

Translating Snort rules to STATL scenarios

Steven T. Eckmann

Reliable Software Group
Department of Computer Science
University of California
Santa Barbara, CA 93106
eckmann@cs.ucsb.edu

Abstract

The number of intrusion detection systems (IDSs) is large and growing. Most IDSs are signature based, which means that they include signatures for some collection of known attacks, and monitor an event stream looking for instances of any signature in their collection. There is an enormous duplication of effort within the IDS community, as each newly discovered attack requires independent specification for each IDS. Sharing of signature collections has obvious advantages for the IDS community as a whole, mainly by (1) allowing better allocation of scarce resources (developers and researchers) and (2) supporting peer review of signature collections, which can lead to better signatures and better detectors.

Snort is an IDS with a large published collection of signatures. This paper considers automated translation of Snort rules to STATL scenarios. Automatically translating Snort rules to STATL scenarios has the practical effect of allowing the use of Snort's large signature collection with NetSTAT sensors, with essentially no new work as new Snort signatures are developed.

A snort2statl translator has been developed that implements the described translation scheme. Most of the signature-specifying elements of Snort's rule language are easy to translate to STATL, but developing the translation scheme and its implementation, and then translating the complete set of rules in the standard Snort rule set into STATL scenarios, raised a few issues that are discussed.

Keywords: *Security, Intrusion detection, Networks, STAT, STATL, Snort.*

1 Introduction

The number of intrusion detection systems (IDSs) is large and growing. Most IDSs are signature based, which means that they include *signatures* for some collection of known attacks, and monitor an event stream looking for instances of any signature in their collection. Each signature-based IDS has its own mechanism for specifying signatures and loading them into the sensor. Consequently, there is an enormous duplication of effort within the IDS community, as each newly discovered attack requires independent specification for each IDS. Some of this duplication is by choice. For example, vendors of commercial IDSs typically are unwilling to share their signature collections because doing so is seen as giving away valuable, proprietary information. To the extent possible, however, sharing of signature collections has obvious advantages for the IDS community as a whole, mainly by (1) allowing better allocation of scarce resources (developers and researchers) and (2) supporting peer review of signature collections, which can lead to better signatures and better detectors.

Snort is an open-source, lightweight, network-based IDS for which a large collection of signatures has been developed and published [4, 5]. Snort's simple rule language supports matching single network packets, generating alerts or log messages, and responding or reacting to matched packets. Snort supports three protocols explicitly – TCP, UDP, and ICMP – implicitly supports the IP protocol, and also supports simple decoding of RPC requests. Snort enjoys wide popularity and is well supported by a large community.

This paper considers automated translation of Snort rules to STATL scenarios. STAT is a framework developed by the Reliable Software Group at UCSB for building IDSs [6]. The STAT framework includes a domain-independent attack description language called STATL [1]. The family of IDSs that have been built as STAT extensions includes a network-based system called NetSTAT [7]. NetSTAT includes support for the Snort-supported protocols and others (e.g., ethernet and ARP). Automatically translating Snort rules to STATL scenarios has the practical effect of allowing the use of Snort's large signature collection with NetSTAT sensors, with essentially no new work as new Snort signatures are developed.

The next section is an overview of the Snort rule language, with a discussion of how some example rules would be translated to STATL scenarios. Section 3 specifies how to translate Snort to STATL, defining a translation for each Snort rule element. Section 4 summarizes what was learned from this translation exercise, identifies inter-IDS translation issues that arose, and makes recommendations for signature sharing.

It should be noted that the translation being discussed in this paper is from Snort to NetSTAT, not Snort to (abstract) STATL. STATL provides no event types or predicates for specifying signatures over the TCP/IP domain. Those types and predicates are in the TCP/IP extension library that is an essential part of NetSTAT.

2 Snort rules

Each Snort rule has a *rule header* and *rule options*. The rule header contains the rule *action*, the *protocol*, source and destination *IP addresses* and *netmasks*, and source and destination *ports*. The rule options are predicates on packet fields, or the text of messages to be logged/displayed for matching packets, or response/reaction directives. An example rule is:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 \  
  (msg:"FTP passwd attempt";flags: A+; content:"passwd");
```

The rule header consists of the action keyword `alert` and everything else before the left parenthesis, and the parenthesized list contains the rule options.¹ This rule matches TCP packets from any external source IP address and port, to port 21 on the local network, containing the string `passwd` and having at least the ACK flag set. Whenever a matching packet is found, an alert is generated with the text “FTP passwd attempt”. This example will be seen again in section 2.4, where an equivalent STATL scenario is constructed.

2.1 The rule header

The general form of a Snort rule header is:

action protocol left_IP left_port direction right_IP right_port

The elements of a Snort rule header are defined as follows:

- The *action* is one of `alert`, `log`, `pass`, `activate`, or `dynamic`. The first generates an alert, the second generates a log record, and the third causes the packet to be passed with no alerts or log records. The last two provide a simple form of rule linking: an `activate` rule is like an `alert` rule that also activates a corresponding `dynamic` rule. The `dynamic` rule is like a `log` rule with a counter. In STATL terms, the `activate` rule would be a transition to an intermediate state s_1 and the `dynamic` rule would be a looping transition from s_1 to itself, with an exit transition based on a counter. An example is given in section 2.4, and details are in section 3.2.1.
- The *protocol* is one of `tcp`, `udp`, or `icmp`.
- An IP address is the keyword `any`, which matches any IP address; or a standard numeric IP address with a netmask in CIDR form (e.g., `192.168.1.13/24` specifies the class C network `192.168.1.0`); or a comma-separated list of IP addresses enclosed in square brackets (e.g., `[192.168.0.13/24,192.168.1.13/24]`). A negated IP address, `!address`, matches any address except the one specified. For example, `!192.168.1.0/24` matches any IP address that is *not* in the class C network `192.168.1.0`.
- Ports are specified by the keyword `any`, which matches any port; or a single number (e.g., `25` matches port `25`); or a range (e.g., `6000:6010` matches any port in the range `6000` to `6010`); or a negated port number or range (e.g., `!25` matches any port except `25`, and `!6000:6010` matches any port not in the range `6000` to `6010`); or a comma-separated list of port numbers (e.g., `3018,2600`).² Either the upper or the lower bound of a range may be left out (e.g., `:6000` is the range of ports less than or equal to `6000`).
- A direction operator between the left address and port and the right address and port specifies whether traffic matches in one direction or both directions. Single-direction matching is specified with “`->`” or “`<-`”,³ and bidirectional matching with “`<>`”.

¹Each Snort rule must be on a single line. The examples in this paper are split across lines for clarity.

²The port list syntax is used in the standard Snort rule set but undocumented.

³The direction operator `<-` is used in the standard Snort rule set but undocumented.

2.2 Rule options

Rule options are enclosed in parentheses and each option is terminated by a semicolon:

```
‘( option ‘;’ ... ‘)’
```

Rule options are either predicates on packet fields, or they are related to logging or responding to a match. Each option is of the form:

```
option-name ‘:’ option-value
```

The example on page 2 had `msg`, `flags`, and `content` options. Snort rule options are enumerated in Section 3.

2.3 Other Snort features

Snort includes other features for specifying signatures and handling detections, besides the rule language just described:

- Snort *variables* are constant configuration parameters. For example, many Snort rules refer to a variable `$HOME_NET`, which must be defined in an installation-specific configuration file. For example, given the definition:

```
var HOME_NET 192.168.1.0/24
```

all occurrences of `$HOME_NET` in rules refer to `192.168.1.0/24`.

- Snort *includes* provide a way to include files by reference within rule files. This capability is irrelevant to rule translation. That is, a translator should process included files, but that requirement does not affect the translation schemas defined in the next section.
- Snort *preprocessors* are invoked between the sniffer and the rule processor. Each preprocessor is hard-coded for a signature that cannot be expressed in the Snort rule language. Preprocessors are beyond the scope of Snort-to-STATL translation, so preprocessors are not discussed in this paper.
- Snort *output modules* support customization of log and alert outputs. In STAT, log and alert customization, and responses in general, are handled dynamically, separate from the STATL signature language. Since Snort output modules are related to reporting, not detection, they are not discussed in this paper.
- Snort *ruletypes* provide a macro-like shorthand for specifying one or more output modules to be associated with a set of rules. For example, after the definition:

```
ruletype suspicious
{
  type log
  output log_tcpdump: suspicious.log
}
```

it is possible to specify a rule:

```
suspicious udp any any -> $HOME_NET 5400 (msg:"Blade Runner");
```

as shorthand for:

```
output log_tcpdump: suspicious.log
log udp any any -> $HOME_NET 5400 (msg:"Blade Runner");
```

Using the ruletype facility allows parameterized specification of output configurations. Since that is outside the scope of rule translation, however, the only aspect of ruletypes that will be considered in this paper is that the action type of each user-defined ruletype must be extracted. In the example the action type was `log`.

2.4 Rule examples

This section presents example Snort rules and the STATL scenarios to which they would be translated. First is a typical alert rule from `ftp.rules`, already seen above on page 2:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 \  
  (msg:"FTP passwd attempt";flags: A+; content:"passwd");
```

A STATL representation of this signature is:

```
scenario ftp_19 ( ipAddrList EXTERNAL_NET, ipAddrList HOME_NET )  
{  
  transition t (s0->s1) nonconsuming  
  {  
    [IP ip [TCP tcp]] :  
      ip.header.src.match(EXTERNAL_NET)  
      && ip.header.dst.match(HOME_NET)  
      && (tcp.header.dst == 21)  
      && (tcp.header.flags & ACK)  
      && tcp.payload.contains("passwd",0,0,false)  
    { alert("FTP passwd attempt"); }  
  }  
  
  initial state s0 {}  
  state s1 {}  
}
```

The scenario name `ftp_19` is constructed, somewhat arbitrarily, from the Snort filename in which the rule appears and the rule’s linenummer in that file. The two Snort variable references are translated to STATL parameters, whose types are inferred from their use in the Snort rule. The `tcp` protocol in Snort is translated to a STATL *event spec* that specifies a TCP segment encapsulated within an IP datagram, named `tcp` and `ip`, respectively. The source and destination IP addresses and ports are translated to STATL predicates, as is each rule option that tests some property of the packet – “`flags: A+;`” and “`content: "passwd" ;`” in this example. These options are described in sections 3.3.2 and 3.3.5, respectively. The STATL `contains` function, described in section 3.3.5, takes a string, an offset, a length, and a flag indicating whether matching should ignore character case. The transition’s codeblock calls the STAT builtin function `alert` because the Snort rule type was `alert`, and the `msg` option string is used as the alert’s text.⁴

Most Snort rules have translations roughly like this example, though the `content` option often requires more sophisticated treatment than this, as discussed in section 3.3.5.

The same rule with a `log` action instead of an `alert` action would translate to a STATL scenario with the same event spec and transition assertion, but with a call to `log` instead of a call to `alert` in the transition’s codeblock.

The Snort `pass` action doesn’t translate to anything in STATL, so no example is given here (and no examples of its use occur in the standard Snort rule set). Snort’s `activate` and `dynamic` action types also are not used in the standard Snort rule set, but an example is given in the manual:

```
tcp !$HOME_NET any -> $HOME_NET 143 \  
  (flags: PA; content: "|E8C0FFFFFF|\bin"; activates: 1; msg:"IMAP buffer overflow");  
dynamic tcp !$HOME_NET any -> $HOME_NET 143 (activated_by: 1; count: 50;)
```

This pair of rules tells Snort to generate an alert when it detects an attempt by a nonlocal host to exploit an IMAP buffer overflow on a local host, then log the next 50 packets from outside to any inside host’s imap port (143). A STATL representation of this signature is:

```
scenario test_rules_1 ( ipAddrList HOME_NET )  
{  
  int count=0;  
  
  transition activate (s0->s1) nonconsuming  
  {  
    [IP ip [TCP tcp]] :
```

⁴Generating an alert is not actually part of the attack signature, so this part of the translation could be made optional.

```

        !ip.header.src.match(HOME_NET)
        && ip.header.dst.match(HOME_NET)
        && (tcp.header.dst == 143)
        && (tcp.header.flags == (PSH | ACK))
        && tcp.payload.contains("|E8C0FFFFFF|\bin",0,0,false)
    { alert("IMAP buffer overflow"); }
}

transition dynamic (s1->s1) consuming
{
    [IP ip [TCP tcp]] :
        count < 50
        && !ip.header.src.match(HOME_NET)
        && ip.header.dst.match(HOME_NET)
        && (tcp.header.dst == 143)
    { ++count; log("IMAP buffer overflow"); }
}

transition exit (s1->s2) consuming
{
    [IP ip [TCP tcp]] :
        count >= 50
        && !ip.header.src.match(HOME_NET)
        && ip.header.dst.match(HOME_NET)
        && (tcp.header.dst == 143)
}

initial state s0 {}
state s1 {}
state s2 {}
}

```

The STATL version has three transitions: the `activate` transition is constructed as for an `alert` rule. The `dynamic` transition is constructed as for a `log` rule, except that it loops and increments a counter. The `exit` transition exits the scenario instance after the specified number of matching packets have been detected.

3 Detailed translation rules

This section defines how each Snort rule element is translated to an equivalent STATL language element.

3.1 Variables

Simple variable references of the form `$NAME` can be replaced by their values at translation time if those values are available, but a better approach is to replace Snort variable references with STATL parameter references, and ensure that the corresponding parameter values appear in NetSTAT's Factbase. Snort supports two other types of variable reference, for providing a default value, and for aborting execution if no value is defined:

- `$var : -default` is replaced with the value of `var` if it is defined, otherwise with `default`.
- `$var : ?message` is replaced with the value of `var` if it is defined, otherwise Snort prints the provided error message and exits.

The default style can be translated to a STATL parameter declaration with a default value. It isn't obvious what is the best way to handle the runtime abort style. Only simple references are currently used in the standard Snort rule set.

3.2 Rule header

3.2.1 Action

The Snort rule actions `alert` and `log` indicate rules that are translated to single-transition scenarios. The only difference between them is that one produces an `alert` call in the transition's codeblock, and the other produces a `log` call. In

general, the scenario produced from an alert rule is:

```
scenario <name>_<number> ( <params> )
{
  transition t (s0->s1) nonconsuming
  {
    <event_spec> :
      <assertion>
    { alert(<msg>); }
  }

  initial state s0 {}
  state s1 {}
}
```

where the bracketed items (i.e., items enclosed in '<' and '>') depend on other parts of the rule header and on rule options.

The Snort `pass` action causes matching packets to *not* generate alerts or log messages. The inclusion of `pass` in Snort suggests that rules are evaluated in the order in which they are read into the sensor, and a `pass` rule causes subsequent rules to not be evaluated. That is, ordinarily all rules would be applied to each packet. When a `pass` rule matches, the IDS stops its traversal of the rule set for that packet. The semantics of STATL includes neither ordering of signature evaluation nor direct short-circuiting of the rule set traversal, so nothing in STATL provides the semantics of Snort `pass`. There are no `pass` rules in the current standard Snort rule set, however, so its implementation can safely be deferred.

The Snort `activate` and `dynamic` rule actions must come in pairs: the first specifies option `activate: k` and the second specifies options `activated_by: k` and `count: n`. In general, given an `activate/dynamic` pair:

```
activate ... (... activates: k; ...)
dynamic ... (activated_by: k; count: n;)
```

the scenario produced is:

```
scenario <name>_<number> ( <params> )
{
  int count=0;

  transition activate (s0->s1) nonconsuming
  {
    <event_spec_activate> :
      <assertion_activate>
    { alert(<msg>); }
  }

  transition dynamic (s1->s1) consuming
  {
    <event_spec_dynamic> :
      count < n
      && <assertion_dynamic>
    { ++count; log(<msg>); }
  }

  transition exit (s1->s2) consuming
  {
    <event_spec_dynamic> :
      count >= n
      && <assertion_dynamic>
  }

  initial state s0 {}
  state s1 {}
  state s2 {}
}
```

where again the bracketed items depend on other parts of the rule headers and on rule options. An alternative translation that was considered ignores the `dynamic` rule:

```

scenario <name>_<number> ( <params> )
{
  transition activate (s0->s1) nonconsuming
  {
    <event_spec_activate> :
      <assertion_activate>
      { alert(<msg>); }
  }

  initial state s0 {}
  state s1 {}
}

```

This alternative was rejected because the signature parts (i.e., the protocol, IP addresses, ports, and options related to matching) might differ between an `activate` rule and its companion `dynamic` rule, in which case the logging loop is reporting different kinds of packets than those selected by the `activate` rule.

3.2.2 Protocol

Each Snort rule specifies one of three protocols: `tcp`, `udp`, or `icmp`, which translate to the following three STATL event specs:

- `tcp` translates to `[IP ip [TCP tcp]]`
- `udp` translates to `[IP ip [UDP udp]]`
- `icmp` translates to `[IP ip [ICMP icmp]]`

The IP encapsulation of the specified protocol is needed in STATL because many IP datagram fields, most commonly source and destination addresses, are referred to in Snort rules. The STATL spec cannot refer to those fields without a name for the IP event. The Snort protocol also determines which event (`tcp` or `udp`) is used to look up ports, as described in section 3.2.4 below.

3.2.3 IP addresses

In Snort rules an IP address is the keyword `any`, or a numeric IP address and netmask, or a negated IP address and netmask, or a bracketed list of IP addresses. IP address specifications in Snort rules become conditions in a transition assertion. Translations are as follows:

- `any`
Indicates that there are no restrictions on the IP address, so no condition is generated.

- numeric IP address and netmask (as CIDR block)

Translate:

```
dot4address/cidr
```

to:

```
ip.header.src.match("dot4address/cidr")
```

if the address appears as the source address, similarly for the destination address.⁵ The `match()` method returns true if the given string specifies a host or net address that matches the corresponding TCP field.

- negated IP address and netmask
Translate the IP address, then put “!” in front of it.

⁵Whether an address-port pair indicate a source or destination depends on the direction specifier and the pair’s position (left or right side of the direction specifier). See section 3.2.5.

- IP address list

Translate:

```
[dot4address_1/cidr, ..., dot4address_n/cidr]
```

to:

```
( ip.header.src.match("dot4address_1/cidr")
  || ...
  || ip.header.src.match("dot4address_n/cidr"))
```

if the address list appears as the source address, similarly for the destination address.

3.2.4 Ports

A Snort port number is the keyword `any`, or a single number, or a range, or a negated port number or range, or a list of port numbers. Assuming the protocol is `tcp`, translations are as follows, with the obvious changes for `udp`:

- `any`

Indicates that there are no restrictions on the port, so no condition is generated.

- single number

Translate:

```
portnum
```

to:

```
(tcp.header.src == portnum)
```

if the port appears as the source port, similarly for the destination port.

- range with both ends specified

Translate:

```
low:high
```

to:

```
(tcp.header.src >= low) && (tcp.header.src <= high)
```

if the port range appears as the source port, similarly for the destination port.

- range with upper bound specified

Translate:

```
:high
```

to:

```
(tcp.header.src <= high)
```

if the port range appears as the source port, similarly for the destination port.

- range with lower bound specified

Translate:

```
low:
```

```
to:
```

```
(tcp.header.src >= low)
```

if the port range appears as the source port, similarly for the destination port.

- portnum list

Translate:

```
portnum_1, ..., portnum_n
```

```
to:
```

```
( (tcp.header.src == portnum_1)
  || ...
  || (tcp.header.src == portnum_n) )
```

if the port range appears as the source port, similarly for the destination port.

- negated port spec

Translate the port spec, then put “!” in front of it.

3.2.5 Direction

A direction operator between the left address and port and the right address and port specifies whether traffic matches in one direction or both directions. Single-direction matching is specified with “->” (source on the left, destination on the right) or “<-” (source on the right, destination on the left), and bidirectional matching with “<>”. The bidirectional operator means that the rule can match with left as source and right as destination, or with left as destination and right as source. Therefore addresses and ports are translated as disjunctions of the two cases. That is,

```
addr_left port_left <> addr_right port_right
```

is translated to:

```
( (ip.header.src == addr_left)
  && (tcp.header.src == port_left)
  && (ip.header.dst == addr_right)
  && (tcp.header.dst == port_right) )
|| ( (ip.header.dst == addr_left)
     && (tcp.header.dst == port_left)
     && (ip.header.src == addr_right)
     && (tcp.header.src == port_right) )
```

and so on.

3.3 Rule options

3.3.1 Options related to IP datagram fields

The Snort IP options are `tTL`, `tos`, `id`, `ipoption`, `fragbits`, and `dsize`. Translations are as follows:

- `tTL:n` is translated to `ip.header.ttl == n`, and `tTL:>n` is translated to `ip.header.ttl > n`.⁶
- `tos:n` is translated to `ip.header.tos == n`.
- `id:n` is translated to `ip.header.id == n`.

⁶The second syntax is undocumented but used in the standard Snort rule set.

- `ipoption:v`, where v is one of `rr`, `eol`, `nop`, `ts`, `sec`, `lsrr`, `ssrr`, `lsrre`, or `satid`,⁷ is translated to `ip.option.v'`, where v' is the `tcpiplib` method name corresponding to v . For example, `ipoption:lsrr` is translated to `ip.option.hasLooseSourceRoute()`.
- The `fragbits` option is of the form: `fragbits:bits`, where $bits$ is one or more of `R` (Reserved bit), `D` (Don't Fragment bit), or `M` (More Fragments bit), optionally followed by one of the modifiers `'+'` (match if all the specified flags are set), `'*'` (match if any of the specified flags are set), or `'!'` (match if all the specified flags are not set). If none of the modifiers appears, then translate:

```
fragbits:bits
```

to

```
ip.header.frag == bits'
```

where $bits'$ is the bitwise-or of the STATL flags corresponding to the Snort flags.

If the `'+'` modifier is used, then translate:

```
fragbits:bits+
```

to

```
ip.header.frag & bits'
```

where $bits'$ is the bitwise-or of the STATL flags corresponding to the Snort flags.

If the `'*'` modifier is used, then translate:

```
fragbits:bits*
```

to

```
(ip.header.frag & bits'_1) || ... || (ip.header.frag & bits'_n)
```

where $bits'_j$ is the STATL flag corresponding to the j^{th} Snort flag in $bits$.

If the `'!'` modifier is used, then translate:

```
fragbits:bits!
```

to

```
!(ip.header.frag & bits'_1) && ... && !(ip.header.frag & bits'_n)
```

where $bits'_j$ is the STATL flag corresponding to the j^{th} Snort flag in $bits$.

- `dsize:n` is translated to `ip.header.length - ip.header.header.length == n`. If the Snort option includes `'<'` or `'>'`, then the STATL operator will be the same as that Snort operator, instead of `"=="`.

3.3.2 Options related to TCP fields

The Snort TCP options are `flags`, `seq`, `ack`, and `session`. Translations are as follows:

- The `flags` option is of the form: `flags:bits`, where $bits$ is one or more of `F` (FIN), `S` (SYN), `R` (RST), `P` (PSH), `A` (ACK), `U` (URG), `2` (Reserved bit 2), `1` (Reserved bit 1), or `0` (undocumented but used in the standard rule set; presumably means no flags are set), optionally followed by one of the modifiers `'+'` (match if all the specified flags are set), `'*'` (match if any of the specified flags are set), or `'!'` (match if all the specified flags are not set). If none of the modifiers appears, then translate:

```
flags:bits
```

to

```
tcp.header.flags == bits'
```

where $bits'$ is the bitwise-or of the STATL flags corresponding to the Snort flags.

If the `'+'` modifier is used, then translate:

⁷The `lsrre` value is undocumented but used, and `any` is undocumented and unused, although it is in the Snort rule parser.

```
flags:bits+
```

to

```
tcp.header.flags & bits'
```

where *bits'* is the bitwise-or of the STATL flags corresponding to the Snort flags.

If the '*' modifier is used, then translate:

```
flags:bits*
```

to

```
(tcp.header.flags & bits'_1) || ... || (tcp.header.flags & bits'_n)
```

where *bits'_j* is the STATL flag corresponding to the *j*th Snort flag in *bits*.

If the '!' modifier is used, then translate:

```
flags:bits!
```

to

```
!(tcp.header.flags & bits'_1) && ... && !(tcp.header.flags & bits'_n)
```

where *bits'_j* is the STATL flag corresponding to the *j*th Snort flag in *bits*.

- *seq:number* is translated to `tcp.header.th_seq == number`.
- *ack:number* is translated to `tcp.header.th_ack == number`.
- The `session` option specifies that application data from a TCP session should be captured to the log file. It is not used for specifying signatures, and is therefore beyond the scope of translation to STATL.

3.3.3 Options related to UDP fields

There are no UDP-specific Snort options.

3.3.4 Options related to ICMP fields

The Snort ICMP options are `itype`, `icode`, `icmp_id`, and `icmp_seq`. Translations are as follows:

- *itype:number* is translated to `icmp.header.type == number`.
- *icode:number* is translated to `icmp.header.code == number`.
- *icmp_id:number* is translated to `icmp.header.id == number`.
- *icmp_seq:number* is translated to `icmp.header.seq == number`.

3.3.5 Options for string matching

The Snort options related to string matching within packet payloads are `content`, `offset`, `depth`, and `nocase`. The `content` option specifies the string to be matched. Within the string, '|' characters delimit hexadecimal data, separating it from character data. The other three Snort options modify how the string is matched: `offset` specifies where in the payload to start searching; `depth` specifies a maximum length for the search region; `nocase` makes the search case-insensitive.

STATL has no built-in string-matching support, and NetSTAT's `tcpiplib` extension provides regex string matching. It is not clear whether functions to support matching like that provided by Snort's `content` option and related options should be part of STATL and the STAT core, or part of the `tcpiplib` extension used by NetSTAT, or neither. A tentative decision has been made to translate `content` and its modifiers as follows:

```
content:string; offset:n; depth:m;
```

is translated to

```
tcp.payload.contains(string, n, m, nocase?)
```

where *nocase?* is true if the Snort *nocase* option appears, otherwise false, *n* is taken to be 0 if no *offset* option appears, and *m* is taken to be 0 if no *depth* option appears (indicating no limit). The rule's protocol determines whether the translation of a *content* option refers to `tcp.payload` or `udp.payload`.

Another Snort string-matching option is *content-list*, the value of which is a filename containing a list of strings to be used like a sequence of *content* options, but more conveniently stored. The *content-list* option is used only with the *react* option, which is used to close connections, such as connections to disallowed websites. No attempt has been made to translate *content-list* options, mainly because this sort of filtering is typically provided by firewalls.

3.3.6 Other options to be translated to conditions

The *rpc* option tests the application, procedure, and program version from an RPC request. For RPC requests encapsulated within TCP packets, translate:

```
rpc:program,procedure,version;
```

to

```
[IP ip [TCP tcp [RPC rpc]]] :  
  (rpc.header.program == program)  
  && (rpc.header.procedure == procedure)  
  && (rpc.header.version == version)
```

and similarly for RPC requests encapsulated in UDP packets.

3.3.7 Output options

The Snort output options are *msg* and *logto*. The *msg* option specifies a string to be used in alerts and log messages, and is copied verbatim to the corresponding STATL alert or log string. The *logto* option specifies a filename to which a matching packet should be logged, instead of standard output. It is not translated to anything, since it is not part of the signature and there is no obvious translation.

3.3.8 Response and reaction options

The Snort options for responding or reacting to matching packets are *resp* and *react*. Each option specifies one or more fixed responses, such as sending RST packets. Responses are not part of the STATL language since they do not describe attack scenarios, so Snort response options are not translated to STATL. However, the STAT core deals explicitly with responses, which can be dynamically loaded and unloaded.

4 Conclusions and future work

This paper has described a scheme for translating Snort rules to equivalent STATL scenarios. The primary objectives of the work were to compare two ways of representing network attack signatures, and to leverage the ongoing work done by the Snort community in building signatures. A `snort2statl` translator has been developed that implements the described translation scheme. The complete standard Snort ruleset – the rule database available at `www.snort.org` – was translated to STATL scenarios. This ruleset contains nearly 1000 signatures in 23 files. Most of the signature-specifying elements of Snort's rule language are easy to translate to STATL, but developing the translation scheme and its implementation and translating the complete set of rules into STATL scenarios, raised a few issues:

- There are several instances of “families” of Snort rules (e.g., those in `ftp.rules`) that differ only in the string that they match using a *content* option. Searching for packets that match members of such families is obviously more efficient if the common tests are done only once, and then the *content* options are checked. It isn't clear whether Snort does this sort of rule set optimization internally. One-to-one translation of such families from Snort to STATL produces a great deal of redundancy that the STAT core is not designed to optimize away internally. A hand-constructed STATL signature set would typically represent such a family with a single scenario having a list of strings to match (where this list would be a parameter).

- The Snort rule set contains duplication between TCP and UDP – some Snort rules are identical except the protocol: `tcp` or `udp`. That is, no field of either protocol is tested, and the protocol is therefore irrelevant to the abstract signature. In such cases a single NetSTAT scenario could be used for the pair. Currently no attempt is made to eliminate the redundancy.
- STATL and the STAT core include nothing like the semantics of Snort’s `pass` action. It isn’t clear that such a feature is compatible with STAT semantics, or that it would be useful.

This paper has reported on very preliminary work. We plan to complete the implementation of the `contains` string-matching function, then experiment with Snort rules as STATL scenarios, with the expectation that we will routinely incorporate Snort signatures into NetSTAT sensors with no hand translation. We also plan to conduct similar experiments with other signature languages for other IDSs, such as P-BEST (Emerald [2]) and N-code (NFR [3]), and to investigate translation in the other direction, from STATL scenarios to signatures for other IDSs. These efforts should lead to improved insight into requirements for and the value of signature sharing between IDSs.

Acknowledgments

Professors Richard A. Kemmerer and Giovanni Vigna at UCSB provided valuable comments on Snort-to-STATL translation issues, suggested that I turn a rough note into this paper, and also helped with editing of the paper.

This research was supported by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-1-0207, and by the National Security Agency’s University Research Program, under agreement number MDA904-98-C-A891. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Rome Laboratory, the National Security Agency, or the U.S. Government.

References

- [1] S. Eckmann, G. Vigna, and R. Kemmerer. STATL: An attack language for state-based intrusion detection. In *Proceedings of WIDS (held in conjunction with ACMCCS 2000)*, Athens, Greece, November 2000.
- [2] P.G. Neumann and P.A. Porras. Experience with EMERALD to Date. In *1st USENIX Workshop on Intrusion Detection and Network Monitoring*. USENIX, April 1999.
- [3] M.J. Ranum, K. Landfield, M. Stolarchuck, M. Sienkiewicz, A. Lambeth, and E. Wall. Implementing a Generalized Tool for Network Monitoring. In *Eleventh Systems Administration Conference (LISA '97)*. USENIX, October 1997.
- [4] M. Roesch. *Writing Snort Rules: How To write Snort rules and keep your sanity*. <http://www.snort.org>.
- [5] M. Roesch. Snort – lightweight intrusion detection for networks. In *Proceedings of USENIX LISA '99*, November 1999.
- [6] G. Vigna, S. Eckmann, and R. Kemmerer. The STAT Tool Suite. In *Proceedings of DISCEX 2000*, Hilton Head, South Carolina, January 2000. IEEE Computer Society Press.
- [7] G. Vigna and R.A. Kemmerer. NetSTAT: A Network-based Intrusion Detection System. *Journal of Computer Security*, 7(1):37–71, 1999.