SNORTRAN: An Optimizing Compiler for Snort Rules

Sergei Egorov
esl@fidelissec.com

Gene Savchuk
savchuk@fidelissec.com

Fidelis Security Systems, Inc.
www.fidelissec.com

Abstract

We developed an optimizing compiler for intrusion detection rules popularized by an open-source Snort Network Intrusion Detection System (www.snort.org). While Snort and Snort-like rules are usually thought of as a list of independent patterns to be tested in a sequential order, we demonstrate that common compilation techniques are directly applicable to Snort rule sets and are able to produce high-performance matching engines. SNORTRAN combines several compilation techniques, including cost-optimized decision trees, pattern matching precompilation, and string set clustering. Although all these techniques have been used before in other domain-specific languages, we believe their synthesis in SNORTRAN is original and unique.

Introduction

Snort [RO99] is a popular open-source Network Intrusion Detection System (NIDS). Snort is controlled by a set of pattern/action rules residing in a configuration file of a specific format. Due to Snort's popularity, Snort-like rules are accepted by several other NIDS [FSTM, HANK].

In this paper we describe an optimizing compiler for Snort rule sets called SNORTRAN that incorporates ideas of pattern matching compilation based on cost-optimized decision trees [DKP92, KS88] with setwise string search algorithms popularized by recent research in high-performance NIDS detection engines [FV01, CC01, GJMP]. The two main design goals were performance and compatibility with the original Snort rule interpreter.

The primary application area for NIDS is monitoring IP traffic inside and outside of firewalls, looking for unusual activities that can be attributed to external attacks or internal misuse. Most NIDS are designed to handle T1/partial T3 traffic, but as the number of the known vulnerabilities grows and more and more weight is given to internal misuse monitoring on high-throughput networks (100Mbps/1Gbps), it gets harder to keep up with the traffic without dropping too many packets to make detection ineffective. Throwing hardware at the problem is not always possible because of growing maintenance and support costs, let alone the fact that the problem of making multi-unit system work in realistic environment is as hard as the original performance problem.

Bottlenecks of the detection process were identified by many researchers and practitioners [FV01, ND02, GJMP], and several approaches were proposed [FV01, CC01]. Our benchmarking supported the performance analysis made by M. Fisk and G. Varghese [FV01], adding some interesting findings on worst-case behavior of setwise string search algorithms in practice.

Traditionally, NIDS are designed around a packet grabber (system-specific or **libcap**) getting traffic packets off the wire, combined with preprocessors, packet decoders, and a detection engine looking for a static set of signatures loaded from a rule file at system startup. Snort [SNORT] and

Firestorm [FSTM] use similar techniques to interpret those rules at run time: rules are preprocessed at startup time and their internal representation in a form of multidimensional tree-like data structure is used at run time. Snort goes a long way to make detection based on this structure effective; common attributes are identified and used to funnel the matching process into the respective subtrees, different kinds of attribute predicates are implemented as pointers to functions chosen at startup time etc.

While these techniques are valuable and yield good results (Snort's matching engine is one of the fastest in the NIDS space), they fall short of what is available in other domains. Ten years of research in Domain-Specific Languages provided us with the tools and methods to replace Snort's *ad-hoc* precompilation with a full-scale compiler capable of comprehensive analysis of a complete rule set, cost-based optimization of attribute evaluation, and balancing of string sets for effective string matching at run time.

Preliminary Benchmarks

To raise interest in our approach, we present the benchmarks of a SNORTRAN-generated detection engine, and compare them with original Snort engine. Benchmarks are in no way a reflection of real performance in all situations, but they are easy to reproduce and can motivate further research in this area.

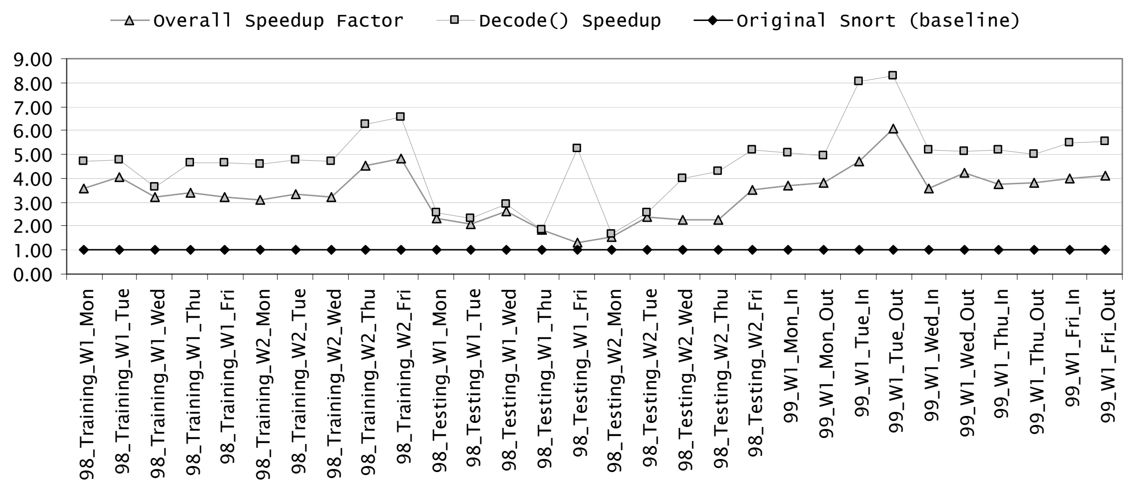

Fig.1. Relative performance of SNORTRAN-based Snort system

Our benchmarks measured two things: overall Snort performance (including all standard preprocessors, complete set of rules, and the fast output plugin), and engine performance (time spent in the function `Detect`). We used captured **tcpdump** traffic provided by MIT's Lincoln Lab as a part of DARPA-sponsored IDS evaluations performed in 1998 and 1999 [LL99]. This traffic has been generated to imitate statistical characteristics of real traffic observed at the Air Force computer center in order to test various characteristics of IDS systems. The set of rules used was taken from the default Snort 1.8.7 distribution; we turned on all the rules in the distribution, including rules turned off by default. The total amount of rules in our tests was 1603.

The chart on Fig.1 demonstrates that in our tests SNORTRAN-generated detection engine has an average speedup factor of 4.5. This speedup is reflected in overall Snort performance, but overall benefits are about 3.5 on average.

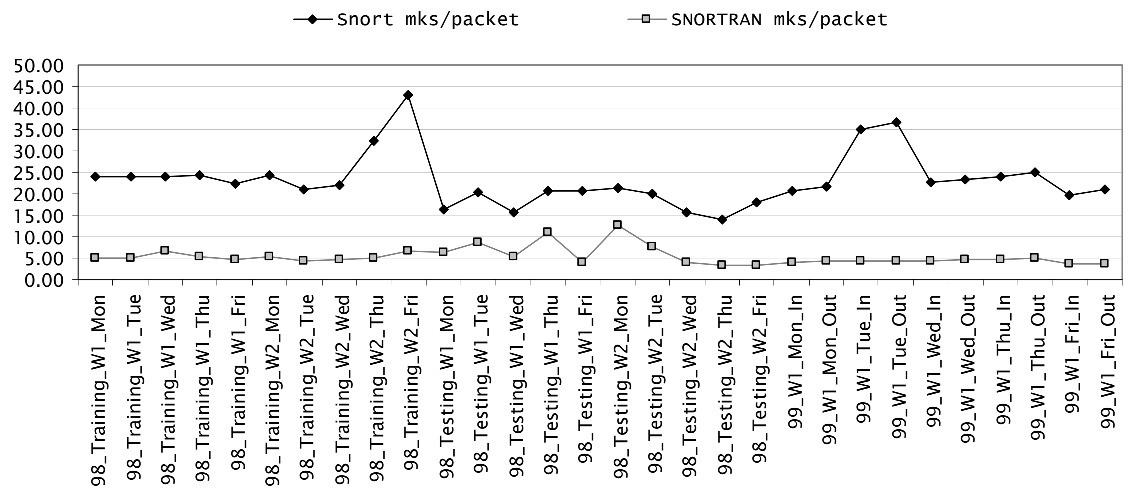

Fig.2. Time-per-packet in microseconds for original and SNORTRAN-generated detection engines

Time-per-packet measurements shown on Fig. 2 reflect the speed of the detection function alone. To put this in context, our experiments show that 1Gbit traffic amounts to one packet sent about every 8 microseconds (your mileage may vary). This means that even on fast stock hardware (in these benchmarks we used 2Ghz Pentium IV-based system with 1GB RAM), SNORTRAN-based system in its current form will lose some packets (regular Snort will lose a lot more). Tuning the system to get top performance from every component can help to get under the desired 8 microseconds per packet.

It is hard to compare our results to results of research systems described in [FV01] and [CC01] because of their authors' focus on benchmarking setwise algorithms, not full detection engines, and use of specially selected rule sets. The total Snort run time improvements measured by the Silicon Defense team using a benchmarking method similar to ours were modest: 1.02 to 1.18 times Snort's performance on 800+ rules / Defcon 8 traffic (improvements in content matching itself were from 1.3 to 3.32) [CC01]. Experimental results of M. Fisk and G. Varghese [FV01] are more diverse: they range from negative improvement (per-protocol string sets) to 1.52 times on balanced mix of setwise algorithms; local traffic was matched against 800+ rules of Snort 1.8.6 distribution. Pure content matching improvements were about 4.6 (HTTP rules vs. HTTP traffic).

The above SNORTRAN benchmarks are preliminary—many optimizations are yet to be implemented and the values of various static and dynamic parameters most probably represent just a local minimum in the parameter space. Nevertheless, as demonstrated by these benchmarks, SNORTRAN produces matching engines that are better than and in many cases significantly better than engines currently used in Snort and Firestorm. Most importantly, it demonstrates that Snort-like rules can be compiled into efficient matching engines, competitive with ones encoded in "procedural" domain-specific languages like NFR's N-code [NFR97].

Not just string search

The importance of fast string search is a common theme in most articles dedicated to NIDS performance[1]. Profiling of Snort done by several authors attributes 30% time on average to the string search, making it the single most expensive procedure in the whole Snort program. While this is generally true, we have to consider other factors at play before deciding the best way to proceed:

- Percentage of time spent in Snort's implementation of Boyer-Moore string search varies with traffic significantly: the largest number we have seen was 80% (heavy http traffic with all rules turned on), while the smallest number was less then 1% (in 98_Testing_W1_Fri, 80% of the load is shifted to **stream4** preprocessor).
- A network under attack behaves quite differently from a "normal" network, so one has to choose what to optimize for: there is a trade-off between sensitivity on normal traffic (no dropped packets) and quality/volume of alerts under attack.
- Rule sets are not created equal. Small changes to rules can lead to disproportional performance gains or losses.

Our profiling of Snort on various data sets (Lincoln Lab [LL98] and Defcon [CCTF] traffic) lead us to believe that Snort has four major performance bottlenecks:

- **Stream4** preprocessor (heavy load in some situations)
- Header matching (heavy load on small and non-TCP packet attacks)
- Content matching (heavy load in HTTP- reach traffic)
- Output plug-ins (heavy load on any large-volume attack)

SNORTRAN focuses on bottlenecks 2 and 3; together they account for 80% of the total execution time on "normal" traffic when matched against the full Snort 1.8.7 rule set (~1600 rules). Output plug-in overload is a Snort-specific problem that is likely to go away with version 1.9. Stream4, stream reassembly and stateful analysis in general are topics of our future research.

Compiler Structure

There are five phases in SNORTRAN compiler:

1. Parsing
2. Attribute normalization
3. Creation of the optimal decision tree
4. String set clustering
5. Code generation.

[1] In this article we differentiate between *performance* as raw speed, and *effectiveness* as percentage of detected attacks; in traditional IDS model, effectiveness is simply a derivative of performance (fewer dropped packets) and rule set quality (more high-quality rules).

We assume that the reader is familiar with basic compiler construction methods, so we will concentrate on domain-specific phases (2-4).

Attribute Normalization

Snort-style intrusion detection rules consist of a *header* containing common IP/TCP/UDP checks (protocol, ports, IP addresses), followed by a list of *options* specifying additional tests such as content string. SNORTRAN converts rules to an internal representation suitable for correlation analysis. In this representation, each rule is transformed into an *attribute vector* (two vectors for bidirectional rules).

All attribute vectors have the same length, equal to the total number of supported tests. In most cases there is one-to-one correspondence between attributes and individual options or header tests, for example, source port is a single attribute made from a part of the header, packet size is another single attribute made from an option, and content string is a single attribute made from several options specifying various parameters of string search.

Attribute values are normalized to exhibit a uniform set-like behavior; they can include one another, be disjoint, intersect etc. After converting all rules to attribute vectors, SNORTRAN builds a *lattice* of relations between attribute values for each supported attribute (vector index), reflecting the internal structure of the value space. The lattices range from very simple (all values are the same) to very complex (numerical ranges including smaller numerical ranges up to individual numbers). The shape of the lattice serves as a measure of variability of the corresponding attribute and is used to rank attributes according to their importance in the detection process.

Attribute ranking is based on estimates of *entropy* (an *uncertainty function*) as a measure of information gained in performing a particular test or series of tests. Entropy takes into account probabilities of possible outcomes and gives a measure of importance of a test in comparison with other candidate tests (see [ASH65] for details on entropy in information theory).

Creation of the Optimal Decision Tree

When attribute vectors are built and attributes are ranked, SNORTRAN creates a decision tree by iterating through ranks and attribute vectors. For each rank, the compiler chooses a set of candidate tests and orders them based on change in entropy, which is basically an amount of "uncertainty" dispelled by each test (intuitively, the best test is the one dividing the search space in half). Entropy dispelled by a test is weighted before comparison with other tests; weighting allows the compiler to take into account other factors such as mutual correlation between tests and cost of performing the test.

Correlation is measured with the help of the relation lattices calculated on the previous phase. Cost is measured by making an estimate of processor time needed to perform the test (we use standard Pentium instruction timing information, making educated guesses in complex cases).

At any stage of decision tree building, the *dominant* test is the one that has the largest value of weighted entropy delta. When the dominant test is chosen, the compiler adds the corresponding node to the decision tree, and descends recursively to the subtrees corresponding to possible outcomes of the test. Each outcome is processed in a context remembering the outcomes of the parent tests, thus eliminating redundant tests down the road and keeping the tree size small. In a new context, new values of entropy deltas are calculated, and so forth.

The decision tree built on this stage contains several types of nodes corresponding to the supported kinds of run-time tests; current version of SNORTRAN supports simple conditional branches, multi-way branches (table jumps), and calls to dispatch functions. This tree serves as a blueprint for subsequent stages of the compilation.

String Set Clustering

Due to the importance of effective content matching, SNORTRAN pays special attention to content matching attributes. Its task is to combine content matches that can be performed simultaneously into optimal groups. Content groups are formed by bringing together patterns that are likely to be used together at run time and may benefit from parallel matching. The compiler considers the following factors:

- Setwise string matching algorithms (variations of Aho-Corasick, Commentz-Walter and generalizations of Boyer-Moore algorithms) have different performance characteristics, partially dependent on characteristics of the string set to be matched, such as the length of the shortest string.
- Large groups tend to cause many unnecessary tests (only a subset of test results is needed by subsequent tests). Unnecessary tests affect setwise algorithms directly (by increasing the number of run-time comparisons) and indirectly (by making larger skips impossible). On the other hand, for well-balanced groups extra tests may incur little or no overhead.
- Small groups with significant variability are ineffective due to common factors affecting the performance of setwise algorithms (the details are given below). Extreme examples are singleton groups that are better served by the Boyer-Moore-Horspool algorithm, and groups of two or three strings with one short string causing noticeable drop in setwise performance.

The information gathered on previous stages is used to provide estimates for probability of co-occurrence of two strings in same-packet run-time tests. SNORTRAN calculates a hash of decision tree path leading to a particular node and uses this hash to define the similarity on pattern strings. Other factors affecting the similarity function are common prefixes / suffixes, similar length, and basic parameters of pattern search (case sensitivity, offset and depth options). Given this similarity function and the total set of strings, the compiler utilizes a simple clustering algorithm to calculate optimal string clusters.

Dynamic Worst-case Avoidance

Authors' benchmarking of various setwise matching algorithms provided useful insights into differences between the original problem solved by those algorithms and realities of NIDS packet matching.

Setwise algorithms were designed to find positions of *all* matches of each string in the string set by scanning the full length of the target string. This formulation of the problem is similar but not equivalent to what Snort needs: it looks for the information on which strings of the set are *present* in the target string. Answers provided by setwise algorithms contain information Snort doesn't use and this information doesn't come for free.

A simple example of worst-case behavior is demonstrated by Lincoln Lab's traffic file with oversized ICMP packet attack. The attack contains a large volume of 1K packets with all-zeroes content. The pattern set matched against this content contains a 20-byte-long all-zeroes pattern from "`ICMP Nemesis v1.1 Echo`" rule (one of nine patterns). Setwise algorithms will look for *every* place where this all-zeroes pattern matches the all-zeroes content because all patterns are matched together; this means that even when the first match of the zeroes pattern is discovered at offset 0, it still will affect the matching of all other patterns until the end of the packet. Although some setwise algorithms take internal repeats into account and may not fall back to 1-byte skip, practical setwise performance is noticeably worse than that of a naïve Boyer-Moore-Horspool loop that looks for a single occurrence of each pattern.

Setwise algorithms do exactly what Snort needs in a single, but very important case: when no matches are found in the packet. When one or more matches are found, benefits of proceeding with setwise match diminish up to the point when setwise algorithms run several times slower than simple Boyer-Moore-Horspool loop. SNORTRAN-generated engines use this heuristic as well as comparisons of relative performance of Aho-Corasick, Commentz-Walter and generalizations of Boyer-Moore algorithms on different kinds of string sets and target strings (short target strings have their own specifics) to switch between algorithms when the size of the target string becomes known and on-the-fly when matches show up.

Some of the implementations we used in our benchmarks were written by the authors (two variants of setwise Boyer-Moore-Horspool), others were adapted from various sources, including **strmat** package [STMT] (Aho-Corasick, setwise Boyer-Moore) and GNU grep [GREP] (Commentz-Walter). Although dynamic switching between algorithms proved beneficial in our testing, is not yet benchmarked systematically and requires further research.

Compiler Output

SNORTRAN utilizes the freely available GCC compiler on the backend. Driving GCC allows for many GCC-specific optimizations such as global register allocation. The resulting code is linked with string search algorithms and utility procedures to produce a library used by modified version of Snort. Snort modifications are localized in the `rules.c` file and are quite simple: Snort's implementation of `Detect()` is replaced by a call to our engine with the same interface.

In addition to the detection engine in a library form, the compiler produces a filtered version of the configuration/rules file with all rules removed. This file contains all the information required for the Snort host: plug-in specifications, alert classification table etc. To support effective spooling of alerts and logs, SNORTRAN generates rule map file in Barnyard-compatible format. Information in this file complements the "unified" Snort output.

Snort Compatibility

A detection engine produced by the compiler is a functional equivalent of Snort's own Detect function and in most cases generates exactly the same events given the same input traffic. Some incompatibilities still exist, though; overlapping patterns can lead to differences in events due to differences in internal order of matches and Snort's "first match wins" policy. As Snort shifts towards "report all matches" model, this difference will disappear.

Another source of incompatibility is SNORTRAN's missing support for advanced Snort features like per-rule alert configuration ("`ruletype`" declarations), modification of the default order of event processing ("`config order`"), and activate/dynamic rules. The importance of these omissions is subject for discussion; we will just note that none of these features are used neither by the default Snort configuration nor by the rule sets supplied by the Whitehats community database [WHTS].

Future Directions

We plan to continue benchmarking and tuning the compiler and the runtime library, add more optimizations, improve Snort compatibility. Improving compiler's performance and lowering its memory requirements is also high on our To Do list; getting inside practical limits in this regard will allow the compiler to reside on the Snort box and be used interactively.

Since SNORTRAN is fully independent of Snort itself, it is easy to adapt it to other similar IDS systems; our next target is Firestorm [FSTM]. We will investigate the preprocessor bottleneck and possibilities for better load balancing on multiprocessor hardware.

References

[LL99] MIT Lincoln Laboratory, 1998/1999 DARPA Off-Line Intrusion Detection Evaluation**,** http://www.ll.mit.edu/SST/ideval/

[CCTF] The Shmoo Group, Capture the Capture The Flag, http://www.shmoo.com/cctf/

[ASH65] R. B. Ash, Information Theory, Dover Publications, NY, 1965

[DKP92] S. Debray, S. Kannan, M. Paithane, Weighted Decision Trees, Proceedings of the Joint International Conference and Symposium on Logic Programming, Washington, USA, 1992.

[KS88] S. Kliger and E. Shapiro, A Decision Tree Compilation Algorithm for FCP, Proc. Fifth Int. Conf. on Logic Programming, Seattle, Aug. 1988, pp. 1315--1336. MIT Press.

[FV01] M. Fisk and G. Varghese. Fast content-based packet handling for intrusion detection. Technical Report CS2001-0670, University of California, San Diego, Department of Computer Science and Engineering, June 2001.

[CC01] J. McAlerney, C. Coit, S. Staniford. Towards faster pattern matching for intrusion detection. DARPA Information Survivability Conference and Exposition, 2001.

[RO99] Martin Roesh. Snort: Lightweight intrusion detection for networks, in Proceedings of the 13th Systems Administration Conference. 1999, USENIX.

[GJMP] S. Gossin, N. Jones, N. McCurdy, R. Persaud, Pattern Matching in Snort

[ND02] N. Desai, Increasing Performance in High Speed NIDS, 2002

[NFR97] M. Ranum, K. Landfield, M. Stolarchuk, M. Sienkiewicz, A. Lambeth and E. Wall, Implementing a generalized tool for network monitoring, Proc. LISA '97, USENIX 11th Systems Administration Conference, San Diego, 1997.

[SNORT] Snort.org, http://www.snort.org

[FSTM] Firestorm NIDS, http://www.scaramanga.co.uk/firestorm/

[HANK] Hank NIDS, http://hank.sourceforge.net/

[WHTS] Whitehats, arachNIDS, http://www.whitehats.com/ids/

[STMT] Dan Gusfield, Strmat package, http://www.cs.ucdavis.edu/~gusfield/strmat.html

[GREP] GNU Grep, Free Software Foundation, http://www.gnu.org/software/grep/grep.html