used with, available inside COND as ZEEK_VERSION. For example:

```zeek
@if ZEEK_VERSION < 30200
  <EVT code for Zeek versions older than 3.2>
@else
  <EVT code for Zeek version 3.2 or newer>
@endif
```

Supported comparisons in COND are ==, !=, <, <=, >, >=. Note that COND does not support more complex expressions (e.g., no nested expressions or parentheses).

2.7.4 Compiling Analyzers

Once you have the *.spicy and *.evt source files for your new analyzer, you have two options to compile them, either in advance, or just-in-time at startup.

Ahead Of Time Compilation

You can precompile analyzers into *.hlto object files containing their final executable code. 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
```

While this approach requires an additional step every time something changes, starting up Zeek now executes quickly. 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. Alternatively, you can override that location by setting the environment variable SPICY_MODULE_PATH to a set of colon-separated directories to search for precompiled analyzers.

Run spicyz -h to see some additional options it provides, which are similar to spicy-driver.

Just In Time Compilation

To compile analyzers on the fly, you can pass your *.spicy and *.evt files to Zeek just like any of its scripts, either on the command-line or through @load statements. The Spicy plugin hooks into Zeek’s processing of input files and diverts them the right way into its compilation pipeline.

This approach can be quite convenient, in particular during development of new analyzers as it makes it easy to iterate—just restart Zeek to pick up any changes. The disadvantage is that compiling Spicy parsers takes a noticeable amount of time, which you’ll incur every time Zeek starts up; and it makes setting compiler options more difficult (see below). We generally recommend using ahead-of-time compilation when working with the Zeek plugin.

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()

Triggers a DPD protocol confirmation for the current connection.

function zeek::reject_protocol(reason: string)

Triggers a DPD protocol violation for the current connection.

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::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::file_begin(mime_type: optional<string> = Null)

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.

function zeek::fuid() : string

Returns the current file’s FUID.

function zeek::file_set_size(size: uint64)

Signals the expected size of a file to Zeek’s file analysis.

function zeek::file_data_in(data: bytes)

Passes file content on to Zeek’s file analysis.

function zeek::file_data_in_at_offset(data: bytes, offset: uint64)

Passes file content at a specific offset on to Zeek’s file analysis.
function zeek::file_gap(offset: uint64, len: uint64)

Signals a gap in a file to Zeek’s file analysis.

function zeek::file_end()

Signals the end of a file to Zeek’s file analysis.

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.

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 zeek::confirm_protocol() (see Controlling Zeek from Spicy) from a hook inside your grammar at a point when the parser can be reasonably certain that it is processing the expected protocol. Option ally, you may also call zeek::reject_protocol() when you’re sure the parser is not parsing the right protocol (e.g., inside an %error hook). Doing so will let Zeek stop feeding it more data.

2.7.7 Configuration

The Spicy plugin provides a set of script-level options to tune its behavior, similar to what the spicy-driver provides as command-line arguments:

```zeek
module Spicy;

export {
    # Activate compile-time debugging output for given debug streams (comma-separated list).
    const codegen_debug = "" &rdef;

    # Enable debug mode for code generation.
    const debug = F &rdef;

    # If debug is true, add selected additional instrumentation (comma-separated list).
    const debug_addl = "" &rdef;

    # Save all generated code into files on disk.
    const dump_code = F &rdef;

    # Enable optimization for code generation.
    const optimize = F &rdef;
}
```

(continues on next page)
# Report a break-down of compiler's execution time.
const report_times = F &redef;

# Disable code validation.
const skip_validation = F &redef;

# Show output of Spicy print statements.
const enable_print = F &redef;

# abort() instead of throwing HILTI # exceptions.
const abort_on_exceptions = F &redef;

# Include backtraces when reporting unhandled exceptions.
const show_backtraces = F &redef;

Note, however, that most of those options affect code generation. It’s usually easier to set them through `spicyz` when precompiling an analyzer. If you are using Zeek itself to compile an analyzer just-in-time, keep in mind that any code generation options need to be in effect at the time the Spicy plugin kicks off the compilation process. A `redef` from another script should work fine, as scripts are fully processed before compilation starts. However, changing values from the command-line (via Zeek’s `var=value`) won’t be processed in time due to intricacies of Zeek’s timing. To make it easier to change an option from the command-line, the Spicy plugin also supports an environment variable `SPICY_PLUGIN_OPTIONS` that accepts a subset of `spicy-driver` command-line options in the form of a string. For example, to JIT a debug version of all analyzers, set `SPICY_PLUGIN_OPTIONS=-d`. The full set of options is this:

Supported Zeek-side Spicy options:

- `-A` When executing compiled code, `abort()` instead of throwing HILTI→exceptions.
- `-B` Include backtraces when reporting unhandled exceptions.
- `-C` Dump all generated code to disk for debugging.
- `-d` Include debug instrumentation into generated code.
- `-D <streams>` Activate compile-time debugging output for given debug streams→(comma-separated).
- `-O` Build optimized release version of generated code.
- `-o <out.hlto>` Save precompiled code into file and exit.
- `-R` Report a break-down of compiler's execution time.
- `-V` Don't validate ASTs (for debugging only).
- `-X <addl>` Implies -d and adds selected additional instrumentation (comma-separated).

To get that usage message, set `SPICY_PLUGIN_OPTIONS=-h` when running Zeek.

## 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 be loading the right `*.spicy` or `*.evt` files. 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 EVT file "/Users/robin/work/spicy/master/doc/examples/my-http.evt"
[debug/zeek] Loading events from /Users/robin/work/spicy/master/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 event 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] 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]
[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
```

(continues on next page)
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
    method = GET
    anon_2 =
    uri = /index.html
    anon_3 =
    MyHTTP::Version
    anon = HTTP/
    number = 1.0
    version = [$number=b"1.0"]
    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
    c: connection = [id=[orig_h=127.0.0.1, orig_p=59619/tcp, ...
        ...
        [1] method: string = GET
        [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 anything 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 5: my-http.spicy

```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;
  }
};
```

First, we’ll use spycy to generate a C++ parser from the Spicy source code:

```
# spycy -c my-http.spicy -o my-http.cc
```

Option `-c` (aka `--output-c++`) tells spycy that we want it to generate C++ code (rather than compiling everything down into executable code).

We also need spycy to get generate some additional additional “linker” code implementing internal plumbing necessary for cross-module functionality. That’s what `-l` (aka `--output-linker`) does:
# spycyc -l my-http.cc -o my-http-linker.cc

We’ll compile this linker code along with the `my-http.cc`.

Next, spycyc 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:

# spycyc -P 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
    { ...
        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 ...
    -> 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::Resumable;
    extern auto parse3(spicy::rt::ParsedUnit& gunit, hilti::rt::ValueReference ...
    -> 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. parse1 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 6: my-http-host.cc

```cpp
#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:

```bash
# 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
' 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 7: my-http-host.cc

```cpp
#include <iostream>
#include <hilti/rt/libhilti.h>
#include <spicy/rt/libspicy.h>
#include "my-http.h"

int main(int argc, char** argv) {

    // 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::parse2(stream, {}, {});

    // Wrap up runtime library.
    hilti::rt::done();

    return 0;
}

(continues on next page)

## 2.8. Custom Host Applications
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.
hl::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
GET, /index.html, 1.0
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 RequestLine 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 8: 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 9: my-http-callback.cc

#include <iostream>

(continues on next page)
#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 my-http.spicy -o my-http.cc
# spicyc -l my-http.cc -o my-http-linker.cc
# spicyc -P my-http.spicy -o my-http.h
→$(spicy-config --cxxflags --ldflags)
# ./my-http "GET index.html HTTP/1.0"
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 A.spicy -o A.cc
# spicyc -c B.spicy -o B.cc
# spicyc -c C.spicy -o C.cc
# spicyc -l 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 10: my-http-host.cc

```cpp
#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;
}
```

Using the name of the parser (MyHTTP::RequestLine) we can instantiate it from C++, and then feed it data:

```cpp
// 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);
```

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
' 
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 11: my-driver
```

```c
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]);
```

(continues on next page)
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 content 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 $(spicy-config --cxxflags --ldflags
−−dynamic-loading)
# spycyc -j my-http.spicy >my-http.hlto
# ./my-driver my-http.hlto "$(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 incre-
mentally 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

- Change Log
- Migrating from the old prototype
2.9.1 Change Log

0.4.0-1455 | 2021-02-18 17:05:01 +0100
* Document `&byte-order` unit attribute. (Benjamin Bannier, Corelight)
* Document `&requires` unit attribute. (Benjamin Bannier, Corelight)
* Convert %byte-order unit property to an attribute. (Benjamin Bannier, Corelight)
* Convert %requires property to an attribute. (Benjamin Bannier, Corelight)
* Make tuple index type calculation more resilient. (Benjamin Bannier, Corelight)
* GH-744: Fix out-of-bounds access in tuple index type resolution.
   This code was previously indexing into a vector beyond the end when
determining the type of a tuple index operation. We now
return an unknown type here so that we can reject the operation
during later validation.

0.4.0-1446 | 2021-02-18 09:31:13 +0000
* GH-709: Fix codegen's result type of unit.offset(). (Robin Sommer,
  Corelight)
* GH-709: Add description of enum use to Zeek plugin documentation.
  (Robin Sommer, Corelight)
* Fix bug with custom enum types not being available in time for
  Zeek scripts. Turns out that for precompiled Zeek analyzers it
didn't work to automatically make the Spicy-side enum types
available to Zeek scripts because of a timing issue. (Robin
Sommer, Corelight)
* Add grouping() function to builder API. (Robin Sommer, Corelight)
* Add new function linkage "preinit" to HILTI. (Robin Sommer,
  Corelight)

0.4.0-1437 | 2021-02-17 11:08:16 +0100
* GH-682: Remove TODO in docs. (Benjamin Bannier, Corelight)

0.4.0-1435 | 2021-02-17 11:07:03 +0100
* GH-723: Provide constructor for ports.
  Closes #723. (Benjamin Bannier, Corelight)

0.4.0-1430 | 2021-02-17 11:06:27 +0100
* GH-731: Add unit `position` method.
  This method returns an iterator to the current position in the unit's
  input stream.

(continues on next page)
Closes #731. (Benjamin Bannier, Corelight)

0.4.0-1428 | 2021-02-16 16:04:42 +0000

* GH-690: Add `%finally` hook to Spicy units.

``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.)

* Generalize `%error` to catch all exceptions. So far it was catching
  only HILI-side errors. Now we do general catch-all so have a
  reliably hook for error handling no matter what's the cause.
  (Robin Sommer, Corelight)

* Support finalizers in HILTI structs. There's a now new magic hook
  `~finally` that will run automatically when a struct instance gets
  destroyed. (Robin Sommer, Corelight)

* Fix double normalization in ID strings. (Robin Sommer, Corelight)

0.4.0-1422 | 2021-02-16 10:49:34 +0000

* GH-696: Add support for shared unit contexts with lifetime
determined by host application. (Robin Sommer, Corelight)

This introduces a new unit property for public unit types:

```
%context = <type>;
```

The unit's new `context()` method will then return a value of type
`<type>&` (i.e., a reference to an instance of the given type). This will
be initialized with a reference to a default-constructed instance of the
type.

What makes this special is that the host application can determine how a
context instance is being shared across different parsers: it can (1)
let multiples units link their context to the same instance, and (2)
retain a context instance beyond any specific unit's lifetime.

We use this new mechanism to provide contexts with connection-level
semantics: the Zeek plugin, and also spicy-driver's batch mode, will
create a single context instance that the two originator-side and
responder-side units will share (they must declare matching contexts
for that to work). The object will live until the connections gets
torn down (also meaning it'll be reused across the per-packet parsers
for UDP).

See the documentation for examples.

* Add `'<vector>.pop_back'`. (Robin Sommer, Corelight)
* Support assignment to tuple elements. (Robin Sommer, Corelight)

* Extend the spicy-driver batch format to be able to express bi-direction connection semantics. This increases the batch format version to v2. (Robin Sommer, Corelight)

* Add new variant of external function linkage to HILTI that skips the resumable wrapping. (Robin Sommer, Corelight)

0.4.0-1415 | 2021-02-16 10:29:53 +0000

* GH-726: $$ wasn’t accessible from inside an `&until-including` expression. (Robin Sommer, Corelight)

* GH-740: May address the unstable behavior by rewriting the code surrounding the `&while` parsing (but keeping semantics the same). (Robin Sommer, Corelight)

0.4.0-1412 | 2021-02-16 10:07:45 +0000

* Activate CI for Debian 10 on master. (Robin Sommer, Corelight)

* GH-629: Use linb::any instead of std::any across the code base. (Robin Sommer, Corelight)

* GH-738: Fix crash in spicy-driver when used with units that require parameters. (Robin Sommer, Corelight)

* GH-737: Fix non-converging AST in case of ambiguous ID usage. (Robin Sommer, Corelight)

0.4.0-1403 | 2021-02-09 13:23:55 +0000

* Fix missing f-string in Spicy Pygments lexer for floats/reals (Jon Siwek, Corelight)

* GH-722: Bump pathfind. (Benjamin Bannier, Corelight)

* GH-732: Add test for setting input beyond current position. (Benjamin Bannier, Corelight)

* Add missing dependency of header precompilation target on script. (Benjamin Bannier, Corelight)

* Fix documentation of `set_input`. (Benjamin Bannier, Corelight)

0.4.0-1394 | 2021-02-05 07:09:51 +0000

* Fix compatibility with Zeek 4.1 API changes/removals. (Jon Siwek, Corelight)

0.4.0-1392 | 2021-02-04 15:14:31 +0000

* GH-962: Fix type inference for constructors of vector/list/set/map, and for locals/globals. (Benjamin Bannier, Corelight; Robin Sommer, Corelight)
* Remove unused function typeOfExpressions. (Benjamin Bannier, Corelight)

0.4.0-1385 | 2021-02-04 14:59:06 +0000

* Add Pygments lexers for `.spicy` and `.evt` files. (Jon Siwek, Corelight)

* Use syntax highlighting for Spicy/EVT/Zeek/C++ in documentation. (Jon Siwek, Corelight)

* Add 'livehtml' target to `doc/Makefile` that will run `sphinx-autobuild`, plus some cleanup. (Jon Siwek, Corelight)

0.4.0-1375 | 2021-02-03 11:43:19 +0100

* Enhance parsing of single-element container constructors so that trailing `,` is optional. (Benjamin Bannier, Corelight)

* GH-711: Fix formatting of a number of predefined enum values. (Benjamin Bannier, Corelight)

* GH-697: Allow trailing ',' in enum declarations. (Benjamin Bannier, Corelight)

* Update some links in the documentsations. (Benjamin Bannier, Corelight)

* GH-699: Add script to store and upload release artifacts. (Benjamin Bannier, Corelight)

0.4.0-1358 | 2021-02-01 10:40:23 +0100

* GH-701: Make offsets usable from unit field attributes.

  We previously updated parse positions when entering productions. This is insufficient as the evaluation of attributes might already access the position via `self.offset()`.

  With this patch we now only update the parse position before we start parsing of a field.

  Closes #701. (Benjamin Bannier, Corelight)

* Use ubuntu-20 as base for published Docker image. (Benjamin Bannier, Corelight)

* Set PATH and ZEEK_PLUGIN_PATH for ubuntu Docker images. (Benjamin Bannier, Corelight)

* Validate Zeek installation for Docker CI tasks. (Benjamin Bannier, Corelight)

* Fix Docker images for possible lib libdir.

  This patch fixes the `ZEEK_PLUGIN_PATH` set in Docker images so Zeek can find plugins even if the Spicy installation was performed with a libdir named `lib` instead of `lib64`. (Benjamin Bannier, Corelight)
0.4.0-1346 | 2021-01-29 08:26:26 +0000

* GH-685: Allow to define a minimum Spicy version that a module requires. Adding `%spicy-version = "1.0";` to a module now lets it require to compile with a Spicy of at least that version. The version can be specified as either `x.y` or `x.y.z`. (Robin Sommer, Corelight)

* GH-688: Add functions `zeek::uid()` and `zeek::fuid()` to access UIDs and FUIDs, respectively. (Robin Sommer, Corelight)

* GH-689: Allow to pass a MIME type to `file_new()` when beginning file analysis. (Johanna Amann, Corelight)

* GH-700: Support `&size` attribute at the unit level.

  This limits input to the given number of bytes for the unit:

  ```
  type Bar = unit(n: uint64) {
    x: bytes &eod;
  } &size=n;
  ```

  Parsing less than `n` bytes is an error. (Robin Sommer, Corelight)

* Fix --git-root in autogen-version call. (Robin Sommer, Corelight)

* Fix using `Null` as a parameter default value for `optional<T>`. (Robin Sommer, Corelight)

* Add timestamp removal to the Zeek log canonification in tests. (Johanna Amann, Corelight)

0.4.0-1330 | 2021-01-28 10:22:55 +0100

* GH-701: Add strptime Spicy library function.

0.4.0-1324 | 2021-01-26 17:59:30 +0000

* Update reproc to release version. (Robin Sommer, Corelight)

0.4.0-1321 | 2021-01-26 13:23:11 +0100

* Lots of CI / clang-format / Docker cleanup. (Benjamin Bannier, Corelight)

0.4.0-1305 | 2021-01-21 15:34:39 +0000

* Fix failing Zeek tests. (Robin Sommer, Corelight)

0.4.0-1304 | 2021-01-21 13:37:38 +0000

* Fix a couple of issues with Zeek 4.0. (Robin Sommer, Corelight)

* GH-640: Extend spicy-doc to document begin/end operators. (Robin Sommer, Corelight)

* Bulk update of all test baselines. (Robin Sommer, Corelight)
* Update copyright headers. (Robin Sommer, Corelight)
* Do not use #pragma once in libhilti.h. (Robin Sommer, Corelight)
* Disable output from header compilation. (Robin Sommer, Corelight)
* Update README. (Robin Sommer, Corelight)
* Remove unused TLS baselines. (Robin Sommer, Corelight)
* Prettify configure output. (Robin Sommer, Corelight)

0.4.0-1286 | 2021-01-21 12:04:58 +0100
* Add CI job to validate release tarball. (Benjamin Bannier, Corelight)
* Document more platforms in the installation docs. (Benjamin Bannier, Corelight)
* Use "default" compiler if possible in Docker images. (Benjamin Bannier, Corelight)
* Make master-only CI tasks triggerable. (Benjamin Bannier, Corelight)

0.4.0-1277 | 2021-01-20 19:55:47 +0000
* Add final newline to ASCII output of spicy-dump. (Robin Sommer, Corelight)
* GH-671: Fix parsing of empty look-ahead vectors. (Robin Sommer, Corelight)
* Fix Zeek plugin build when toolchain is disabled. (Robin Sommer, Corelight)
* Let the autogen-version script execute out of the top-level directory. (Robin Sommer, Corelight)

0.4.0-1270 | 2021-01-20 14:04:16 +0100
* Allow to enforce sequential execution of jobs spawned during JIT. If the environment variable 'HILTI_JIT_SEQUENTIAL' is set jobs are executed sequentially. (Benjamin Bannier, Corelight)
* Use sequential JIT mode in CI to save memory. (Benjamin Bannier, Corelight)
* GH-582: Add master CI task for macOS Big Sur. (Benjamin Bannier, Corelight)
* GH-670: Remove pre-commit hooks which require external tool installation. (Benjamin Bannier, Corelight)
* Move `spicy::rt::getenv` to `hilti::rt`. (Benjamin Bannier,
Corelight

0.4.0-1262 | 2021-01-19 08:28:41 +0000

* macOS fixes (Robin Sommer, Corelight)

  - GH-664: Increase minimum fiber stack size.
  - Pass CMAKE_OSX_SYSROOT through to JIT linker.

* Add a couple cron scripts to repository. (Benjamin Bannier, Corelight)

  - `ci/cron/spicy-benchmark.sh:` run Spicy benchmarks
  - `ci/cron/update_spicy_dockerhub.sh: update Spicy Docker images

0.4.0-1255 | 2021-01-18 08:45:26 +0000

* Port JIT to use host C++ compiler instead of libclang/libllvm.
  (Robin Sommer, Benjamin Bannier; Corelight).

This completely removes the dependency on an existing clang installation. Instead, JIT now requires only a C++17 host compiler. Internally, this comes with a substantial refactor of the JIT components.

This changes also includes the following additional pieces:

  - We pull in `https://github.com/DaanDeMeyer/reproc` as a submodule for spawning the host compiler.

  - The environment variable `HILTI_CXX` overrides the executable to use as the C++ compiler during JIT. It won't work to switch in a completely different compiler that way (because we'd continue to make some assumptions based on the default compiler), but it allows to provide an additional wrapper around the default compiler.

  - We remove the distinction between building the toolchain with/without JIT. We now always assume we have the JIT capability available, but will fail gracefully when unable to use the host compiler.

  - JIT compilation captures & reports C++ error output.

* Extend `(hilti,spicy)-config` with options `--cxx-flags-hlto`/`--cxx-flags-hlto` printing compiler flags for building HLTO files.

* CMake: Redoing object library linking. We weren't consistent in how we linked them, which led to some off effects. (Robin Sommer, Corelight)

* Codegen: Construct initializer lists from literals. Benjamin Bannier, Corelight)

* Cleanup left-over ASAN code that was no longer in use. (Robin Sommer, Corelight)

* Increase CI memory limit for Ubuntu debug task. (Robin Sommer, Corelight)
* Bump CMake in CI Docker image. (Benjamin Bannier, Corelight)

0.4.0-1227 | 2021-01-13 07:41:35 +0000

* Re-implement the runtime's fiber (co-routine) code. (Benjamin Bannier, Robin Sommer; Corelight).

This should be mostly transparent to users, but internally the main pieces are:


- GH-561: Support "shared stack" mode for fibers where the runtime only allocates a single stack that's used by all fibers, vs. individual per-fiber stacks. This comes with a slight performance penalty for context switching, but substantially improves memory usage (and, in turn, performance with lots of concurrent fibers). The shared stack is used by default now. There's also new benchmark hilti-rt-fiber-benchmark to profile performance.

- GH-460: Add runtime checks for risk of stack overflow. When the HILTI codegen creates a function, it now always inserts a stack check at the beginning that ensures that a minimal amount of stack space remains available. If it is not, we abort with a runtime exception. This gives us well-defined behavior if call stacks go too deep. The minimum amount of stack space is configurable through a global option.

Along with this comes large refactor of the runtime's fiber code.

* Fix Linux' --ldflags for pthread. (Robin Sommer, Corelight)

* Make hilti::rt::isInitialized() save to be used when no global state is available. (Robin Sommer, Corelight)

* When running the CI container through the Makefile, use local user's UID/GID. (Robin Sommer, Corelight)

* Adding missing call to rt::done() to default main(). (Robin Sommer, Corelight)

* Fix CMake logic for hilti-rt. (Robin Sommer, Corelight)

* Enable leak checking by default in ASAN builds. ASAN itself enables it only on Linux by default. (Robin Sommer, Corelight)

* Let autogen-version find VERSION independent of current directory. (Robin Sommer, Corelight)

0.4.0-1201 | 2021-01-11 21:03:10 +0100

* Activate building Spicy in CentOS 7 Docker image. (Benjamin Bannier, Corelight)
0.4.0-1199 | 2021-01-10 14:03:48 +0000

* Remove manual installation of Zeek CMake files on FreeBSD CI. (Benjamin Bannier, Corelight)

0.4.0-1196 | 2021-01-09 19:13:50 +0100

* GH-650: Add infrastructure to upload a release tarball for tags. (Benjamin Bannier, Corelight)
* GH-633: Fixes for FreeBSD, plus CI support. (Benjamin Bannier, Corelight)
* Bump minimal version of BTest in CI to 0.66. (Benjamin Bannier, Corelight)

0.4.0-1175 | 2021-01-06 07:28:23 +0000

* Fix for macOS 10.16. (Robin Sommer, Corelight)

0.4.0-1173 | 2021-01-05 11:01:34 +0100

* GH-644: Remove Spicy support for `else` on `while`.

0.4.0-1148 | 2021-01-04 08:10:25 +0000

* Overhauling regexp matching semantics.

We were inconsistent in how regular expressions (1) handled being anchored vs not, and (2) allowed to retrieve the matching data. We fix this as follows:

- We rename the regexp methods for more consistency: `match` now means "matching is anchored at the beginning of the data", and `find` means we find matches anywhere in the data. This leads to the following renames:

  * `<regexp>.find` -> `<regexp>.match`
  * `<regexp>.find_groups` -> `<regexp>match_groups`
  * `<regexp>.find_span` -> `<regexp>.find`

  `<bytes>.match` adapts to the new semantics as well.

- We remove the `&anchor` attribute because that's the default now.

- `&no_sub` remains available to override the default choice of the matcher type: it'll use the minimal matcher and disable capture group support.

0.4.0-1145 | 2021-01-02 20:43:29 +0100

* GH-635: Collection of cleanups from consuming Spicy in external project. (Benjamin Bannier)
* GH-620: Add paths to Spicy and Zeek to Ubuntu Docker image PATHs. (Benjamin Bannier, Corelight)
* Extend Docker images. (Benjamin Bannier, Corelight)
- Add Debian-10 Docker image.
- Add Debian-9 Docker image.
- Add Fedora-32 Docker image.
- Add Fedora-33 Docker image.
- Add Ubuntu-16 Docker image.
- Add Ubuntu-18 Docker image.
- Add Ubuntu-20 Docker image.
- Remove unsupported Ubuntu-19.10.
- Fix yum repository name in CentOS 8 Docker images. (GH-619)
- Use correct download location for Zeek packages.

* GH-575: Add support for FreeBSD. (Benjamin Bannier, Corelight)

* Fix test 'spicy.rt.time'. (Benjamin Bannier, Corelight)

* C++ fix to be explicit on which variant we construct in `Result`. (Benjamin Bannier, Corelight)

0.4.0-1118 | 2020-12-11 07:18:52 +0000

* GH-602: Install Zeek from packages. (Benjamin Bannier, Corelight)

0.4.0-1112 | 2020-12-10 10:33:31 +0000

* GH-25: Add support for manual backtracking to Spicy. If a field is marked with `&try`, a later call to `self.backtrack()` anywhere down in the parse tree will return to that position and continue there. See the documentation for more, including an example. (Robin Sommer, Corelight).

* Support &eod for vector parsing. (Robin Sommer, Corelight)

* Fix Sphinx helper script that emits command lines. (Robin Sommer, Corelight)

0.4.0-1106 | 2020-12-10 10:10:16 +0000

* Make capture groups available inside hooks of regular expression fields. (Robin Sommer, Corelight)

  Example:

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

  This will print the pieces corresponding to the three groups inside the expression.

  Couple notes:

  - We don't make $0 available as that would be the same as `$$`.

  - Matching with capture groups is more expensive that without. If you don't use any groups in the regexp, the overhead is automatically removed. If you use groups, but don't need them to
* Make new captures() method available on regexp matching state in HILTI. (Robin Sommer, Corelight)

* Fix argument mismatches between Spicy generated C++ prototypes and Zeek plugin runtime code. (Robin Sommer, Corelight)

* Fix compile error on non-JIT version of plugin, and fix unsafe use of string_view. (Robin Sommer, Corelight)

* Fix warnings reported by Apple's compiler. (Robin Sommer, Corelight)

0.4.0-1093 | 2020-12-10 08:39:26 +0100

* Switch runtime library to use "fiber" library instead of libtask. We now leverage https://github.com/simonfxr/fiber. (Benjamin Bannier, Corelight)

* GH-599: Fix construction of `Bytes`. (Benjamin Bannier, Corelight)

* GH-603: Disable precompiled headers if we are building with GCC. (Benjamin Bannier, Corelight)

* GH-94: Explicitly enable leak checking on tests relying on it being active. (Benjamin Bannier, Corelight)

* GH-492: Add unit tests for IntrusivePtr. (Benjamin Bannier, Corelight)

* Make run-zeek script also work with ASAN on macOS. (Benjamin Bannier, Corelight)

* Remove unused Zeek compiler driver variable. (Benjamin Bannier, Corelight)

* Add missing runtime cleanups in a few places. (Benjamin Bannier, Corelight)

* Fix some documentation links and link checking. (Benjamin Bannier)

* Fix GitHub link. (Robin Sommer, Corelight)

0.4.0-1071 | 2020-11-30 18:16:35 +0000

* Series of speed-ups to the Spicy compiler. (Robin Sommer, Corelight)

  - Add caching of derived information to some AST nodes.
  - Use optional_ref<T> for returning references of AST node attributes instead of copies. (Robin Sommer, Corelight)
  - Cache location instances.
  - Returning const references for lots of accessor methods. (Robin Sommer, Corelight)
  - Mark two AST node classes as singletons.
  - Switch AST infrastructure from shared_ptr to IntrusivePtr.
  - Use forward declaration for nlohmann::json where we can.
* Relabel profiling collector names in JIT code. (Robin Sommer, Corelight)

* Fix for type erasure profiling. (Robin Sommer, Corelight)

0.4.0-1057 | 2020-11-23 08:35:04 +0000

* GH-584: Add new vector methods resize/sub/at. (Robin Sommer, Corelight)

* Revert accidental unbumping of tinyformat. (Benjamin Bannier, Corelight)

0.4.0-1053 | 2020-11-18 08:33:32 +0000

* GH-568: Change parsing of look-ahead vectors to use a loop instead of recursion. (Robin Sommer, Corelight)

* GH-583: Fix `foreach` inside a unit hook. The `foreach` attribute wasn't passed on, and the `$$` lookup didn't work inside a non-inline hook. (Robin Sommer, Corelight)

* GH-574: Implement and document hook priorities. Robin Sommer, Corelight)

* GH-389: Add additional tuple constructor expression (`tuple(...)`). (Robin Sommer, Corelight)

* GH-544: Let Zeek plugin search for HLTO files at startup. By default, the plugin now loads all *.hlto files found in `$prefix/lib/spicy/ZeeK_Spicy/modules/`. This can be changed by setting the environment variable SPICY_MODULE_PATH to a colon-separated list of directories to search. (Robin Sommer, Corelight)

* Force upgrade to well-defined BTest version in our Docker containers. (Robin Sommer, Corelight)

* Forward Zeek plugin's debug messages to Zeek's debug logger. That means that `zeek -B plugin-ZeeK-Spicy` will send them into Zeek's debug.log. (Robin Sommer, Corelight)

0.4.0-1031 | 2020-11-17 14:05:32 +0100

* GH-571: Fix padding handling when formatting nan and inf values.

  This patch bumps tinyformat to a version containing the upstream fix for https://github.com/c42f/tinyformat/issues/76 (and the follow up fix https://github.com/c42f/tinyformat/pull/80).

  We also add a regression test.

  Closes #571. (Benjamin Bannier, Corelight)

* GH-589: Fix "'_debug' was not declared in this scope" build error (Robin Sommer, Corelight)

* Mention header precompilation in installation docs. (Benjamin Bannier, Corelight)
* Disable precompiled headers for gcc CI job.

While mixing GCC and Clang with precompiled headers is an issue for JIT (JIT always uses Clang to compile modules while precompiled headers are created with the compiler used to compile Spicy itself), we disable precompiled headers for the non-JIT, GCC job since GCC emits a non-suppressible warning, see

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=89808,

```bash
  build/hilti/runtime/include/hilti/rt/libhilti.h:3:9: error: #pragma once in
  
  #pragma once
  ^~~~

  cc1plus: all warnings being treated as errors

```

Instead of not treating warnings as errors anymore we disable precompiled headers in this build since it does not use them anyway. (Benjamin Bannier, Corelight)

* Add missing include. (Benjamin Bannier, Corelight)

* Optionally precompile libhilti.h.

This patch adds precompilation of libhilti.h for developer setups. This header is included in all files during JIT, and precompiling it can for individual files reduce JIT time by up to 30%. This behavior can be disabled by setting the CMake option `HILTI_DEV_PRECOMPILE_HEADERS=OFF`.

Specifically, we add a cmake target which creates debug and release versions of a precompiled libhilti.h. We modify HILTI's built-in CXXFLAGS to check for the presence of the needed precompiled header in the expected location. We expect the header either in a specific location in the build directory, or optionally in a version directory in `$HOME/.cache/spicy`.

In this patch we enable use of the precompiled header only during JIT, but e.g., not when building Spicy targets. This allows us to profit from faster JIT while still making efficient use of e.g., ccache. Additionally, this approach allows us to punt on fixing a number of issues in header files which get exposed during JIT (JIT artifacts are not affected at the moment).

The original goal was to precompile libhilti.h before installation which is not reached. In general, it seems that precompiled headers are not intended to be moved after creation. This poses a challenge when the header gets installed: while we could in principle generate the precompiled header at the final location during install time, this seems not reliably possible when generating packages which can be expanded in arbitrary locations.

Additionally, we also install a script for users to take advantage of precompiled headers once Spicy has been installed. (Benjamin Bannier, Corelight)

0.4.0-1020 | 2020-11-12 11:04:57 +0000

* Fix performance regression by removing wasteful formatting of debug message
in release mode in Spicy runtime driver. (Robin Sommer, Corelight)

* Fix fiber result handling on abort. (Robin Sommer, Corelight)

* GH-578: Fix glib decompression. Robin Sommer, Corelight)

* Add helper tool 'spicy-batch-extract' that extracts a single session out of a Spicy batch file. (Robin Sommer, Corelight)

* Add preliminary benchmark script `scripts/run-benchmarks`. See developer docs for more. (Robin Sommer, Corelight)

0.4.0-1011 | 2020-11-05 17:09:53 +0100

* GH-451: Support building on centos7 (Benjamin Bannier, Corelight)

0.4.0-1006 | 2020-11-05 12:56:56 +0000

* GH-497: Add support for packet analyzers to Zeek plugin. (Robin Sommer, Corelight)

Zeek's master branch received support for adding low-level "packet analyzers" through plugins. This extends the Spicy plugin to support that. The documentation gained a new section explaining how to sue that.

Note: This feature will not "officially" be supported until Zeek 4.0 comes out (because we don't have Zeek master in CI).

* Add --zeek-version argument to spicy-config. (Robin Sommer, Corelight)

* Compile fix for GCC on Ubuntu. (Robin Sommer, Corelight)

0.4.0-1001 | 2020-11-05 08:47:23 +0000

* Use stable names for hook implementations, plus a text fix.

0.4.0-999 | 2020-11-04 15:18:45 +0100

* Use a lightweight lambda class to callbacks executing on fibers. (Benjamin Bannier, Corelight)

* Require all callbacks executing on fibers to return something. (Benjamin Bannier, Corelight)

* Remove unused type_info::overload. (Benjamin Bannier, Corelight)

* GH-546: Model `TypeInfo` with a union instead of a `std::variant`. (Benjamin Bannier, Corelight)

0.4.0-994 | 2020-11-03 09:22:40 +0000

* Update test baseline. (Robin Sommer, Corelight)

0.4.0-993 | 2020-11-02 16:09:13 +0000

* GH-159: Add some missing operator<< to runtime library. This now lets "-X trace" work for Spicy parsers. (Robin Sommer, Corelight)
* GH-531: Make generated grammar symbols globally unique during Spicy codegen. (Robin Sommer, Corelight)

* GH-176: Avoid infinite recursion when printing ASTs with self-recursive types. (Robin Sommer, Corelight)

* Fix premature library shutdown in spicy-dump, which could lead to crashes during cleanup of the parsed unit. (Robin Sommer, Corelight)

* Fix error message in validator. (Robin Sommer, Corelight)

* Aggressively compress package artifact for static build. (Benjamin Bannier, Corelight)

* Remove packaging tasks for static CI build. Static builds tend to produce very large binaries which cannot reliably be compressed to be inside Cirrus CI's 1GB artifact size limit. (Benjamin Bannier, Corelight)

* Fix documentation source link. (Benjamin Bannier, Corelight)

* Consistently use `hilti::rt::filesystem` instead of `std::filesystem`. (Benjamin Bannier, Corelight)

0.4.0-979 | 2020-10-23 14:11:27 +0000

* GH-95: Zeek plugin: Respect %port and %mime-type properties in units. (Robin Sommer, Corelight)

* GH-513: Catch if an interval's number of seconds is out of range. (Robin Sommer, Corelight)

* Clean up parsers' bounds checking of signed integer values. (Robin Sommer, Corelight)

* GH-515: Change `{time, interval}::nanoseconds()` to return integers instead of real. (Robin Sommer, Corelight)

* Fix bashism in test helper script. (Robin Sommer, Corelight)

* GH-490, GH-507: Provide error message when attempting to have spicy-driver compile inputs when JIT is not available. (Robin Sommer, Corelight)

* Reject regexp unit fields that aren't constant. (Robin Sommer, Corelight)

* GH-524: Fix spicy-driver usage message. (Robin Sommer, Corelight)

* GH-526: Show executable's name in usage/error messages of binaries replying on the spicy-driver host code. (Robin Sommer, Corelight)

* GH-527: Updating usage output in docs. (Robin Sommer, Corelight)

* GH-528: Clarify spicyc's error message on needing HLTO filenames with -o. (Robin Sommer, Corelight)
* GH-529: Clarifying libhilti's usage() fallback message. (Robin Sommer, Corelight)

* GH-540: Fix bug in regular expression parsing. (Robin Sommer, Corelight)

* GH-270: Fix for unit filters. (Robin Sommer, Corelight)

* GH-500: Remove --include-linker option from driver. The better way to do all this is using "-l" to generate *just* the linker code, then compile the two together as two compilation units. (Robin Sommer, Corelight)

* GH-499: Provide has_label() method for enum values. Returns true if the enum maps to a known type value other than Undef. (Robin Sommer, Corelight)

* GH-556: Fix coercion of optional. (Robin Sommer, Corelight)

* GH-481: Correctly set locale in centos8 image. (Benjamin Bannier, Corelight)

* Use working and stable link in host application documentation. (Benjamin Bannier, Corelight)

* Fix documentation to reference latest support alpine version. (Benjamin Bannier, Corelight)

* GH-546: Use plain function pointers for type info accessors. (Benjamin Bannier, Corelight)

0.4.0-946 | 2020-10-23 09:35:04 +0200

* GH-522: Switch filesystem impl over to ghc::filesystem.

Closes #522. (Benjamin Bannier, Corelight)

* Fix use of uninitialized value when numbering unit fields.

When were using a non-const member function to call a delegated base class constructor. Since at the point where the member function was called the derived class was not fully constructed, the members it accessed and mutated were uninitialized, we were reading from an unset variable, attempting to set it, and then overwrote its values when the member initializer was triggered (after the base class was constructed). This lead to unit field indices set to large value which were not usable down the line.

We now delegate maintaining the field counter to a mixin class which can be constructed before the base class needing its state. (Benjamin Bannier, Corelight)

0.4.0-939 | 2020-10-21 14:35:30 +0200

* GH-521: Do not create copies when loading HLTO libraries. (Benjamin Bannier, Corelight)
0.4.0-935 | 2020-10-12 07:13:46 +0000

* Add support for new %print hook to Spicy units. (Robin Sommer, Corelight)

  This allows a unit to customize its appearance when printed (or more generally: rendered into a string).

  Example:
  # cat test.spicy
  public type Test = unit {
    on % init { print self; }
    f1: uint32;
    f2: uint8;
    on % done { print self; }
    on % print {
      local f1 = ";
      local f2 = ";
      if ( self?.f1 )
        f1 = "%s" % self.f1;
      if ( self?.f2 )
        f2 = "%s" % self.f2;
      return "%s|%s" % (f1, f2);
    }
    );
  }
  # printf '\x00\x00\x00\x01\x02' | spicy - driver test.spicy
  -1-1

* Allow customizing the string rendering of HILTI structs. If a struct provide a hook __str__(), that will be used instead of the built-in rendering of all its fields. (Robin Sommer, Corelight)

* Make ID getter method constant. (Robin Sommer, Corelight)

0.4.0-930 | 2020-10-09 18:10:13 +0200

* Change `Stream::data` and `stream::View::data` to return a `Bytes` instead of a `string`. (Benjamin Bannier, Corelight)

* Test & CI updates. (Benjamin Bannier, Corelight)
  - Add a CI master step to validate Homebrew formula works.
  - Add missing Cirrus CI instance to Homebrew task.
  - Build compiler toolchain by default when configuring directly from CMake.
  - Bump default Zeek package in Docker images to more recent version.
  - Fix removal of build directory in CentOS 8 Docker container.
  - Increase sleep amount in test to make time measurable in CI.
  - Speed up test `spicy.types.unit.trimming`.

(continues on next page)
* Help clang-tidy understand control flow in a couple of places.
  (Benjamin Bannier, Corelight)

0.4.0-918 | 2020-10-06 12:23:08 +0200

* Introduce the notion of Spicy-specific compiler options.

So far, we only had codegen options for the HILTI compiler. This change allows Spicy to track options as well that control specifics of how the Spicy-to-HILTI code generations operates. Includes a couple of things:

- As there's no appropriate global state on the Spicy side where we could store options easily, we extend the HILTI-side options struct with the notion of "auxiliary options", described by key/value pairs. Spicy can then store its options in there.

- We derived a separate spicy::Driver from hilti::Driver to customize behaviour.

- We introduce hooks into the hilti::Driver to add support for additional command line options. This remains a bit ad-hoc for now. At some point, we'll clean up of command-line handling generally works across the various tools, we have quite a bit of code duplication there right now.

- This introduces a Spicy-side `-Q` option to record stream offsets during parsing. Right now, the option isn't having any effect yet, the implementation is going to come in a subsequent commit. spicy-dump's previous `-o` is renamed to `-Q` as well. (Robin Sommer, Corelight)

* Add an option to emit data offsets from spicy-dump.

  This patch adds an option `-o` to `spicy-dump` which can be used to trigger outputting of the stream offsets where unit fields were parsed from. (Benjamin Bannier, Corelight)

* Make parser struct `__offset` member accessible from Values. (Benjamin Bannier, Corelight)

* Store field offsets during parsing. (Benjamin Bannier, Corelight)

* Make unit fields store their unit-wide field counter.

  Initially the parser will create unit fields as `UnresolvedField`'s. This patch adds an `__index` field and accessor for this field which will be used to number all unit fields in the final, fully constructed and resolved unit so offsets can be computed. (Benjamin Bannier, Corelight)

* Store whether a struct field is internal on type level. (Benjamin Bannier, Corelight)

* Fix `type::struct::Field::is*` methods to return booleans.
These functions were previously returning optional field values which carry no information beyond being set or not. This patch changes these methods to return booleans instead. (Benjamin Bannier, Corelight)

* Fully type constructions of empty optional values. (Benjamin Bannier, Corelight)
* Make `rt::Map::contains` `const`. (Benjamin Bannier, Corelight)

0.4.0-906 | 2020-10-06 08:11:53 +0200
* GH-511: Fix order of flags in spicyc usage message.
* GH-512: Fix typo in error message.

0.4.0-903 | 2020-09-28 18:42:38 +0000
* Reorganize source code to allow for building the runtime library standalone. This adds a corresponding configure option: 
  `--build-toolchain=yes/no` (default: yes)
* Add configure option `--disable-zeek-plugin-install` to disable installing the Zeek plugin into Zeek's plugin path. (Robin Sommer, Corelight)
* Add CMake module that enables Zeek plugins to build Spicy parsers.
* Rename Zeek plugin to "_Zeek::Spicy" to work-around a loading order issue in Zeek. (Robin Sommer, Corelight)
* Turn all 3rdparty code into proper git submodules. (Robin Sommer, Corelight)
* Make using Gold linker default in CMake config. This is no change from when using "configure", but it changes the default when running "cmake" directly. (Robin Sommer, Corelight)
* Fix configuration of Gold linker to apply to cmake modules as well. (Robin Sommer, Corelight)
* Extend hilti-config and spicy-config with options to print the runtime's C++ include directories. (Robin Sommer, Corelight)
* Avoid needing "which" in configure. (Robin Sommer, Corelight)
* Disable a couple unit tests when running as root. (Robin Sommer, Corelight)
* Fix compiler warning. (Robin Sommer, Corelight)

0.4.0-890 | 2020-09-28 18:41:08 +0000
* Port Zeek plugin to current Zeek master. (Robin Sommer, Corelight)
* GH-508: Prevent loss of precision when codgening Time constructors. (Benjamin Bannier, Corelight)
* GH-503: Fix constructor `interval_na` to accept unsigneds like...
documented. (Benjamin Bannier, Corelight)

* Include `netinet/in.h` for `in[6]_addr` in `rt/address.h`. 
  (Benjamin Bannier, Corelight)

* Fix whitespace in spicy-driver batch usage string. (Benjamin Bannier, Corelight)

0.4.0-884 | 2020-09-08 07:39:00 +0000

* Reimplement HILTI's stream type to substantially improve performance. (Robin Sommer, Corelight)

* Move resolving of exceptions' backtrace symbols to point-of-use for improving performance. (Robin Sommer, Corelight)

* Fix problem with fiber reuse. (Robin Sommer, Corelight)

* Fix a unit test that's using multiple filters. (Robin Sommer, Corelight)

* Add macOS work-around for problem with doctests' platform detection. (Robin Sommer, Corelight)

* When configured for debug mode, build tests without -O and link them against (hilti,spicy)-rt-debug. (Robin Sommer, Corelight)

0.4.0-874 | 2020-08-27 14:21:57 +0200

* Update parser lookup tables exclusively from `spicy::rt::init`. 
  We previously were updating `globalState()->parsers_by_mime_type` from `registerParser` while `globalState()->parsers_by_name` was updated by `init`. This patch makes `init` the only function updating the parser lookup tables `parsers_by_name` and `parsers_by_mime_type`. (Benjamin Bannier, Corelight)

* Remove unintentional `return` in `rt::haveEod`. (Benjamin Bannier, Corelight)

* Avoid creating vain global state in `spicy::rt::{done, isInitialized }`. (Benjamin Bannier, Corelight)

* Reinitialize default parser when reinitializing spicy::rt library. (Benjamin Bannier, Corelight)

0.4.0-861 | 2020-08-24 13:04:33 +0200

* Diagnose invalid `zlib::Stream`'s. 
  When a `zlib::Stream` enters a failed state it cannot be reused anymore. Previously we would return empty data whenever such a stream was used again nevertheless. Since empty data is a valid result of decompressing non-empty input this leaves users with no good way to catch such scenarios after the fact.

  This patch modifies `zlib::Stream::decompress` to now throw an exception.

(continues on next page)
instead of returning valid but empty data. (Benjamin Bannier, Corelight)

0.4.0-855 | 2020-08-24 10:56:17 +0000
* Fix Cirrus CI file for Alpine 3.12. (Robin Sommer, Corelight)

0.4.0-854 | 2020-08-24 08:41:23 +0000
* Switch Dockerfile for Alpine from 3.11 to 3.12.
* Compile code with debug information also when in release mode. (Robin Sommer, Corelight)
* Compile the toolchain with -O3 instead of -O2. (Robin Sommer, Corelight)
* Always link against thread library. The performance of some of our runtime code depends on whether the executable is linked against pthread or not. (Robin Sommer, Corelight)

0.4.0-849 | 2020-08-24 08:39:14 +0000
* Fiber runtime library updates. (Robin Sommer, Corelight)
  - Fix state tracking in fibers. When taking a fiber from the cache, we would reset its state to INIT, which then led to restarting it from scratch instead of reusing the already running loop.
  - Add a function to fill the fiber cache with preinitialized instances.
  - Track number of fiber initialization in statistics.
  - Unify fiber debug logging.

0.4.0-843 | 2020-08-21 13:10:42 +0200
* Consistently deal with `std::filesystem` still being experimental in hilti-rt. (Benjamin Bannier, Corelight)

0.4.0-840 | 2020-08-21 10:04:02 +0200
* Prevent use after free in `rt::base64::finish`. (Benjamin Bannier, Corelight)

0.4.0-833 | 2020-08-19 11:06:44 +0200
* Add assertion catching out-of-bounds HILTI module access. (Benjamin Bannier, Corelight)
* Add check to prevent nullptr dereference in `hilti::rt::done`. (Benjamin Bannier, Corelight)
* Add missing context cleanup in `hilti::rt::done`.
  We configure a `Context` owned by `GlobalState` in `hilti::rt::init`.
  Since we destruct that context in `hilti::rt::done` we also need to
reset the context as we would otherwise leave a dangling reference behind. (Benjamin Bannier, Corelight)

* Call out gitlint in commit message style guidelines. (Benjamin Bannier, Corelight)

* Remove guidance on not rebasing development branches. (Benjamin Bannier, Corelight)

* Add developer manual section on pre-commit linters. (Benjamin Bannier, Corelight)

* Add pre-commit linter for commit messages. (Benjamin Bannier, Corelight)

0.4.0-819 | 2020-08-19 11:05:55 +0200

* Explicitly set locale in centos Docker container.

`hilti::rt::init` needs to be able to set the locale on the host system and will abort with a fatal error if it cannot do so. This patch makes sure this is possible in the centos-8 Docker container provided by us.

Note: This already worked in BTests since its config file enforces the needed environment variables. This patch fixes it also for standalone usage of Spicy (and now e.g., the `locale` command emits no more errors). (Benjamin Bannier, Corelight)

0.4.0-817 | 2020-08-18 11:24:52 +0200

* Add `hilti::rt::Library::symbol`.

This patch adds a new member function `Library::symbol` which can be used to obtained a symbol from a specific library.

While currently `Library` users just call `::dlsym` globally to obtain a symbol, this becomes ambiguous if multiple libraries in the process image provide similarly named symbols. Instead they could hold handles to all interesting libraries and obtain symbols with the added function. With that we could also `::dclose` libraries in the `Library` destructor; we'll leave that change for a later point. (Benjamin Bannier, Corelight)

* Fix warning potentially emitted from `rt::Library` destructor. (Benjamin Bannier, Corelight)

* Fix format string in `rt::createTemporaryFile`. (Benjamin Bannier, Corelight)

* Extend error message when `rt::Library::save` fails. (Benjamin Bannier, Corelight)

* Change CI job to use non-priviledged user.

At least on ubuntu `root` seems to be able to modify even read-only directories which makes it hard to write tests checking handling of permission errors.

Change the image used in CI to use a non-priviledged user instead. (Benjamin Bannier, Corelight)

* Change directory used for ccache store in CI.
We were originally using `/var/spool/ccache` which could require extended privileges to create. Switch over to use `/tmp/ccache` which should be easier to create even for unprivileged users. (Benjamin Bannier, Corelight)

* Prevent `rt::Resumable::get` from leaking non-HILTI exceptions. (Benjamin Bannier, Corelight)

* Make `hilti::rt::Resumable`'s operator bool explicit. (Benjamin Bannier, Corelight)

* Consistently provide stringification for all exceptions. (Benjamin Bannier, Corelight)

* Add comparison operators to `rt::Backtrace`. (Benjamin Bannier, Corelight)

* Fix `to_string_for_print` for `DeferredExpression`. Since `DeferredExpression` is a template itself we cannot specialize the master template of the `detail` namespace; instead we need to provide an overload. (Benjamin Bannier, Corelight)

0.4.0-798 | 2020-08-14 07:25:00 +0000

* CI: Move Zeek package specs back into tasks to work around Cirrus issue, per https://github.com/cirruslabs/cirrus-ci-docs/issues/692. (Robin Sommer, Corelight)

0.4.0-796 | 2020-08-14 07:20:31 +0000

* Embed version information into HLTO files that encodes information about the HILTI version/config they were built with. (Robin Sommer, Corelight)

We use this for detecting (1) mismatches in the HILTI version used for building vs running, and (2) running an optimized HLTO file with a debug runtime. In both cases, the runtime will print a warning to stderr, but still proceed processing the library.

Note that this is breaking change that will prevent HLTOs built with earlier versions from loading now.

* Add option `-U` to tools that summarizes runtime resource consumption at termination. The measurement excludes time spent for startup/termination. (Robin Sommer, Corelight)

* Use `cout/cerr in debug logger instead of reopening them separately. Output can get mixed up otherwise if other places write to `cout/cerr` directly. (Robin Sommer, Corelight)

* Update nlohmann-json and move to top-level 3rdparty directory. (Robin Sommer, Corelight)

0.4.0-789 | 2020-08-11 07:47:30 +0000

* Lots of new test coverage for HILTI runtime library: references,
debug logger, configuration, safe-int, iterator, backtrace; along with related fixed and cleanup. (Benjamin Bannier, Corelight)

0.4.0-737 | 2020-08-11 07:39:31 +0000

* Fix exception handling across library boundaries. (Robin Sommer, Corelight)

* Allow limiting the "%port" of a unit to a specific direction. (Robin Sommer, Corelight)

We now allow to add attributes `&originator`/`&responder` to a `%port` unit property to limit it to the corresponding direction of a session. When selecting a parser through `--parser <port>` in spicy-driver, one can now add `&orig` or `&resp` to the port to indicate the desired directionality. If one leaves that off, only parsers are selected that have no directionality defined (i.e., just like it used to be).

* Add batch input mode to spicy-dump. spicy-dump gains an option `-F` to process multiple interleaved input streams pre-recorded in a batch input file. The format of that file is described in the manual, and there’s a Zeek script generating batch files from PCAP input traces. (Robin Sommer, Corelight)

* When naming a parser to use through "spicy-driver -p XXX", one can now also specify a port ("80/tcp") or a MIME type ("application/text") if the desired Spicy unit defines a corresponding property. See the documentation of spicy-driver for more. (Robin Sommer, Corelight)

* Port Zeek plugin to new driver stateful matching API. (Robin Sommer, Corelight)

0.4.0-730 | 2020-08-11 07:38:44 +0000

* Switch to Zeek 3.2.0 release packages for CI. (Robin Sommer, Corelight)

* Add helper Makefile to build and run the CI docker image locally. (Robin Sommer, Corelight)

0.4.0-725 | 2020-08-05 12:44:26 +0200

* Remove conversion of `rt::Time` to `bool`. (Benjamin Bannier, Corelight)

* Fix assignment operator for `rt::Union` of non-copyable types. (Benjamin Bannier, Corelight)

* Fix format string error possibly triggered when adding Times and Intervals. (Benjamin Bannier, Corelight)

0.4.0-716 | 2020-08-03 22:24:56 +0200

* Make default-constructed `rt::Address` instances invalid. (Benjamin Bannier, Corelight)

* Make `rt::Address` remember its address family.
This class was previously unable to distinguish IPv4 and IPv6 addresses
with all bits sets to zero from each other. By explicitly storing its
address family we are able to distinguish them, and can also preserve
the address family across operations like masking. (Benjamin Bannier, Corelight)

* Bump doctest to 2.4.0. (Benjamin Bannier, Corelight)

* Validate prefix length inputs when constructing `rt::Network` values. (Benjamin Bannier, Corelight)

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 0.4.0-703  | 2020-08-03 07:41:51 +0000 | * GH-444: Port Zeek plugin to Zeek 3.2. (Robin Sommer, Corelight)  
* GH-323: Suppress bogus warning by Zeek when installing our plugin. (Robin Sommer, Corelight)  
* Extend EVT file syntax with @if/@else/@endif preprocessor directives. For now, we support one type of expression for "@if": a Zeek version check. Example:  
  @if ZEEK_VERSION < 30200  
  ....  
  @else  
  ....  
  @endif (Robin Sommer, Corelight)  
  Also collecting all the Zeek includes into zeek-compat.h. (Robin Sommer, Corelight)  
* Fix frequent off-by-one errors in EVT error message locations. (Robin Sommer, Corelight) |

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 0.4.0-691  | 2020-07-31 15:19:57 +0200 | * Emitted fully typed initializer lists in codegen.  
* Correctly handle flipping with zero valid bytes in `rt::integer::flip`. (Benjamin Bannier, Corelight)  
* Add additional precondition checks to `rt::integer::bits`. (Benjamin Bannier, Corelight)  
* Consistently use `ntohX` type functions for `rt::integer::ntohX`. (Benjamin Bannier, Corelight) |

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 0.4.0-681  | 2020-07-23 14:55:25 +0000 | * Fix potential invalid iterator deref in debug output. (Robin Sommer, Corelight)  
* Add operator "bytes += uint8" for adding a single byte. (Robin Sommer, Corelight)  
* Fix missing filter end-of-input signal. (Robin Sommer, Corelight)  
* GH-423: Remove C{,XX}_SYSTEM_INCLUDE_DIRS cmake flags. (Robin Sommer, Corelight) |
If this functionality is needed, set CXX_FLAGS instead during configure, e.g.:

```
CXXFLAGS="-isystem /opt/rh/devtoolset-9/root/usr/include/c++/9" ./configure ..
```

* GH-441: Fix generation of lists of reserved words. (Robin Sommer, Corelight)

* GH-449: Fix documentation example. (Robin Sommer, Corelight)

0.4.0-670 | 2020-07-20 15:25:04 +0000

* GH-415: Add missing type coercion for parameterized types. (Robin Sommer, Corelight)

* GH-395: Fix tuple constructor expressions to not return a type until all elements are fully resolved. (Robin Sommer, Corelight)

* GH-402: Allow switch statements without any "case". (Robin Sommer, Corelight)

* GH-387: Extend a test's life time. (Robin Sommer, Corelight)

0.4.0-661 | 2020-07-16 07:06:14 +0000

* Fix build failures from building against Zeek 3.2 dev versions. (Tim Wojtulewicz, Corelight)

0.4.0-659 | 2020-07-16 06:34:30 +0000

* GH-352: Add chapter on custom host applications to manual. (Robin Sommer, Corelight)

* Series of smaller documentation updates. (Robin Sommer, Corelight)

  * Add documentation for types time/interval/optional.
  * Add list of reserved words.
  * Update debugging sections.
  * Add note about commutativity to operator documentation where applicable.
  * Add note about checking if optional unit variables remains unset still.
  * Fix typo in documentation for 'print'.
  * Fix an operator's description.

* GH-408: Remove "time + time" operator. (Robin Sommer, Corelight)

* GH-404: Add `time_ns(X)` and `interval_ns(X)` expressions to instantiate values with nanosecond resolution. (Robin Sommer, Corelight)

* Extend `(hilti,spicy)-config` with a `--dynamic-loading` option. This adapts `--ldflags` for applications that want to dynamically load precompiled parsers. (Robin Sommer, Corelight)

* Make `&cxxname` usable for user code: The code generator now automatically adds prototypes for the corresponding C++ functions.
To optionally disable that, a new attribute `&have_prototype` has been added. We also no longer require explicit global scoping for `&cxxname` IDs. (Robin Sommer, Corelight)

* Fix joint usage of options `spicyc` options `-cKo`. This would end up saving only the linker code, not the combined parsing + linker code, in the specific output file. (Robin Sommer, Corelight)

0.4.0-645 | 2020-07-16 06:33:23 +0000

* GH-423: Update configure's usage message. (Robin Sommer, Corelight)

* GH-433: Add support for declaring constants with explicit types. (Robin Sommer, Corelight)

0.4.0-641 | 2020-07-14 06:13:11 +0000

* GH-429: Fix generation of C++ prototypes for host applications. (Robin Sommer, Corelight)

0.4.0-639 | 2020-07-13 10:01:31 +0000

* GH-400: Fix build with static libraries. (Robin Sommer, Corelight)

* Fix ill-defined order of internal HILTI plugin execution. (Robin Sommer, Corelight)

* Remove JIT's (init,finish)Runtime() methods. They were no-ops already. (Robin Sommer, Corelight)

0.4.0-634 | 2020-07-09 12:32:17 +0200

* Lots of new & updates unit tests for the HILTI runtime. (Benjamin Bannier, Corelight)

* Remove `AddressFamily::` prefix when stringifying values. (Benjamin Bannier, Corelight)

0.4.0-612 | 2020-07-08 15:21:20 +0000

* GH-39: Add `spicy-dump`, a new standalone command line application that compiles and executes Spicy parsers on the fly and prints out the parsed information at the end in either a readable, custom ASCII format, or as JSON. (Robin Sommer, Corelight)

```bash
# 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
    }
}
```

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

(continues on next page)
* GH-56: Add reflection API to libhilti for dynamic, runtime inspection of parsed units. This includes:

  * A "TypeInfo(X)" expression for HILTI that, at runtime, will yield a pointer to a new `hilti::rt::TypeInfo` struct describing the type of `X`, which can be a type or an expression.

  * `include/rt/type-info.h` builds out a comprehensive set of classes to describe HILTI types, and to access their values generically, at runtime.

  * A 3rd kind of host application-side parse function for public units. Spicy's driver's `processInput()` now uses that.

* Change string rendering of ports to skip "Protocol::" prefix. (Robin Sommer, Corelight)

* Fix bug in move assignment operator of ValueReference. (Robin Sommer, Corelight)

* Add stringification for exception type. (Robin Sommer, Corelight)

* Add stringification for "any" type. (Robin Sommer, Corelight)

* Add missing &internal attributes to some codegenerated struct fields. This hides them from user visible output. (Robin Sommer, Corelight)

* Add equality operator to regexp type. (Robin Sommer, Corelight)

* Add operator<< for regexp. (Robin Sommer, Corelight)

* Support anonymous namespaces in C++ output. (Robin Sommer, Corelight)

* Add missing printer code for iterator<set> and iterator<map>. (Robin Sommer, Corelight)

* Add const version of vector iterator's dereference operator. (Robin Sommer, Corelight)

* Add -O0 to libhilti/libspicy for debug builds. (Robin Sommer, Corelight)

0.4.0-572 | 2020-07-08 11:36:53 +0200

* Fix parsing of `rt::Port` for out-of-range port values. (Benjamin Bannier, Corelight)

0.4.0-552 | 2020-07-03 15:47:06 +0200

* Switch `real::unpack` error reporting to use `rt::Result`. (Benjamin Bannier, Corelight)

0.4.0-549 | 2020-07-01 15:03:27 +0200
* Remove unintended conversion of `rt::Result` to `bool`.

0.4.0-547 | 2020-07-01 11:41:06 +0200

* GH-392: Fix handling of vector of bools. (Benjamin Bannier, Corelight)

0.4.0-544 | 2020-07-01 09:35:52 +0200

* Remove `hilti::rt::List` in favor of `hilti::rt::Vector`. (Benjamin Bannier, Corelight)

0.4.0-538 | 2020-06-30 08:08:16 +0000

* Fixed semantics of `hilti::rt::Interval`-bool conversions. They now return true for non-zero intervals. (Benjamin Bannier, Corelight)

* Add tests of `rt::Interval` stringification. (Benjamin Bannier, Corelight)

* Add tests of `result::Error` stringification. (Benjamin Bannier, Corelight)

0.4.0-532 | 2020-06-30 08:03:40 +0000

* Add a DHCP analyzer. (Benjamin Bannier, Corelight)

* Fix filename in getting started docs. (Dirk Loss)

* Move reporting logic for `--report-times` so that it always executes when the HILTI-side driver destructs. (Robin Sommer, Corelight)

* Fix header inclusion in Spicy plugin. (Benjamin Bannier, Corelight)

* Remove unused `is_optional` type trait. (Benjamin Bannier, Corelight)

* Enable translation of optional types on the Spicy-Zeek interface. (Benjamin Bannier, Corelight)

0.4.0-525 | 2020-06-26 14:55:08 +0200

* Add dedicated method to resize and assign a vector. (Benjamin Bannier, Corelight)

0.4.0-522 | 2020-06-24 22:24:34 +0200

* Fix `hilti::rt::is_optional` so it applies to const and ref of optionals also. (Benjamin Bannier, Corelight)

0.4.0-520 | 2020-06-24 06:59:04 +0000

* GH-398: Remove duplicate spicy-driver CMake target. (Benjamin Bannier, Corelight)

0.4.0-518 | 2020-06-24 06:50:48 +0000

* GH-140: Add `#requires`/`%requires` unit attribute/property. (Robin Sommer, Corelight)
Fields may now 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 unparseable in the first place.

One can similarly enforce conditions globally at the unit level through a property ``%requires = EXPR`` , which 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.

* Add Doxygen output to Sphinx documentation. It's integrated into the "Read the Docs build". Locally, "make doxygen" will run it. (Benjamin Bannier, Corelight)
* Clean up Sphinx custom extension. (Benjamin Bannier, Corelight)
  - Remove use of `sphinx.locale.i_`
  - Format with autopep8.

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Commit Details</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.4.0-510</td>
<td>2020-06-23 16:16:43 +0200</td>
<td>* GH-386: Add codegen for Spicy time difference. (Benjamin Bannier, Corelight)</td>
<td></td>
</tr>
<tr>
<td>0.4.0-508</td>
<td>2020-06-22 06:03:34 +0000</td>
<td>* Series of improvements for utility infrastructure (Benjamin Bannier, Corelight)</td>
<td></td>
</tr>
<tr>
<td>0.4.0-496</td>
<td>2020-06-18 15:46:44 +0000</td>
<td>* GH-381: Have the code generator fully declare enum types directly as part of the forward-declarations block. (Robin Sommer, Corelight)</td>
<td></td>
</tr>
<tr>
<td>0.4.0-492</td>
<td>2020-06-17 10:41:13 +0200</td>
<td>* GH-380: Corrections to tutorial. (Duffy Ocraven)</td>
<td></td>
</tr>
<tr>
<td>0.4.0-490</td>
<td>2020-06-17 10:36:36 +0200</td>
<td>* Move hilti's <code>util</code> namespace into the <code>hilti</code> namespace. (Benjamin Bannier, Corelight)</td>
<td></td>
</tr>
</tbody>
</table>
* GH-228: Prevent more overflows when working with `hilti::rt Time and Interval`. (Benjamin Bannier, Corelight)
  - Change `hilti::rt::Interval` to use safe integers.
  - Change `hilti::rt::Time` to use safe integers.
  - Fix some typos in `hilti/base/util.h`.
  - Remove hilti-rt-tests dependency on hilti library.
  - Reorganize test cases for `hilti::util`.

0.4.0-484 | 2020-06-16 07:15:25 +0000

* Add new tutorial chapter. (Robin Sommer, Corelight)

* Add Zeek-side option "Spicy::enable_print" to control whether the output of Spicy-side "print" output is shown. Default is off (which is a change in behaviour compared to previous versions).

* Clean up representation of type arguments and function calls in auto-generated operator reference. (Robin Sommer, Corelight)

* Make the target stream of `hilti::print()` configurable. Default remains standard output. Leaving it unset will silence `hilti::print()`. (Robin Sommer, Corelight)

* Do not use use "using X === ..." for forward-declaring struct types in C++ output, as that could accidentally pull in IDs from outer namespaces. (Robin Sommer, Corelight)

* Fix scanner for accepting comment lines at end of file without terminating newline. (Robin Sommer, Corelight)

0.4.0-477 | 2020-06-15 07:36:43 +0000

* GH-293: Fix precedence of "in" operator. (Robin Sommer, Corelight)

* GH-319: When using &convert with a list expression, transparently turn the result into a vector. (Robin Sommer, Corelight)

* GH-324: Fix error message that was too specific about source type. (Robin Sommer, Corelight)

* GH-325: Fix a couple of segfaults in scope builder when hitting invalid code. (Robin Sommer, Corelight)

* Update spicy-driver usage message. (Robin Sommer, Corelight)

* Handle unknown options in getopt() consistently in driver code. (Robin Sommer, Corelight)

* GH-344: Type check assignments and fix constness propagation. (Robin Sommer, Corelight)

* GH-345: Validate type of &default expressions. (Robin Sommer, Corelight)

* Add more debug information about constness to AST. (Robin Sommer, Corelight)
0.4.0-467 | 2020-06-15 07:18:03 +0000

* Redo testing of `hilti::rt::Stream` and clean up implementation (Benjamin Bannier, Corelight)
  - Add Doxygen documentation to `rt::Stream`.
  - Add `begin` and `end` methods for `rt::list::Empty`.
  - Add additional nullptr checks when accessing `rt::Stream` iterator chunks or `contents`.
  - Fix escaping of `rt::Stream` and `rt::View` in stringification to ensure all non-printable byte sequences are escaped.
  - Make `rt::stream::View` iterators be safe by default.
  - Prevent nullptr dereference in `rt::Stream` iterator `chunk` and `content`.
  - Remove `rt::Stream::safe_[begin|end]`.
  - Remove `rt::stream::View::safe_[begin|end]`.
  - Remove `safe_begin` and `safe_end` extension points.
  - Remove unused `Chunk::begin`.
  - Remove unused `rt::Stream::chain` function.
  - Remove unused `rt::Stream::compare` functions.
  - Remove use of unsafe iterators in a few places.
  - Rename `rt::iterator::safe_range` to drop the 'safe' qualifier.
  - Add lots of new test cases, and reorganize exists test cases.

* Add `begin` and `end` methods for `rt::list::Empty` (Benjamin Bannier, Corelight)

0.4.0-430 | 2020-06-12 08:07:16 +0200

* Reenable some already passing clang-tidy checks (Benjamin Bannier, Corelight)

0.4.0-428 | 2020-06-12 08:06:23 +0200

* GH-274: Unbreak build for when other Spicy headers are already installed.

  If Spicy headers were already installed into a path somewhere in the include lookup, we could have found the same header twice, preferring the installed header over the one of the current checkout. This could have lead to diagnostics about duplicate definitions since due to use of `#pragma once` in our headers the compiler would not have detected that a header was already included (and even if it would have detected this, it would have included the incorrect header).

  This patch adds include paths from the current checkout before any system paths.

0.4.0-426 | 2020-06-09 17:16:13 +0000

* Increase CI memory limit for Ubuntu Docker further (Robin Sommer, Corelight)

* GH-352: Fix Zeek link in README (Robin Sommer, Corelight)

0.4.0-424 | 2020-06-09 15:22:26 +0000

* GH-355: Allow access to unit parameters inside default expressions for unit variables (Robin Sommer, Corelight)
* GH-315: Fix autogen-spicy-lib to extract multi-line comments.  
  (Robin Sommer, Corelight)

0.4.0-417 | 2020-06-08 10:26:32 +0000
* Increase CI memory limit for GCC further. (Robin Sommer, Corelight)
* Fix GCC warning. (Robin Sommer, Corelight)

0.4.0-415 | 2020-06-08 08:28:19 +0000
* GH-368: Fix &convert for vectors of enums. (Robin Sommer, Corelight)
* Add missing HILTI printer code for "unset" operator. (Robin Sommer, Corelight)

0.4.0-411 | 2020-06-05 07:28:40 +0000
* GH-361: Fix construction of vectors with element types that provide default values. (Robin Sommer, Corelight)
* GH-365: Zeek plugin: Catch if more parameters are given in an EVT spec than Zeek expects. (Robin Sommer, Corelight)
* Remove branch name from Cirrus CI emails due to API limitations. (Benjamin Bannier, Corelight)

0.4.0-406 | 2020-06-04 09:14:27 +0000
* GH-354: No longer allow dashes inside identifiers. (Robin Sommer, Corelight)
* GH-350: Restrict parsing of attributes and properties to avoid ambiguities. (Robin Sommer, Corelight)
* Clean up the expression mode in the Spicy Bison parser. (Robin Sommer, Corelight)

  This comes with a couple of grammar changes to avoid ambiguities:
  - "new X" no longer allows full expressions for X, just values or identifiers.
  - Switch cases that use look-ahead now also need to be introduced with an arrow, e.g.:

    ```
    switch {
      ->a : A;
      ->b : B;
      ->c : C;
    };
    ```

  * Reduce resources requested from Cirrus. (Robin Sommer, Corelight)
* Make iterators into vectors ordered. (Benjamin Bannier, Corelight)
* GH-343: Fix duplicate evaluation of if-condition when used with &parse-at. (Robin Sommer, Corelight)
* Add validation check for initializations that use the ID that's currently being declared. (Robin Sommer, Corelight)
* Fix mailing list links in README. (Robin Sommer, Corelight)

0.4.0-393 | 2020-06-04 08:41:28 +0200

* Rework implementation of `hilti::rt::Bytes`.
  - Add test cases for `rt::Bytes` member functions.
  - Fix construction of `rt::Bytes` from string and charset.
  - Fix handling of undefined byte order in ByteOrder conversions to integers.
  - Prevent throwing `std::out_of_range` from `rt::Bytes::extract`.
  - Add additional safety to `bytes::Iterator`.
  - Made some `rt::Bytes` methods `const`.
  - Remove rvalue overload of `rt::Bytes::str`.

* Fix ADL for `rt::to_string`. (Benjamin Bannier, Corelight)
* Fix `rt::util::split` for empty delimiter case. (Benjamin Bannier, Corelight)

0.4.0-364 | 2020-06-04 08:40:25 +0200

* Make functions of `rt::string` throw instead of triggering internal errors for invalid input. (Benjamin Bannier, Corelight)

0.4.0-362 | 2020-06-02 15:20:59 +0000

* GH-282: Set PATH when building documentation so that we find the Spicy tools. (Robin Sommer, Corelight)
* GH-348: Fix integer overflows in regexp parsing & matching. (Robin Sommer, Corelight)

0.4.0-356 | 2020-05-28 10:23:04 +0000

* Small documentation updates, including some notes on DPD integration and standalone plugin builds. (Robin Sommer, Corelight)

0.4.0-348 | 2020-05-27 13:03:22 +0200

* GH-335: Add http_all_headers() event to HTTP EVT file.

  The event isn't used anywhere by default in Zeek, but it's a nice example of transforming Spicy information on the fly to match a Zeek-side event signature.

# 335(Robin Sommer, Corelight)

0.4.0-346 | 2020-05-27 13:00:28 +0200

* jrx: Fix leak in `jrx_match_state_done`. (Benjamin Bannier, Corelight)
* jrx: Fix test harness. (Benjamin Bannier, Corelight)
* Fix leaks of jrx state when copying `regexp::MatchState`. (Benjamin Bannier, Corelight)
* jrx: Allow copying from const. (Benjamin Bannier, Corelight)
* Bump doctest to 2.3.8. (Benjamin Bannier, Corelight)
0.4.0-325 | 2020-05-27 06:49:26 +0000
* GH-333: Validate that unit fields have parseable types. (Robin Sommer, Corelight)

* GH-330: Add range check to port parsing in EVT files. (Robin Sommer, Corelight)

* GH-335: Add support for converting maps from Spicy to Zeek. (Robin Sommer, Corelight)

* Send emails for successful CI builds. (Benjamin Bannier, Corelight)

0.4.0-318 | 2020-05-26 10:07:38 +0000
* Change Slack links to point to Zeek's Connect page. (Robin Sommer, Corelight)

0.4.0-317 | 2020-05-25 09:54:25 +0200
* GH-286: Fix regexp handling to respect &size. (Benjamin Bannier, Corelight)

0.4.0-311 | 2020-05-21 07:54:43 +0000
* Fix JIT-less build of Zeek plugin. (Robin Sommer, Corelight)

0.4.0-309 | 2020-05-20 12:35:36 +0000
* GH-299: Support compiling the Zeek plugin without JIT support. (Robin Sommer, Corelight)

If configured with `--disable-jit`, the Zeek plugin will now still be built and will support loading pre-compiled HLTO files. The plugin will have dependencies only on system libraries, and hence can be installed on systems that don't have the Spicy libraries available.

There's also a new configure option `--disable-jit-for-zeek` that disables JIT support just for the Zeek plugin, creating the same dependency-free result.

* GH-83: Add standalone tool spicyz to precompile analyzers for Zeek. (Robin Sommer, Corelight)

This replaces the previous shell wrapper around Zeek, meaning that we can now precompile Zeek analyzers without actually relying on having a Zeek binary itself available. (We continue to require access to the Zeek development headers.)

As a bonus feature, the new spicyz comes with an option `--c <prefix>` that, instead of producing a fully compiled HLTO, just writes out all the generated C++ code into files, each named `<prefix>-<module ID>.cc`. This goes along way towards supporting compiling Spicy analyzers statically into an application, similar to how BinPAC is being used (GH-72).

* Fix for Bison versions >= 3.6. (Robin Sommer, Corelight)

(continues on next page)
0.4.0-295 | 2020-05-20 07:23:14 +0000
* Fix a couple typos in documentation. (Robin Sommer, Corelight)

0.4.0-293 | 2020-05-19 06:57:49 +0000
* Rework implementation of `hilti::rt::List`. (Benjamin Bannier, Corelight)
  - Hide that `rt::List` inherits from `std::list` from users.
  - Make iterators of `rt::List` safe.
  - Add tests for `rt::List`.

0.4.0-286 | 2020-05-18 10:38:44 +0000
* Remove FAQ pointer from README. (Robin Sommer, Corelight)
* Fix bug in EVT reading. (Robin Sommer, Corelight)

0.4.0-284 | 2020-05-18 08:29:49 +0000
* GH-237: Add library function `spicy::strftime`. (Benjamin Bannier, Corelight)

0.4.0-282 | 2020-05-18 08:28:08 +0000
* GH-312: Fix comment parsing in EVT files. (Robin Sommer, Corelight)

0.4.0-280 | 2020-05-15 07:11:01 +0000
* Rework implementation of `hilti::rt::Map`. (Benjamin Bannier, Corelight)
  - Hide that `rt::Map` inherits from `std::map` from users.
  - Make iterators of `rt::Map` safe.
  - Slim down `AssignProxy`.
  - Fix `rt::Map::get`.
  - Add test for `rt::Map` stringification.
* Fix display of compiler flags in JIT debug log. (Benjamin Bannier, Corelight)
* Document using Docker Hub images. (Benjamin Bannier, Corelight)

0.4.0-270 | 2020-05-14 07:00:54 +0000
* Clean up example code. (Robin Sommer, Corelight)
* Change structure of example section, and add more links. (Robin Sommer, Corelight)
* Make autogen-docs script more robust. (Robin Sommer, Corelight)
* Fix documentation of -= and /= operators. (Robin Sommer, Corelight)

0.4.0-264 | 2020-05-13 14:18:08 +0000
* Add an end-to-end TFTP analyzer for Zeek, 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. (Robin Sommer, Corelight)

* GH-306: Do not fail documentation build for broken links. (Benjamin Bannier, Corelight)

0.4.0-260 | 2020-05-12 10:37:16 +0000

* Documentation updates. (Robin Sommer, Corelight)

  - Document list type.
  - Include CHANGES into the documentation's release notes.
  - Update link to macOS LLVM binaries. (Robin Sommer, Corelight)

* Clean up parsing of list constructor. (Robin Sommer, Corelight)

* Clean up cast operators. (Robin Sommer, Corelight)

* Remove a couple of empty source files. (Robin Sommer, Corelight)

0.4.0-251 | 2020-05-12 10:33:24 +0000

* Redo implementation of `hilti::rt::Set`. (Benjamin Bannier, Corelight)

  - Hide that `rt::Set` inherits from `std::set` from users.
  - Make iterators of `rt::Set` safe.
  - Add test of `rt::Set` stringification.

0.4.0-245 | 2020-05-11 08:39:49 +0000

* Redo implementation of `hilti::rt::Vector`. (Benjamin Bannier, Corelight)

  - Hide that `rt::Vector` inherits from `std::vector` from users.
  - Replace iterators of `rt::Vector` with safe alternatives.
  - Add Doxygen comments to `rt::Vector`.
  - Remove support for stringification of `std::vector` in `hilti::rt`.
  - Add test for stringification of `Empty` vector.
  - Make `rt::MIMEType` default-constructible.
  - Fix includes of `rt/mime.h`.
  - Use less ambiguous form for generated `rt::Vector` constructors.

* Require Zeek plugin for Zeek protocol parser tests. (Benjamin Bannier, Corelight)

0.4.0-234 | 2020-05-11 08:33:55 +0000

* GH-229: Add a Spicy function to render bytes as hex string. (Benjamin Bannier, Corelight)

  ```
  ## Returns a bytes value rendered as a hex string.
  public function bytes_to_hexstring(value: bytes) : string;
  ```

* GH - 200 : Add library function `spicy::getenv()` to retrieve environment variables. (Robin Sommer, Corelight)
## Returns the value of an environment variable, if set.

```c
public function getenv(name
    : string)
: optional<string>;
```

* GH-200: Add option -f to `spicy-driver` to read input from file.
  (Robin Sommer, Corelight)

* Use fully qualified IDs for internal `&cxxname` in libraries `hilti.hlt` and `spicy.spicy`. (Benjamin Bannier, Corelight)

* Fix shell commands in `docker/Makefile`. (Benjamin Bannier, Corelight)

0.4.0-226 | 2020-05-06 15:12:52 +0000

* Apply a set of tweaks to specifics of &size. (Robin Sommer, Corelight)
  
  - We now enforce that a field consumes all its &size characters of input, unless explicitly overridden where a types allow for that (as, e.g., "bytes" does with an additional &eod).
  
  - We slightly change semantics for parsing bytes: A &size of zero will now set the field an empty bytes value; it used to leave the field unset.
  
  - Document that &size works with all fields (which it already did).
  
  - Clean up the implementation of &size.

* GH-289: Do not perform look-ahead parsing for vectors fed from &parse-from/at. (Robin Sommer, Corelight)

* GH-288: Rename "&until_including" attribute to "&until-including". (Robin Sommer, Corelight)

* GH-269: Add support for "&until-including" when parsing bytes. (Robin Sommer, Corelight)

* GH-286: Fix Bison parser's attribute association for regular expressions. (Robin Sommer, Corelight)

* Add an "unset" operator to HILTI's struct type to clear optional types. (Robin Sommer, Corelight)

0.4.0-216 | 2020-05-05 12:25:40 +0000

* GH-212: Resize eagerly when accessing out-of-bounds Vector elements. (Benjamin Bannier, Corelight)

0.4.0-214 | 2020-05-04 10:51:06 +0000

* GH-267: Fix spicy-build to support absolute output paths. (Robin Sommer, Corelight)
* GH-275, GH-276: Sort options in the usage messages of various tools. (Robin Sommer, Corelight)

0.4.0-208 | 2020-05-04 10:49:21 +0000

* Use doctest test suites as top-level grouping. (Benjamin Bannier, Corelight)

* GH-268: Properly escape data when stringifying streams. (Benjamin Bannier, Corelight)

0.4.0-203 | 2020-05-03 15:37:30 +0000

* GH-265: Require unit properties to end with a semicolon. We were inconsistent with that. Note that this will break existing code that wasn't using the semicolons, including some examples from the documentation (which have been updated). For example, it's now `%filter;` instead of `%filter`. (Robin Sommer, Corelight)

* GH-266: Fix missing validation of unit properties. (Robin Sommer, Corelight)

* GH-262: Allow signs in intN/uintN() constructors. (Robin Sommer, Corelight)

* GH-254: Make UTF8 default for bytes::decode(). (Robin Sommer, Corelight)

* GH-248: Fix AST inconsistency with how modules were stored in scopes. (Robin Sommer, Corelight)

* GH-263: Improve error messages for unresolved operators and IDs. (Robin Sommer, Corelight)

* Improve suppression of repetitive errors. (Robin Sommer, Corelight)

* Avoid leaking passwords into command lines in `docker/Makefile`. (Benjamin Bannier, Corelight)

0.4.0-189 | 2020-04-30 10:33:54 +0000

* GH-196: Reject container constructors with inconsistent types, and coerce elements once the type is known. (Robin Sommer, Corelight)

0.4.0-187 | 2020-04-30 10:26:13 +0000

* Improve `vector` implementation. (Benjamin Bannier, Corelight)

  - Fix semantics of safe mutating access to `hilti::rt::Vector` elements.
  - Remove broken `rt::Vector::set`.
  - Fix includes of `hilti/rt/types/vector.h`.
  - Add unit test for a couple of `hilti::rt::Vector` methods.

* GH-272: Document precompiled MacOS packages. (Benjamin Bannier, Corelight)

0.4.0-179 | 2020-04-29 15:15:14 +0200
* GH-192: Remove internal errors that we ended up running into during AST validation. (Robin Sommer, Corelight)

* GH-178: Fix spurious semicolons in C++ output and potential problem in C++ block rendering (Robin Sommer, Corelight)

0.4.0-173 | 2020-04-29 08:20:21 +0000

* Report column information in error messages. Locations are emitted as `\langle line\rangle: \langle character\rangle` so that e.g., the second character on the fourth line would be displayed as `4:2`. This extends to ranges as well which look e.g., like `1:1-2:30`. (Benjamin Bannier, Corelight)

0.4.0-171 | 2020-04-28 19:08:50 +0000

* Add a section to the development manual on the structure of the runtime libraries. (Robin Sommer, Corelight)

* Fix `hilti::rt::to_string` for character literals. (Benjamin Bannier, Corelight)

* GH-179: Enable `hilti::rt::to_string` for `std::vector`. (Benjamin Bannier, Corelight)

* Remove return of rvalue when generating parse functions. (Benjamin Bannier, Corelight)

0.4.0-162 | 2020-04-28 10:14:01 +0000

* GH-219: Clean up string interface exposed by `rt::Bytes` to remove overloading ambiguities. (Benjamin Bannier, Corelight)

* GH-257: Fix documentation whitespace issues and remove outdated FIXMEs around usages of `Result`. (Benjamin Bannier, Corelight)

0.4.0-154 | 2020-04-27 11:58:00 +0000

* GH-242: Suppress any output from doc's `make check` by default. (Benjamin Bannier, Corelight)

* GH-251: Ignore package links in documentation linkcheck. (Benjamin Bannier, Corelight)

* Edit pass on documentation's reassembly section. (Robin Sommer, Corelight)

0.4.0-148 | 2020-04-24 13:19:22 +0000

* GH-226: Add CI setup for building published Docker images. This patch adds an `update_dockerhub` target to `docker/Makefile`, which builds, tests and uploads images to Dockerhub. It also changes the Dockerfiles in `docker/` back to include Spicy build commands, so that they become practically self-documenting. (Benjamin Bannier, Corelight).
* GH-222: Output changes to make rendered ASTs easier to parse
  (Benjamin Bannier, Corelight)

0.4.0-140 | 2020-04-24 09:18:51 +0000
* Tune update-changes configuration. (Robin Sommer, Corelight)
* GH-245: Fix typo in documentation. (Dominik Charousset)

0.4.0-136 | 2020-04-23 16:53:22 +0000
* Update link in README. (Yacin Nadji)
* Fix link to FAQ in README.
* Remove left-over debug output from DNS grammar.

0.4.0-130 | 2020-04-23 10:44:12 +0000
* GH-170: Change "+<int>" to create a signed integer value. It used
  to create an unsigned integer.
* GH-225: Preserve parentheses inside expressions during code
  generation.
* GH-231: Add range check to digits in Spicy's atoi_n().
* GH-210: Catch & report broken escape sequences in HILTI/Spicy code
  at parse time.
* GH-217: Fix code generation of escaped C++ string literals.

0.4.0-120 | 2020-04-22 15:40:34 +0000
* Fix package artifact links in docs.
* Fix documentation links to Spicy mailing lists.
* Fix issue base URL in documentation.
* Add documentastion build target to check links.
* Fix package creation in CI.

0.4.0-114 | 2020-04-22 13:17:05 +0000
* Make sure auto-generated documentation items have stable ordering.
* Cache CI ccache even if build failed.

0.4.0-111 | 2020-04-22 10:23:15 +0000
* Build documentation from documentation folder instead of from a
  symlink inside the build directory.

0.4.0-109 | 2020-04-21 11:40:41 +0000

(continues on next page)
* Execute CI jobs automatically.
* Fix potential sign-unsigned mismatch in comparison.
* Fix unused variable warning.

0.4.0-105 | 2020-04-21 10:41:25 +0000
* Update documentation links in README.

0.4.0-104 | 2020-04-21 10:15:14 +0000
* Tweak documentation build for standalone operation.
* GH-172: Re-add missing autogenerated usage for spicy-build.
* Add map & set to the auto-documented types.

0.4.0-96 | 2020-04-21 08:27:51 +0000
* GH-129: Add infrastructure and documentation for providing binary packages.

0.4.0-91 | 2020-04-21 07:38:18 +0000
* GH-174: Fix a number of smaller documentation issues.
* GH-191: Change bytes iterator's deref operator to return uint8 instead of uint64, and fix implementation to no longer use a signed integer internally.
* GH-153: Prettify error messages for fmt() problems.
* GH-154: Fix crash when accessing an unknown tuple element.
* GH-171: Fix spelling in some diagnostic messages.
* GH-175: Catch unresolved unit fields during validation.
* GH-198: Show functions as "<function>" when rendering them.
* GH-203: Add type check for elements in vector/set/map constructors.
* GH-204: Remove support for legacy syntax for parsing sequences. "vector<T>" is no longer supported for parsing sequences; one must now use the new "T[]" syntax instead.
* Remove Spicy's legacy list type (which was just an alias for "vector").
* GH-181: Validate correct usage of struct/unit parameters.
* Add a number of validation checks for unit hooks.
* GH-195: Add range check for bytes' to_int() base parameter, which now must be between 2 and 36.
* GH-155: Overhaul AST error reporting for more concise error messages.

* GH-119: Generally include location information when throwing runtime exceptions.

* Add packaging to CMake setup and CI jobs.

* Set proper CMake project version.

* Execute installation tests in Docker CI tasks.

** 0.4.0-48 | 2020-04-15 16:56:28 +0000

* GH-163: Fix initialization of &default struct parameters.

* GH-156: Fix look-ahead parsing for vectors terminated through end-of-data.

* GH-182: Fix "for" loop's type resolution for the iteration variable.

* GH-180: Fix printing of bytes instances that have non-printable characters. This also changes how we print "\n/\r/\t" inside bytes: We now render them as "\xXX", like any other non-printable character.

* Add missing 'Undef' to byte-order enum.

* Add clang-format to pre-commit config.

* Add minimal CI test runs for CentOS 8, Alpine 3.11, and Ubuntu 19.10.

* Clean up our Docker setup/usage.

** 0.4.0-29 | 2020-04-15 09:04:28 +0000

* Add pre-commit check that generated docs are up-to-date.

* Clean up spicy-doc-to-rst and make output stable.

* Add auto-generated docs to the tree so that one can build the full documentation without needing to build Spicy itself.

** 0.4.0-20 | 2020-04-09 15:11:43 +0000

* GH-159: Properly close quotes and parens in "hilti-flow" debug statement.

* GH-146: Validate that &byte-order attributes have an argument.

** 0.4.0-16 | 2020-04-09 15:06:04 +0000

* GH-92: Documentation updates.

** 0.4.0-13 | 2020-04-09 11:23:15 +0000

(continues on next page)
2.9.2 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., `%random-access;`).
- 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.

• Units using any of the random access methods (e.g., input()), now need to explicitly add a unit property %random-access.

• 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 &ge;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 a smaller subset of tests for quick results

`btest -g spicy-core` (or `make test-core` from the top-level directory) will run a smaller set of selected tests that exercise Spicy’s main pieces. This is not sufficient to see if everything works, but can quickly determine that something clearly doesn’t. :)

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, 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 `btest`s, 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.

**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:

```bash
# 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:

```bash
# 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:

```bash
# 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:

```bash
# ../scripts/run-benchmark -t /path/to/spicy-benchmark-m57/long run
```

```
<table>
<thead>
<tr>
<th></th>
<th>User</th>
<th>System</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>http-static</td>
<td>1.55</td>
<td>0.00</td>
<td>1.55</td>
</tr>
<tr>
<td>http-hlto</td>
<td>1.77</td>
<td>0.00</td>
<td>1.77</td>
</tr>
<tr>
<td>http-zeek-spicy</td>
<td>4.97</td>
<td>4.87</td>
<td>5.02</td>
</tr>
<tr>
<td></td>
<td>1.77</td>
<td></td>
<td>1.77</td>
</tr>
<tr>
<td>http-zeek-std</td>
<td>3.69</td>
<td>3.57</td>
<td>3.74</td>
</tr>
<tr>
<td></td>
<td>4.90</td>
<td></td>
<td>4.90</td>
</tr>
<tr>
<td>dns-static</td>
<td>0.01</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>dns-hlto</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>dns-zeek-spicy</td>
<td>0.97</td>
<td>0.94</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>1.00</td>
<td></td>
<td>1.00</td>
</tr>
<tr>
<td>dns-zeek-std</td>
<td>0.80</td>
<td>0.76</td>
<td>0.76</td>
</tr>
<tr>
<td></td>
<td>1.00</td>
<td></td>
<td>1.00</td>
</tr>
</tbody>
</table>
```

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:

```bash
# 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:

```bash
$ 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:

```bash
$ 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 captial 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 $\geq$ 10 because we need a style option that wasn’t available earlier. There’s a format target in the top-level Makefile that checks all relevant source files for conformance with our style. A second target format-fixit reformats Spicy’s code where it doesn’t confirm.
To ensure the right clang-format version is being used, either have it in your PATH or set the environment variable CLANG_FORMAT to the full path of the binary before running either of these targets.

Spicy’s CI runs make 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_undercore 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 3

Index

- genindex
- modindex
- search
Symbols
%on_gap (method), 76
%on_overlap (method), 76
%on_skipped (method), 76
%on_undelivered (method), 76
A
address::Equal (operator), 58
address::family (method), 58
address::Unequal (operator), 58
B
bitfield::Member (operator), 59
bool_:::Equal (operator), 59
bool_:::Unequal (operator), 59
bytes::at (method), 60
bytes::Begin (operator), 61
bytes::decode (method), 60
bytes::End (operator), 61
bytes::Equal (operator), 61
bytes::find (method), 60
bytes::Greater (operator), 61
bytes::GreaterEqual (operator), 61
bytes::In (operator), 61
bytes::iterator::Deref (operator), 62
bytes::iterator::Difference (operator), 62
bytes::iterator::Equal (operator), 62
bytes::iterator::Greater (operator), 62
bytes::iterator::GreaterEqual (operator), 62
bytes::iterator::IncrPostfix (operator), 62
bytes::iterator::IncrPrefix (operator), 62
bytes::iterator::Lower (operator), 62
bytes::iterator::LowerEqual (operator), 62
bytes::iterator::Sum (operator), 62
bytes::iterator::SumAssign (operator), 62
bytes::iterator::Unequal (operator), 62
bytes::join (method), 60
bytes::lower (method), 60
bytes::Lower (operator), 61
bytes::LowerEqual (operator), 61
bytes::match (method), 60
bytes::Size (operator), 61
bytes::split (method), 60
bytes::split1 (method), 60
bytes::starts_with (method), 60
bytes::strip (method), 60
bytes::sub (method), 60
bytes::Sum (operator), 61
bytes::SumAssign (operator), 61, 62
bytes::to_int (method), 60, 61
bytes::to_time (method), 61
bytes::to_uint (method), 61
bytes::Unequal (operator), 62
bytes::upper (method), 61
E
enum_:::Call (operator), 63
elem_:::Cast (operator), 63
enum_:::Equal (operator), 63
elem_:::has_label (method), 63
elem_:::Unequal (operator), 63
I
integer::BitAnd (operator), 64
integer::BitOr (operator), 64
integer::BitXor (operator), 64
integer::Cast (operator), 64
integer::DecrPostfix (operator), 64
integer::DecrPrefix (operator), 64
integer::Difference (operator), 64
integer::DifferenceAssign (operator), 65
integer::Division (operator), 65
integer::DivisionAssign (operator), 65
integer::Equal (operator), 65
integer::Greater (operator), 65
integer::GreaterEqual (operator), 65
integer::IncrPostfix (operator), 65
integer::IncrPrefix (operator), 65
integer::Lower (operator), 65
integer::LowerEqual (operator), 65
integer::Modulo (operator), 65, 66
integer::Multiple (operator), 66
integer::MultipleAssign (operator), 66
integer::Negate (operator), 66
integer::Power (operator), 66
integer::ShiftLeft (operator), 66
integer::ShiftRight (operator), 66
integer::SignNeg (operator), 66
integer::Sum (operator), 66
integer::SumAssign (operator), 66
integer::Unequal (operator), 66

L
list::Begin (operator), 68
list::End (operator), 68
list::Equal (operator), 68
list::Size (operator), 68
list::Unequal (operator), 68

M
map::Begin (operator), 69
map::clear (method), 69
map::Delete (operator), 69
map::End (operator), 69
map::Equal (operator), 69
map::get (method), 69
map::In (operator), 69
map::Index (operator), 69
map::iterator::Deref (operator), 69
map::iterator::Equal (operator), 69
map::iterator::IncrPostfix (operator), 69
map::iterator::IncrPrefix (operator), 69
map::iterator::Unequal (operator), 69
map::Size (operator), 69
map::Unequal (operator), 69

O
optional::Deref (operator), 70

P
port::Equal (operator), 70
port::protocol (method), 70
port::Unequal (operator), 70

R
real::Cast (operator), 71
real::Difference (operator), 71
real::DifferenceAssign (operator), 71
real::Division (operator), 71
real::DivisionAssign (operator), 71
real::Equal (operator), 71
real::Greater (operator), 71
real::GreaterEqual (operator), 71
real::Lower (operator), 71
real::LowerEqual (operator), 71
real::Modulo (operator), 71
real::Multiple (operator), 71
real::MultipleAssign (operator), 71
real::Power (operator), 71
real::SignNeg (operator), 71
real::Sum (operator), 72
real::SumAssign (operator), 72
real::Unequal (operator), 72
regexp::find (method), 72
regexp::match (method), 72
regexp::match_groups (method), 72
regexp::token_matcher (method), 73

S
set::Add (operator), 73
set::Begin (operator), 73
set::clear (method), 73
set::Delete (operator), 73
set::End (operator), 73
set::Equal (operator), 73
set::In (operator), 73
set::iterator::Deref (operator), 74
set::iterator::Equal (operator), 74
set::iterator::IncrPostfix (operator), 74
set::iterator::IncrPrefix (operator), 74
set::iterator::Unequal (operator), 74
set::Size (operator), 74
set::Unequal (operator), 74
sink::close (method), 74
sink::connect (method), 74
sink::connect_filter (method), 74
sink::connect_mime_type (method), 75
sink::gap (method), 75
sink::sequence_number (method), 75
sink::set_auto_trim (method), 75
sink::set_initial_sequence_number (method), 75
sink::set_policy (method), 75
sink::Size (operator), 76
sink::skip (method), 75
sink::trim (method), 75
sink::write (method), 75
stream::at (method), 77
stream::Begin (operator), 77
stream::End (operator), 77
stream::freeze (method), 77
stream::is_frozen (method), 77
stream::iterator::Deref (operator), 77
stream::iterator::Difference (operator), 77