Fast Packet Classification for Snort by Native Compilation of Rules

Alok Tongaonkar, Sreenaath Vasudevan, and R. Sekar – Stony Brook University

ABSTRACT

Signature matching, which includes packet classification and content matching, is the most expensive operation of a signature-based network intrusion detection system (NIDS). In this paper, we present a technique to improve the performance of packet classification of Snort, a popular open-source NIDS, based on generating native code from Snort signatures.[1] An obvious way to generate native code for packet classification is to use a low-level language like C to access the contents of a packet by treating it as a sequence of bytes. Generating such low-level code manually can be cumbersome and error prone. Use of a high-level specification language can simplify the task of writing packet classification code. Such a language needs features that minimize the likelihood of common errors as errors in the packet processing code can crash the intrusion detection system, which may leave it open to attacks.

To overcome these problems, we use a rule-based specification language with a type system for specifying the structure and contents of packets. The compiler for the specification language generates C code for packet classification. This code can be compiled into native code using a C-compiler and loaded into Snort as shared library. Our experiments using real and synthetic traces show that the use of native code results in a speedup of the packet classification of Snort up to a factor of five.

Introduction

Recent years have seen a rapid escalation of security threats making Network Intrusion Detection Systems (NIDS) critical components of modern network infrastructure. A NIDS inspects each packet for a match against a large signature set. The NIDS must work at near wire speed to be effective in an online analysis mode. Typical NIDS perform a number of operations upon receiving a packet like buffering the packet, matching the packet against signature set, and logging packets or alerts. The performance of each of these operations affects the performance of the system as a whole. Even so, the signature matching component remains one of the most important factors that determines the performance of these systems.

Generally, the signature matching used in IDS consists of two distinct operations: i) *packet classification*, which involves examining the values of packet header fields, and ii) *deep packet inspection*, in which the packet payload is matched against a set of predefined patterns. According to [3], packet classification and deep packet inspection are the most expensive parts of Snort (a popular open source IDS) [10], accounting for 21% and 31% of the execution time. The signatures for Snort-like systems are usually specified using simple rule-based language. Typically, the IDS uses an interpreter to check whether any rule matches an incoming packet.

A lot of research has focused on improving the performance of signature matching component of Snort. Most of the research has focused on deep packet inspection which involves string matching and regular expression matching. Current versions of Snort (i.e., starting with version 2.0) use an improved detection engine that matches strings in parallel. Snort uses many efficient and high-speed string matching algorithms like Aho-Corasick [1] and Wu-Manber [13] to match strings in parallel. If the string match succeeds for certain rules, then the packet header fields in thoses rules are checked sequentially. Snort uses Perl Compatible Regular Expression (PCRE) library for checking regular expressions. The regular expressions are also checked sequentially for the rules for which string matching has succeeded.

In this paper, we look at the problem of improving the performance of packet classification used in Snort. We use a technique based on generating native code for packet classification. Use of native compilation to speed up programs is a common technique used in programming language domain. For example, Johansson and Jonsson [5] have shown how simple native compilation can increase the speed of Erlang programs. Native code for packet classification can be generated by writing C code which understands the grammar of packet formats, and performs necessary byte-order and alignment adjustments. However, writing such C code is cumbersome and error-prone. Moreover, maintaining such low-level code is tedious as making even a small change to the rule set may involve making lot of changes in the C code. Making frequent large-scale changes to the low-level C code makes it difficult to have high confidence that the code is performing signature matching correctly.

[1]This research is supported in part by ONR grant N00014 0710928 and NSF grant CNS-0627687.

To overcome this problem, we use a high-level specification language with a special type-system for packet processing to specify Snort rules. The compiler for the language generates C code which can be compiled using common C compiler like GCC to get native code for packet classification. We provide a translator to convert existing Snort rules to a specification in this language. This way system administrators need not learn the high-level specification language and can continue specifying rules in the Snort rule language.

In the Snort Overview section, we discuss Snort's detection engine. The Packet Classification Code section discusses different approaches to generate packet classification code. We describe the type system used to generate native code in the Specification Language section. The implementation details are provided in the Implementation section which is followed by Evaluation, Related Work, and the Conclusion.

Snort Overview

In this section we discuss the rule language of Snort and the signature matching scheme used in Snort.

Snort Language

Snort uses a simple rule-based language to specify signatures. Snort signatures are written in a configuration file which is read when Snort starts up. A Snort signature file consists of variable declarations and rules. The variable declarations are similar to typedefs in C; the value of the variable is substituted in the rules for signature matching. The rules themselves consists of a *rule header* and a *rule body*.

The rule header consists of *action*, *protocol*, *ip addresses*, *ports*, and *direction operator*. Rule actions specify the action like logging or alerting that Snort should perform when a rule matches a packet. Each rule is applicable to packets belonging to a particular protocol like TCP, UDP, ICMP, or IP. For TCP and UDP rules, the header specifies the source and destination ip addresses and port fields for which the rule is to be applied. Specifying *any* for one of these fields means that the field in the rule matches for any value in a packet. The fields to the left of the direction operator (→) are the source fields, while the ones on the right hand side are for the destination. An alternative operator, called bidirectional operator (<>), indicates that the rule is to be applied to both directions of the flow. Consider the following variable declaration and rule:

```
var internal_host 192.168.2.0/24
alert tcp $internal_host any -> 192.168.1.1 80
```

"internal_host" is a variable whose value is the host address 192. 168. 2. 0 with subnet mask of 24 bits. So any host with this subnet address matches internal_host variable. This rule generates an alert when it sees a tcp packet from any port on an internal host to host 192. 168. 1. 1 on port 80.

Rule body consists of rule options which belong to one of the following categories: i) *meta-data* options provide information about the rule but are not used in signature matching operation, ii) *payload* options are concerned with tests for deep packet inspection, iii) *non-payload* options specify other tests including tests on packet header fields, and iv) *post-detection* options specify some triggers which are fired when a rule matches a packet. Consider the rule body appended to the previous Snort rule:

```
var internal_host 192.168.2.0/24
alert tcp $internal_host any ->
  192.168.1.1 80 (msg:''web-attack'';
  ttl: 5; content: ''abc'';
  logto: ''logfile'';)
```

In this rule, *msg* is a meta-data option that specifies the message to be generated when a packet matches this rule. *logto* is a post-detection option that specifies the file to be used for logging. *content* is a payload option which means that the rule is matched by a packet only if the payload contains the string "abc". Further, the packet has to satisfy the constraint that *ttl* field value is equal to 5 for the rule to match.

Snort Detection Engine

Starting with version 2, Snort uses an improved detection engine for matching signatures. The rules are first grouped based on the protocol field. Thus, all rules are put in one of the groups corresponding to TCP, UDP, ICMP, and IP. The rules are further grouped based on certain fields – *source* and *destination ports* for TCP and UDP, *type* for ICMP, and *protocol* for IP rules. For each group, the strings specified by rules in the group are combined to form an Aho-Corasick automaton. The Aho-Corasick automaton is a deterministic finite automaton that can be used to match the payload against multiple strings in parallel.

When Snort receives a packet, it identifies the group to which the packet belongs. Then the payload of the packet is matched against the Aho-Corasick automaton corresponding to that group. Aho-Corasick algorithm identifies all the rules whose *content option* is matched. For each of these rules, an interpreter checks whether the other payload and non-payload options are satisfied by the packet. If all the options of a rule are satisfied, then a match is announced for that rule.

Packet Classification Code

Signature matching operation consists of two main components: i) *packet classification*, and ii) *deep packet inspection*. Packet classification is the problem of identifying the rules matching a packet based on the values in the packet header fields. The performance of packet classification can be improved by using native code instead of interpreted code. Native code can be generated from packet classification code written in a low-level language like C. The most straight forward way to write such code is by treating the packet as a

sequence of bytes. There are many problems with this approach. To access any field of the packet, the offset of that field from the start of the byte sequence has to be calculated. This way of accessing fields with offset calculations has many potential pitfalls. For example, to access the source port field of tcp header, one needs to first ensure that the packet is a tcp packet. The offset for tcp source port depends on the length of the variable-length options field of ip header. Also, the bytes at those offsets need typecasting to `unsigned short` and conversion to the host order. It is clear that writing such code is very tedious and error-prone.

A better approach is to overlay the packet header structure on the byte sequence and then access packet header fields as fields of the structure. Even this approach does not solve the problem completely due to the presence of variable length fields and the need to perform protocol decoding before accessing any field. Another approach is to use a special language developed explicitly for packet processing. Such a language can have a hand-crafted type checker for particular network protocols or have a generic type checker that supports different network protocols. In the former approach, the packet structure for supported protocols are hard coded into the compiler. This approach is very rigid and supporting new protocols requires modification to the compiler. We use the latter approach which is more flexible and extensible.

In [11] we presented a special type system that can capture packet structures while providing the capabilities to dynamically identify packet types at runtime. In the next section we describe the features of the high-level language that uses that type system.

Specification Language

The language for specifying packet classifiers is rule-based. Specifications consist of variable and type declarations, followed by a list of rules. In the following sections we describe each of these components of specifications.

Packet Structure Description

The structure of the packets has to be specified using packet type declarations before specifying the rules. The syntax of type declaration for packets is similar to that of the C-language. For example, Listing 1 describes an Ethernet header.

The nested structure of protocol header can be captured using a notion of inheritance. For example, an IP header can be considered as a sub-type of Ethernet header with extra fields to store information specific to IP protocol. The specification language permits multilevel inheritance to capture protocol layering. Inheritance is augmented with constraints to capture conditions where the lower layer protocol data unit (PDU) has a field identifying the higher layer data that is carried over the lower layer protocol. For instance, IP header derives from Ethernet header only when `e_type` field in the Ethernet header equals `0800h`; see Listing 2.

To capture the fact the same higher layer data may be carried in different lower layer protocols, the language provides a notion of disjunctive inheritance. The semantics of the disjunctive inheritance is that the derived class inherits fields from exactly one of the possibly many base classes. Listing 3 shows a specification that IP may be carried within an Ethernet or a token ring packet.

```
#define ETHER_LEN 6
struct ether_hdr {
    byte e_dst[ETHER_LEN];       /* Ethernet destination address */
    byte e_src1[ETHER_LEN];      /* Ethernet source address */
    short e_type;                /* Protocol of carried data */
};
```

Listing 1: Ethernet header description.

```
#define ETHER_IP 0x0800
struct ip_hdr : ether_hdr with e_type == ETHER_IP {
    bit          version[4];     /* IP Version */
    bit          ihl[4];         /* Header Length */
    byte         tos;            /* Type Of Service */
    short        tot_len;        /* Total Length */
    ...
    short        check_sum;      /* Header Checksum */
    unsigned int s_addr;         /* Source IP Address Bytes */
    unsigned int d_addr;         /* Destination IP Address Bytes */
};
```

Listing 2: IP header with inherited Ethernet header.

```
struct ip_hdr : (ether_hdr with e_type == ETHER_IP) or
                (tr_hdr with tr_type == TOKRING_IP) {
    ...
}
```

Listing 3: Headers for Ethernet and Token Ring.

We can declare a variable of type *ether_hdr* and access various packet fields by using the fields in the respective structure. For example, to access the source port of a tcp packet, we declare a variable corresponding to packets as follows:

```
ether_hdr p;
```

Now, *p.tcp_sport* stands for source port of a tcp packet.

Rules

The rules are of the form "*cond* → *actn*", where *actn* specifies the action to be taken on a packet that matches the condition *cond*. The *cond* is a conjunction of tests on packet fields. The language supports various tests like equality, disequality, and inequality along with bit-masking operations on packet fields. A packet matches a rule if all tests in the rule succeed. If multiple rules match at the same time, actions associated with each rule are launched. Consider the rule in Listing 4. The first test in the rule is equivalent to checking whether the source address of the packet belongs to 192.168.2.0/24. Here, `0xc0a80200` is the hex representation of 192.168.2.0 and `0xffffff00` corresponds to the 24-bit subnet mask. This rule further checks that destination address is `0xc0a80100` (192.168.1.0) and destination port is 80. This rule is equivalent to the Snort rule shown in the Snort Language section.

The compiler generates a C function for this rule set which takes a network packet (as byte sequence) as input, performs the matching, and returns the rules that match. The packet matching code contains appropriate offset calculation, byte alignment, and order adjustment code.

Constraint Checking

An important requirement for the language to be type safe is that the constraints must hold before the fields corresponding to a derived type are accessed. Note that at compile time the actual type of the packet is not known. For example, a packet on an Ethernet interface must have the header given by *ether_hdr*. But it is not known whether the packet carries an ARP or an IP packet. So the constraint associated with *ip_hdr* must be checked at runtime before accessing the IP-relevant fields. Similarly, before accessing TCP relevant fields, the constraints on *tcp_hdr* must be checked. Furthermore, the constraints on *ip_hdr* must be checked before checking constraints on *tcp_hdr*.

The compiler automatically inserts the appropriate constraints before each field test using the packet structure specification (described in the Packet Structure Description section). For example, the compiler automatically transforms the previous rule to that shown in Listing 5.

Here, the test to the left of `:` is the precondition that needs to be satisfied before the test on the right can be performed. For the first test in the rule, the compiler figures out that *s_addr* is a field in the *ip_hdr* structure. So it adds the constraint on *ip_hdr*, i.e., *e_type* ≡`0x800`, to this test as precondition. Before accessing tcp fields, the constraints on *ip_hdr* and *tcp_hdr* need to be satisfied. As shown in the test on *tcp_dport* in this example, the compiler adds these constraints as a list. Note that the compiler adds these constraints in the order defined by the inheritance chain of packet structures.

Implementation

We implemented a Perl based translator for converting Snort rules into a specification for our language. The translator generates the packet structure specification and generates a rule in the specification for each rule in a Snort rule file. So there is a one-to-one correspondence between the rules in the Snort rules file and the rules in our specification file. The rules in our specification contain only the tests on packet header fields. The other tests in the rules are checked by using the detection engine of Snort itself.

For each non-payload detection option of Snort rules we generate the corresponding packet field test in our language. For example, consider the following rule in Snort,

```
alert tcp $EXTERNAL_NET any
    -> INTERNAL_NET $HTTP_PORT
    (..., ttl: 5; ...)
```

This rule generates alerts for tcp packets with *ttl* field of 5 from external network to internal network on

```
R1: (p.s_addr & 0xffffff00 == 0xc0a80200) &&
    (p.d_addr == 0xc0a80100) &&
    (p.tcp_dport == 80) -> alert(R1);
```

Listing 4: Sample packet matching rule.

```
R1: (p.e_type == 0x800):(p.s_addr & 0xffffff00 == 0xc0a80200) &&
    (p.e_type == 0x800):(p.d_addr == 0xc0a80100) &&
    (p.e_type == 0x800):(p.proto == 0x11):(p.tcp_dport == 80) -> alert(R1);
```

Listing 5: Transformed R1.

```
R1: (p.proto == tcp) && (p.s_addr == $EXTERNAL_NET) &&
    (p.d_addr == $INTERNAL_NET) && (p.tcp_dport == $HTTP_PORT) &&
    (p.ttl == 5) -> alert(R1)
```

Listing 6: Alerts for tcp packets with *ttl* field of 5.

port for http (80). The corresponding rule in our specification language is shown in Listing 6.

We use our compiler to compile the Snort rules in our specification format into C code. The compiler generates a backtracking automaton that matches each rule sequentially. Then it generates the C-code for matching this automaton in a straight-forward way using *if-then-else* branching. We use C compiler like *gcc* to generate native packet classification code in the form of a shared library. So to update the rules, all that one needs to do is to compile the rules offline and then reload the shared library. We note that this approach is no more disruptive than that of Snort where the rules need to be re-read and recompiled.

We load the shared library containing the packet classification code when Snort starts up. At runtime, when a packet is delivered to Snort by *pcap library*, we pass on the packet to the shared library. The shared library matches the packet against all the rules and returns the rules that match. At this point control is transferred to the default Snort detection engine. From this point on, the usual Snort processing (like logging) is performed on the packet.

We note that using this approach does not modify the behavior of Snort. In particular, for any packet the modified Snort matches the same rules as the original Snort. This is because we are just changing the way packet classification is performed without changing the actual tests in a rule.

Evaluation

We evaluated the effectiveness of the proposed technique using Snort 2.6.1.5. Our experiments were performed on a system with 3.06 GHz Intel Xeon processor and 3 GB memory, running Fedora Core 5 (Linux kernel 2.6.15). To understand the impact of native compilation of Snort rules we used the default signatures that come with two different versions of Snort – Snort-1.8 and Snort-2.6. Snort-1.8 uses a slower scheme for matching packet fields where each rule is checked sequentially without first grouping the rules. To compare the performance of native code for packet classification, we generate native code for matching similar to the way used in Snort-1.8. This is a very simple way of matching where a packet is matched against all the rules for the protocol that it belongs to. For example, a tcp packet will be matched against all tcp rules. On the other hand, in Snort-2.6 it will be matched against only the tcp rules whose source and destination port values are compatible with the values in the packet.

We converted the signatures to a specification for our language as described in implementation section by combining rules which differed only in payload detection options. So we were left with around 300 unique rules on packet header fields for Snort-1.8 rule set and 600 rules for Snort-2.6 rule set. We ignore the payload options for evaluating the packet classification schemes.

We used two sets of packet traces for measuring runtime performance. The first one consists of all packets captured at the external firewall of a medium-size University laboratory that hosts about 30 hosts. Since the firewall is fully open to the Internet (i.e., the traffic is not pre-screened by another layer of firewalls in the University or elsewhere), the traffic is a reasonable representative of what one might expect a NIDS to be exposed to.

Our packet trace consisted of about 8 million packets collected over a few days. The second one corresponds to 10 days of packets from the MIT Lincoln Labs IDS evaluation data set [8], consisting of 17 million packets. We used Snort-2.6 in offline mode to perform these experiments. In offline mode, Snort reads the packets from a trace file using pcap_loop function call of pcap library. For each packet, pcap_loop calls a callback function.

In Snort, the callback function first decodes the packet and passes it to detection engine, i.e., the component that performs signature matching. We measured the time spent in the callback function by calling `times()` function before and after the call to pcap_loop.

To get the time spent in detection engine, we first ran Snort without making any call to the detection engine and measured the time spent in the callback function. This is the time spent in reading the packets from file and decoding them. Then we measured the time spent in the callback function for unmodified Snort. The difference in the two measured times gives the time spent in the detection engine.

We modified Snort-2.6 such that after decoding the packet was passed to the function in shared object which performs packet classification. We used the above technique to measure the time spent in this function. To compare the effect of the number of rules on the matching time, we used configuration files with different number of rules and found the packet classification time for the original Snort and the modified Snort that uses native code.

Figures 1 and 2 show the per packet classification time for Snort-1.8 rule set for the first and second packet traces. Even though Snort-2.6 groups the rules based on certain fields for each protocol, it is five times slower for the first packet trace and two times for the second trace when the complete rule sets are used. As expected, the matching time for compiled rules increases linearly with the number of rules but is always better than the interpreted rules by a factor of at least 2. Snort-2.6 stores the rules as some simple data structure in memory. At runtime, this data structure needs to be traversed. This traversal involves many memory accesses. The compiled code on the other hand performs this traversal using simple conditional and unconditional branching intructions. This is

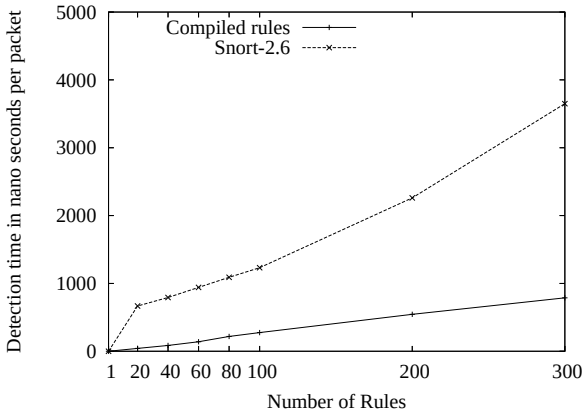

Figure 1: Matching time for Snort-1.8 rules for first trace.

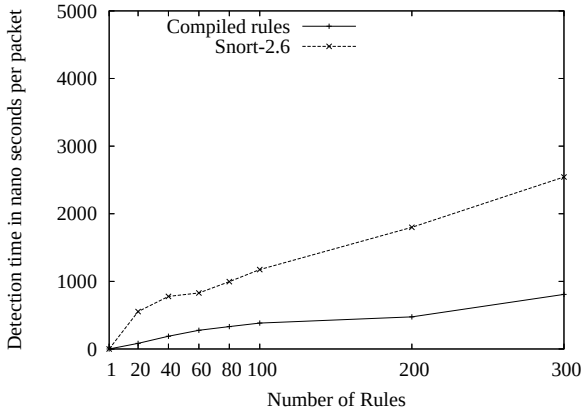

Figure 2: Matching time for Snort-1.8 rules for second trace.

the main reason for the performance improvement obtained by using native code over interpreted code.

The results for Snort-2.6 (Figures 3 and 4) are qualitatively similar showing that native compilation of packet classification, even with a naive matching scheme, performs about 30% better for the complete rule sets than the interpreted method that uses a more sophisticated scheme.

Related Work

Chandra, et al. developed Packet Types [2], a high-level specification language that provides a type system for packet formats similar to our language. It uses a construct called as *refinement* to capture the notion of inheritance. The inheritance mechanism of Packet Types offers more power than that of our language in that it can capture protocols that use trailers also. Our language trades off this power for simplicity. For the kind of protocols that Snort rules handle, this additional expressive power is not required.

Vern Paxson developed Bro [9] which is another popular open source NIDS. Bro has a powerful policy language that allows the use of sophisticated signatures. Bro comes with a translator to convert Snort

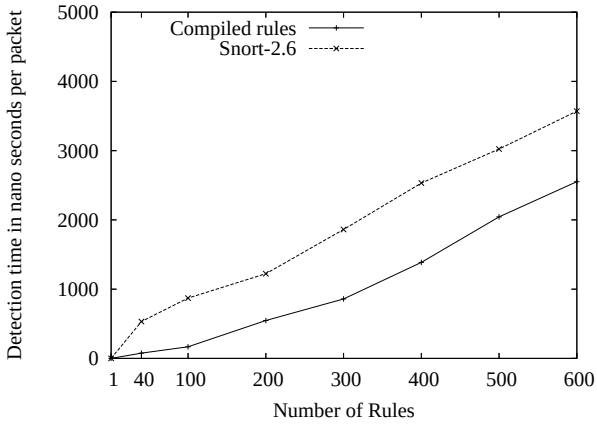

Figure 3: Matching time for Snort-2.6 rules for first trace.

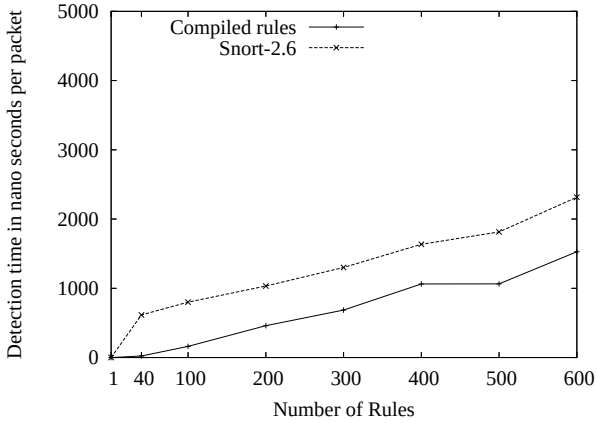

Figure 4: Matching time for Snort-2.6 Rules for second trace.

rules to Bro signatures [12]. But the semantics of matching differ in Bro and Snort. Unlike snort, which uses signatures that are based on individual packets, Bro performs matching on data streams obtained after packet reassembly. Our goal was to develop a plug-in for Snort that speeds up Snort while preserving the matching semantics.

Kruegel and Toth developed the Snort-NG [6] system, which demonstrated the performance gains achievable by parallelizing the signature matching. They use a two-stage approach where a decision tree is used for packet classification followed by content search. They use an interpreter based approach for packet classification. In that respect, our technique can help them in speeding-up the packet classification.

Previous research [14, 16] has looked at speeding up packet classification using hardware based approach. But these works focus on IP lookup problem, i.e., classifying packets based only on source and destination addresses and ports. Such an approach is useful in routers but not in intrusion detection systems like Snort which use additional fields like *ttl*, *window*, and *tcp flags*.

There has been a lot of research on speeding up the content matching component of Snort. Various software [3, 7] and hardware [15, 4] based solutions have been proposed in this area. In that respect our work can be used in conjunction with these solutions. Our packet classifier can quickly filter out the packets that do not need the expensive content matching operation.

Conclusion

In this paper, we presented a technique for improving the performance of packet classification used in IDS like Snort by using native code while preserving the matching semantics. Generating native code by hand is tedious and error prone. So we used a type system tailored for packet formats to generate type safe code. Our experiments with real and synthetic traces show that use of native code can result in speeding up the packet classification of Snort from 30% up to 80%. In the future, we want to generate native code for content matching.

Acknowledgments

We would like to thank Manuel Rivera and Lohit Vijayrenu who helped in writing the translator for Snort rules. We would also like to thank our shephard Brent Hoon Kang for his insightful comments and Rob Kolstad for his help in formatting the final version of the paper.

Bibliography

[1] Aho, A. and M. Corasick, ‘‘Efficient String Matching: An Aid to Bibliographic Search,’’ *Communications of the ACM*, Vol 18, Num. 6, pp. 333-343, 1975.

[2] Chandra, Satish and Peter J. McCann, ‘‘Packet Types,’’ *Second Workshop on Compiler Support for Systems Software (WCSSS)*, May, 1999.

[3] Fisk, M. and G. Varghese, *Fast Content Based Packet Handling for Intrusion Detection*, 2001.

[4] Jacob, Nigel and Carla Brodley, ‘‘Offloading IDS Computation to the GPU,’’ *ACSAC ’06: Proceedings of the 22nd Annual Computer Security Applications Conference*, IEEE Computer Society, pp. 371-380, 2006.

[5] Johansson, Erik and Christer Jonsson, *Native Code Compilation for erlang*, Technical Report, 1996.

[6] Kruegel, Christopher and Thomas Toth, ‘‘Using Decision Trees to Improve Signature-Based Intrusion Detection,’’ *6th Symposium on Recent Advances in Intrusion Detection (RAID)*, 2003.

[7] Kumar, Sailesh, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner, ‘‘Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection,’’ *SIGCOMM ’06: Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pp. 339-350, ACM, 2006.

[8] MIT Lincoln Labs, *DARPA Intrusion Detection Evaluation*, 1999.

[9] Paxson, V., ‘‘Bro: A System for Detecting Network Intruders in Real-Time,’’ *USENIX Security*, 1998.

[10] Roesch, Martin, ‘‘Snort – Lightweight Intrusion Detection for Networks,’’ *13th Systems Administration Conference*, USENIX, 1999.

[11] Sekar, R., Y. Guang, S. Verma, and T. Shanbhag, ‘‘A High-Performance Network Intrusion Detection System,’’ *ACM Conference on Computer and Communications Security*, pp. 8-17, 1999.

[12] Sommer, R. and V. Paxson, ‘‘Enhancing Byte-Level Network Intrusion Detection Signatures with Context,’’ *ACM CCS*, 2003.

[13] Wu, S. and U. Manber, ‘‘A Fast Algorithm for Multi-Pattern Searching,’’ *Technical Report TR-94-17*, 1994.

[14] Yu, Fang and Randy Katz, ‘‘Efficient Multi-Match Packet Classification with TCAM,’’ *High Performance Interconnects*, 2004.

[15] Yu, Fang, Randy H. Katz, and T. V. Lakshman, ‘‘Gigabit Rate Packet Pattern-Matching Using TCAM,’’ *12th IEEE International Conference on Network Protocols*, 2004.

[16] Yu, Fang, T. V. Lakshman, Marti Austin Motoyama, and Randy H. Katz, ‘‘SSA: A Power and Memory Efficient Scheme to Multi-Match Packet Classification,’’ *Symposium on Architectures for Networking and Communications Systems*, 2005.