Enhancing Byte-Level Network Intrusion Detection Signatures with Context

Robin Sommer
TU München
Germany
sommer@in.tum.de

Vern Paxson
International Computer Science Institute and
Lawrence Berkeley National Laboratory
Berkeley, CA, USA
vern@icir.org

ABSTRACT

Many network intrusion detection systems (NIDS) use byte sequences as signatures to detect malicious activity. While being highly efficient, they tend to suffer from a high false-positive rate. We develop the concept of *contextual signatures* as an improvement of string-based signature-matching. Rather than matching fixed strings in isolation, we augment the matching process with additional context. When designing an efficient signature engine for the NIDS Bro, we provide low-level context by using regular expressions for matching, and high-level context by taking advantage of the semantic information made available by Bro's protocol analysis and scripting language. Therewith, we greatly enhance the signature's expressiveness and hence the ability to reduce false positives. We present several examples such as matching requests with replies, using knowledge of the environment, defining dependencies between signatures to model step-wise attacks, and recognizing exploit scans.

To leverage existing efforts, we convert the comprehensive signature set of the popular freeware NIDS Snort into Bro's language. While this does not provide us with improved signatures by itself, we reap an established base to build upon. Consequently, we evaluate our work by comparing to Snort, discussing in the process several general problems of comparing different NIDSs.

Categories and Subject Descriptors: C.2.0 [Computer-Communication Networks]: General - *Security and protection*.

General Terms: Performance, Security.

Keywords: Bro, Network Intrusion Detection, Pattern Matching, Security, Signatures, Snort, Evaluation

1. INTRODUCTION

Several different approaches are employed in attempting to detect computer attacks. *Anomaly-based* systems derive (usually in an automated fashion) a notion of "normal" system behavior, and report divergences from this profile, an approach premised on the notion that attacks tend to look different in some fashion from legitimate computer use. *Misuse detection* systems look for particular, explicit indications of attacks (*Host-based* IDSs inspect audit logs for this while *network-based* IDSs, or NIDSs, inspect the network traffic).

CCS'03, October 27–30, 2003, Washington, DC, USA.

In this paper, we concentrate on one popular form of misuse detection, network-based *signature matching* in which the system inspects network traffic for matches against exact, precisely-described patterns. While NIDSs use different abstractions for defining such patterns, most of the time the term *signature* refers to raw byte sequences. Typically, a site deploys a NIDS where it can see network traffic between the trusted hosts it protects and the untrusted exterior world, and the signature-matching NIDS inspects the passing packets for these sequences. It generates an alert as soon as it encounters one. Most commercial NIDSs follow this approach [19], and also the most well-known freeware NIDS, Snort [29]. As an example, to detect the buffer overflow described in `CAN-2002-0392` [9], Snort's signature #1808 looks for the byte pattern `0xC0505289-E150515250B83B000000CD80` [2] in Web requests. Keeping in mind that there are more general forms of signatures used in intrusion detection as well—some of which we briefly discuss in §2—in this paper we adopt this common use of the term *signature*.

Signature-matching in this sense has several appealing properties. First, the underlying conceptual notion is simple: it is easy to explain what the matcher is looking for and why, and what sort of total coverage it provides. Second, because of this simplicity, signatures can be easy to share, and to accumulate into large "attack libraries." Third, for some signatures, the matching can be quite *tight*: a match indicates with high confidence that an attack occurred.

On the other hand, signature-matching also has significant limitations. In general, especially when using tight signatures, the matcher has no capability to detect attacks other than those for which it has explicit signatures; the matcher will in general completely miss novel attacks, which, unfortunately, continue to be developed at a brisk pace. In addition, often signatures are not in fact "tight." For example, the Snort signature #1042 to detect an exploit of `CVE-2000-0778` [9] searches for "`Translate: F`" in Web requests; but it turns out that this header is regularly used by certain applications. Loose signatures immediately raise the major problem of *false positives*: alerts that in fact do not reflect an actual attack. A second form of false positive, which signature matchers likewise often fail to address, is that of *failed attacks*. Since at many sites attacks occur at nearly-continuous rates, failed attacks are often of little interest. At a minimum, it is important to distinguish between them and successful attacks.

A key point here is that the problem of false positives can potentially be greatly reduced if the matcher has additional *context* at its disposal: either additional particulars regarding the exact activity and its semantics, in order to weed out false positives due to overly general "loose" signatures; or the additional information of how the attacked system responded to the attack, which often indicates whether the attack succeeded.

In this paper, we develop the concept of *contextual signatures*, in which the traditional form of string-based signature matching is augmented by incorporating additional context on different levels when evaluating the signatures. First of all, we design and implement an efficient pattern matcher similar in spirit to traditional *signature engines* used in other NIDS. But already on this low-level we enable the use of additional context by *(i)* providing full regular expressions instead of fixed strings, and *(ii)* giving the signature engine a notion of full connection state, which allows it to correlate multiple interdependent matches in both directions of a user session. Then, if the signature engine reports the match of a signature, we use this event as the *start* of a decision process, instead of an alert by itself as is done by most signature-matching NIDSs. Again, we use additional context to judge whether something alert-worthy has indeed occurred. This time the context is located on a higher-level, containing our knowledge about the network that we have either explicitly defined or already learned during operation.

In §3.5, we will show several examples to demonstrate how the concept of contextual signatures can help to eliminate most of the limitations of traditional signatures discussed above. We will see that regular expressions, interdependent signatures, and knowledge about the particular environment have significant potential to reduce the false positive rate and to identify failed attack attempts. For example, we can consider the server's response to an attack and the set of software it is actually running—its *vulnerability profile*—to decide whether an attack has succeeded. In addition, treating signature matches as events rather than alerts enables us to analyze them on a meta-level as well, which we demonstrate by identifying *exploit scans* (scanning multiple hosts for a known vulnerability).

Instrumenting signatures to consider additional context has to be performed manually. For each signature, we need to determine what context might actually help to increase its performance. While this is tedious for large sets of already-existing signatures, it is not an extra problem when developing new ones, as such signatures have to be similarly adjusted to the specifics of particular attacks anyway. Contextual signatures serve as a building block for increasing the expressivess of signatures; not as a stand-alone solution.

We implemented the concept of contextual signatures in the framework already provided by the freeware NIDS Bro [25]. In contrast to most NIDSs, Bro is fundamentally neither an anomaly-based system nor a signature-based system. It is instead partitioned into a *protocol analysis component* and a *policy script component*. The former feeds the latter via generating a stream of *events* that reflect different types of activity detected by the protocol analysis; consequently, the analyzer is also referred to as the *event engine*. For example, when the analyzer sees the establishment of a TCP connection, it generates a `connection_established` event; when it sees an HTTP request it generates `http_request` and for the corresponding reply `http_reply`; and when the event engine's heuristics determine that a user has successfully authenticated during a Telnet or Rlogin session, it generates `login_success` (likewise, each failed attempt results in a `login_failure` event).

Bro's event engine is *policy-neutral*: it does not consider any particular events as reflecting trouble. It simply makes the events available to the policy script interpreter. The interpreter then executes scripts written in Bro's custom scripting language in order to define the response to the stream of events. Because the language includes rich data types, persistent state, and access to timers and external programs, the response can incorporate a great deal of context in addition to the event itself. The script's reaction to a particular event can range from updating arbitrary state (for example, tracking types of activity by address or address pair, or grouping related connections into higher-level "sessions") to generating alerts (e.g., via *syslog*) or invoking programs for a reactive response.

More generally, a Bro policy script can implement signature-style matching—for example, inspecting the URIs in Web requests, the MIME-encoded contents of email (which the event engine will first unpack), the user names and keystrokes in login sessions, or the filenames in FTP sessions—but at a higher semantic level than as just individual packets or generic TCP byte streams.

Bro's layered approach is very powerful as it allows a wide range of different applications. But it has a significant shortcoming: while, as discussed above, the policy script is capable of performing traditional signature-matching, doing so can be cumbersome for large sets of signatures, because each signature has to be coded as part of a script function. This is in contrast to the concise, low-level languages used by most traditional signature-based systems. In addition, if the signatures are matched sequentially, then the overhead of the matching can become prohibitive. Finally, a great deal of community effort is already expended on developing and disseminating packet-based and byte-stream-based signatures. For example, the 1.9.0 release of Snort comes with a library of 1,715 signatures [2]. It would be a major advantage if we could leverage these efforts by incorporating such libraries.

Therefore, one motivation for this work is to combine Bro's flexibility with the capabilities of other NIDSs by implementing a signature engine. But in contrast to traditional systems, which use their signature matcher more or less on its own, we tightly integrate it into Bro's architecture in order to provide contextual signatures. As discussed above, there are two main levels on which we use additional context for signature matching. First, at a detailed level, we extend the expressiveness of signatures. Although byte-level pattern matching is a central part of NIDSs, most only allow signatures to be expressed in terms of fixed strings. Bro, on the other hand, already provides regular expressions for use in policy scripts, and we use them for signatures as well. The expressiveness of such patterns provides us with an immediate way to express *syntactic context*. For example, with regular expressions it is easy to express the notion "string `XYZ` but only if preceded at some point earlier by string `ABC`". An important point to keep in mind regarding regular expression matching is that, once we have fully constructed the matcher, which is expressed as a Deterministic Finite Automaton (DFA), the matching can be done in $O(n)$ time for n characters in the input, and also $\Omega(n)$ time. (That is, the matching always takes time linear in the size of the input, regardless of the specifics of the input.) The "parallel Boyer-Moore" approaches that have been explored in the literature for fast matching of multiple fixed strings for Snort [12, 8] have a wide range of running times—potentially sublinear in n, but also potentially superlinear in n. So, depending on the particulars of the strings we want to match and the input against which we do the matching, regular expressions might prove fundamentally more efficient, or might not; we need empirical evaluations to determine the relative performance in practice. In addition, the construction of a regular expression matcher requires time potentially exponential in the length of the expression, clearly prohibitive, a point to which we return in §3.1.

Second, on a higher level, we use Bro's rich contextual state to implement our improvements to plain matching described above. Making use of Bro's architecture, our engine sends events to the policy layer. There, the policy script can use all of Bro's already existing mechanisms to decide how to react. We show several such examples in §3.5.

Due to Snort's large user base, it enjoys a comprehensive and up-to-date set of signatures. Therefore, although for flexibility we have designed a custom signature language for Bro, we make use of the Snort libraries via a conversion program. This program takes an unmodified Snort configuration and creates a corresponding Bro signature set. Of course, by just using the same signatures in Bro as in Snort, we are not able to improve the resulting alerts in terms of quality. But even if we do not accompany them with additional context, they immediately give us a baseline of already widely-deployed signatures. Consequently, Snort serves us as a reference. Throughout the paper we compare with Snort both in terms of quality and performance. But while doing so, we encountered several general problems for evaluating and comparing NIDSs. We believe these arise independently of our work with Bro and Snort, and therefore describe them in some detail. Keeping these limitations in mind, we then evaluate the performance of our signature engine and find that it performs well.

§2 briefly summarizes related work. In §3 we present the main design ideas behind implementing contextual signatures: regular expressions, integration into Bro's architecture, some difficulties with using Snort signatures, and examples of the power of the Bro signature language. In §4 we discuss general problems of evaluating NIDSs, and then compare Bro's signature matching with Snort's. §5 summarizes our conclusions.

2. RELATED WORK

[4] gives an introduction to intrusion detection in general, defining basic concepts and terminology.

In the context of signature-based network intrusion detection, previous work has focussed on efficiently matching hundreds of fixed strings in parallel: [12] and [8] both present implementations of set-wise pattern matching for Snort [29]. For Bro's signature engine, we make use of regular expressions [18]. They give us both flexibility and efficiency. [17] presents a method to incrementally build the underlying DFA, which we can use to avoid the potentially enormous memory and computation required to generate the complete DFA for thousands of signatures. An extended form of regular expressions has been used in intrusion detection for defining sequences of events [30], but to our knowledge no NIDS uses them for actually matching multiple byte patterns against the payload of packets.

In this paper, we concentrate on signature-based NIDS. Snort is one of the most-widely deployed systems and relies heavily on its signature set. Also, most of the commercial NIDSs are signature-based [19], although there are systems that use more powerful concepts to express signatures than just specifying byte patterns. NFR [28], for example, uses a flexible language called `N-Code` to declare its signatures. In this sense, Bro already provides sophisticated signatures by means of its policy language. But the goal of our work is to combine the advantages of a traditional dedicated pattern matcher with the power of an additional layer abstracting from the raw network traffic. IDS like STAT [35] or Emerald [26] are more general in scope than purely network-based systems. They contain misuse-detection components as well, but their signatures are defined at a higher level. The STAT framework abstracts from low-level details by using transitions on a set of states as signatures. A component called NetSTAT [36] defines such state transitions based on observed network-traffic. Emerald, on the other hand, utilizes P-BEST [20], a production-based expert system to define attacks based on a set of facts and rules. Due to their general scope, both systems use a great deal of context to detect intrusions. On the other hand, our aim is to complement the most common form of signature matching—low-level string matching—with context, while still keeping its efficiency.

The huge number of generated alerts is one of the most important problems of NIDS (see, for example, [23]). [3] discusses some statistical limits, arguing in particular that the false-alarm rate is the limiting factor for the performance of an IDS.

Most string-based NIDSs use their own signature language, and are therefore incompatible. But since most languages cover a common subset, it is generally possible to convert the signatures of one system into the syntax of another. ArachNIDS [1], for example, generates signatures dynamically for different systems based on a common database, and [32] presents a conversion of Snort signatures into STAT's language, although it does not compare the two systems in terms of performance. We take a similar approach, and convert Snort's set into Bro's new signature language.

For evaluation of the new signature engine, we take Snort as a reference. But while comparing Bro and Snort, we have encountered several difficulties which we discuss in §4. They are part of the general question of how to evaluate NIDSs. One of the most comprehensive evaluations is presented in [21, 22], while [24] offers a critique of the methodology used in these studies. [14] further extends the evaluation method by providing a user-friendly environment on the one hand, and new characterizations of attack traffic on the other hand. More recently, [10] evaluates several commercial systems, emphasizing the view of an analyst who receives the alerts, finding that these systems ignore relevant information about the context of the alerts. [15] discusses developing a benchmark for NIDSs, measuring their capacity with a representative traffic mix. (Note, in §4.2 we discuss our experiences with the difficulty of finding "representative" traces.)

3. CONTEXTUAL SIGNATURES

The heart of Bro's contextual signatures is a signature engine designed with three main goals in mind: *(i)* expressive power, *(ii)* the ability to improve alert quality by utilizing Bro's contextual state, and *(iii)* enabling the reuse of existing signature sets. We discuss each in turn. Afterwards, we present our experiences with Snort's signature set, and finally show examples which demonstrate applications for the described concepts.

3.1 Regular Expressions

A traditional signature usually contains a sequence of bytes that are representative of a specific attack. If this sequence is found in the payload of a packet, this is an indicator of a possible attack. Therefore, the matcher is a central part of any signature-based NIDS. While many NIDSs only allow fixed strings as search patterns, we argue for the utility of using *regular expressions*. Regular expressions provide several significant advantages: first, they are far more flexible than fixed strings. Their expressiveness has made them a well-known tool in many applications, and their power arises in part from providing additional *syntactic context* with which to sharpen textual searches. In particular, character classes, union, optional elements, and closures prove very useful for specifying attack signatures, as we see in §3.5.1.

Surprisingly, given their power, regular expressions can be matched very efficiently. This is done by compiling the expressions into DFAs whose terminating states indicate whether a match is found. A sequence of n bytes can therefore be matched with $O(n)$ operations, and each operation is simply an array lookup—highly efficient.

The total number of patterns contained in the signature set of a NIDSs can be quite large. Snort's set, for example, contains 1,715 distinct signatures, of which 1,273 are enabled by default. Matching these individually is very expensive. However, for fixed strings, there are algorithms for matching sets of strings simultaneously. Consequently, while Snort's default engine still works iteratively, there has been recent work to replace it with a "set-wise" matcher [8, 12].[1] On the other hand, regular expressions give us setwise matching for free: by using the union operator on the individual patterns, we get a new regular expression which effectively combines all of them. The result is a single DFA that again needs $O(n)$ operations to match against an n byte sequence. Only slight modifications have been necessary to extend the interface of Bro's already-existing regular expression matcher to explicitly allow grouping of expressions.

Given the expressiveness and efficiency of regular expressions, there is still a reason why a NIDS might avoid using them: the underlying DFA can grow very large. Fully compiling a regular expression into a DFA leads potentially to an exponential number of DFA states, depending on the particulars of the patterns [18]. Considering the very complex regular expression built by combining all individual patterns, this straight-forward approach could easily be intractable. Our experience with building DFAs for regular expressions matching many hundreds of signatures shows that this is indeed the case. However, it turns out that in practice it is possible to avoid the state/time explosion, as follows.

Instead of pre-computing the DFA, we build the DFA "on-the-fly" during the actual matching [17]. Each time the DFA needs to transit into a state that is not already constructed, we compute the new state and record it for future reuse. This way, we only store DFA states that are actually needed. An important observation is that for n new input characters, we will build at most n new states. Furthermore, we find in practice (§4.3) that for normal traffic the growth is *much* less than linear.

However, there is still a concern that given inauspicious traffic—which may actually be artificially crafted by an attacker—the state construction may eventually consume more memory than we have available. Therefore, we also implemented a memory-bounded DFA *state cache*. Configured with a maximum number of DFA states, it expires old states on a least-recently-used basis. In the sequel, when we mention "Bro with a limited state cache," we are referring to such a bounded set of states (which is a configuration option for our version of Bro), using the default bound of 10,000 states.

Another important point is that it's not necessary to combine all patterns contained in the signature set into a *single* regular expression. Most signatures contain additional constraints like IP address ranges or port numbers that restrict their applicability to a subset of the whole traffic. Based on these constraints, we can build groups of signatures that match the same kind of traffic. By collecting only those patterns into a common regular expression for matching the group, we are able to reduce the size of the resulting DFA drastically. As we show in §4, this gives us a very powerful pattern matcher still efficient enough to cope with high-volume traffic.

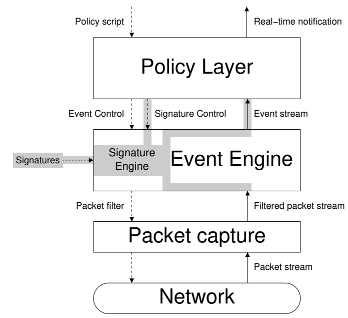
3.2 Improving Alert Quality by Using Context

Though pattern matching is a central part of any signature-based NIDSs, as we discussed above there is potentially great utility in incorporating more context in the system's analysis prior to generating an alert, to ensure that there is indeed something alert-worthy occurring. We can considerably increase the quality of alerts, while simultaneously reducing their quantity, by utilizing knowledge about the current state of the network. Bro is an excellent tool for this as it already keeps a lot of easily accessible state.

The new signature engine is designed to fit nicely into Bro's layered architecture as an adjunct to the protocol analysis event engine (see Figure 1). We have implemented a custom language for defining signatures. It is mostly a superset of other, similar languages, and we describe it in more detail in §3.3. A new component placed within Bro's middle layer matches these signatures against the packet stream. Whenever it finds a match, it inserts a new event into the event stream. The policy layer can then decide how to react. Additionally, we can pass information from the policy layer back into the signature engine to control its operation. A signature can specify a script function to call whenever a particular signature matches. This function can then consult additional context and indicate whether the corresponding event should indeed be generated. We show an example of this later in §3.5.4.

[1]The code of [12] is already contained in the Snort distribution, but not compiled-in by default. This is perhaps due to some subtle bugs, some of which we encountered during our testing as well.

Figure 1: Integrating the signature engine (adapted from [25])



In general, Bro's analyzers follow the communication between two endpoints and extract protocol-specific information. For example, the `HTTP` analyzer is able to extract URIs requested by Web clients (which includes performing general preprocessing such as expanding hex escapes) and the status code and items sent back by servers in reply, whereas the `FTP` analyzer follows the application dialog, matching FTP commands and arguments (such as the names of accessed files) with their corresponding replies. Clearly, this protocol-specific analysis provides significantly more context than does a simple view of the total payload as an undifferentiated byte stream.

The signature engine can take advantage of this additional information by incorporating semantic-level signature matching. For example, the signatures can include the notion of matching against `HTTP` URIs; the URIs to be matched are provided by Bro's `HTTP` analyzer. Having developed this mechanism for interfacing the signature engine with the `HTTP` analyzer, it is now straight forward to extend it to other analyzers and semantic elements (indeed, we timed how long it took to add and debug interfaces for `FTP` and `Finger`, and the two totalled only 20 minutes).

Central to Bro's architecture is its connection management. Each network packet is associated with exactly one connection. This notion of connections allows several powerful extensions to traditional signatures. First of all, Bro reassembles the payload stream of TCP connections. Therefore, we can perform all pattern matching on the actual stream (in contrast to individual packets). While Snort has a preprocessor for TCP session reassembling, it does so by combining several packets into a larger "virtual" packet. This packet is then passed on to the pattern matcher. Because the resulting analysis remains packet-based, it still suffers from discretization problems introduced by focusing on packets, such as missing byte sequences that cross packet boundaries. (See a related discussion in [25] of the problem of matching strings in TCP traffic in the face of possible intruder *evasion* [27].)

In Bro, a signature match does not necessarily correspond to an alert; as with other events, that decision is left to the policy script. Hence, it makes sense to remember which signatures have matched for a particular connection so far. Given this information, it is then possible to specify dependencies between signatures like "signature

A only matches if signature B has already matched," or "if a host matches more than N signatures of type C, then generate an alert." This way, we can for example describe multiple steps of an attack. In addition, Bro notes in which direction of a connection a particular signature has matched, which gives us the notion of *request/reply signatures*: we can associate a client request with the corresponding server reply. A typical use is to differentiate between successful and unsuccessful attacks. We show an example in §3.5.3.

More generally, the policy script layer can associate arbitrary kinds of data with a connection or with one of its endpoints. This means that any information we can deduce from any of Bro's other components can be used to improve the quality of alerts. We demonstrate the power of this approach in §3.5.2.

Keeping per-connection state for signature matching naturally raises the question of *state management*: at some point in time we have to reclaim state from older connections to prevent the system from exhausting the available memory. But again we can leverage the work already being done by Bro. Independently of our signatures, it already performs a sophisticated connection-tracking using various timeouts to expire connections. By attaching the matching state to the already-existing per-connection state, we assure that the signature engine works economically even with large numbers of connections.

3.3 Signature Language

Any signature-based NIDS needs a language for actually defining signatures. For Bro, we had to choose between using an already existing language and implementing a new one. We have decided to create a new language for two reasons. First, it gives us more flexibility. We can more easily integrate the new concepts described in §3.1 and §3.2. Second, for making use of existing signature sets, it is easier to write a converter in some high-level scripting language than to implement it within Bro itself.

Snort's signatures are comprehensive, free and frequently updated. Therefore, we are particularly interested in converting them into our signature language. We have written a corresponding Python script that takes an arbitrary Snort configuration and outputs signatures in Bro's syntax. Figure 2 shows an example of such a conversion.

Figure 2: Example of signature conversion

```
alert tcp any any -> [a.b.0.0/16,c.d.e.0/24] 80
  ( msg:"WEB-ATTACKS conf/httpd.conf attempt";
    nocase; sid:1373; flow:to_server,established;
    content:"conf/httpd.conf"; [...] )
```

(a) Snort

```
signature sid-1373 {
  ip-proto == tcp
  dst-ip == a.b.0.0/16,c.d.e.0/24
  dst-port == 80
  # The payload below is actually generated in a
  # case-insensitive format, which we omit here
  # for clarity.
  payload /.*conf\/httpd\.conf/
  tcp-state established,originator
  event "WEB-ATTACKS conf/httpd.conf attempt"
}%
```

(b) Bro

It turns out to be rather difficult to implement a complete parser for Snort's language. As far as we have been able to determine, its syntax and semantics are not fully documented, and in fact often only defined by the source code. In addition, due to different internals of Bro and Snort, it is sometimes not possible to keep the exact semantics of the signatures. We return to this point in §4.2.

As the example in Figure 2 shows, our signatures are defined by means of an identifier and a set of attributes. There are two main types of attributes: *(i) conditions* and *(ii) actions*. The conditions define *when* the signature matches, while the actions declare *what to do* in the case of a match. Conditions can be further divided into four types: *header*, *content*, *dependency*, and *context*.

Header conditions limit the applicability of the signature to a subset of traffic that contains matching packet headers. For TCP, this match is performed only for the first packet of a connection. For other protocols, it is done on each individual packet. In general, header conditions are defined by using a `tcpdump`-like [33] syntax (for example, `tcp[2:2] == 80` matches `TCP` traffic with destination port 80). While this is very flexible, for convenience there are also some short-cuts (e.g., `dst-port == 80`).

Content conditions are defined by regular expressions. Again, we differentiate two kinds of conditions here: first, the expression may be declared with the `payload` statement, in which case it is matched against the raw packet payload (reassembled where applicable). Alternatively, it may be prefixed with an analyzer-specific label, in which case the expression is matched against the data as extracted by the corresponding analyzer. For example, the `HTTP` analyzer decodes requested URIs. So, `http /(etc\/(passwd|shadow)/` matches any request containing either `etc/passwd` or `etc/shadow`.

Signature conditions define dependencies between signatures. We have implemented `requires-signature`, which specifies another signature that has to match on the same connection first, and `requires-reverse-signature`, which additionally requires the match to happen for the other direction of the connection. Both conditions can be negated to match only if another signature does *not* match.

Finally, context conditions allow us to pass the match decision on to various components of Bro. They are only evaluated if all other conditions have already matched. For example, we have implemented a `tcp-state` condition that poses restrictions on the current state of the `TCP` connection, and `eval`, which calls an arbitrary script policy function.

If all conditions are met, the actions associated with a signature are executed: `event` inserts a `signature_match` event into the event stream, with the value of the event including the signature identifier, corresponding connection, and other context. The policy layer can then analyze the signature match.

3.4 Snort's Signature Set

Snort comes with a large set of signatures, with 1,273 enabled by default [2]. Unfortunately, the default configuration turns out to generate a lot of false positives. In addition, many alerts belong to failed exploit attempts executed by attackers who scan networks for vulnerable hosts. As noted above, these are general problems of signature-based systems.

The process of selectively disabling signatures that are not applicable to the local environment, or "tuning," takes time, knowledge and experience. With respect to Snort, a particular problem is that many of its signatures are too general. For example, Snort's signature #1560:

```
alert tcp $EXTERNAL_NET any
      -> $HTTP_SERVERS $HTTP_PORTS
  (msg:"WEB-MISC /doc/ access";
  uricontent:"/doc/"; flow:to_server,established;
  nocase; sid:1560; [...])
```

searches for the string `/doc/` within URIs of `HTTP` requests. While this signature is indeed associated with a particular vulnerability (`CVE-1999-0678` [9]), it only makes sense to use it if you have detailed knowledge about your site (for example, that there is no valid document whose path contains the string `/doc/`). Otherwise, the probability of a signature match reflecting a false alarm is much higher than that it indicates an attacker exploiting an old vulnerability.

Another problem with Snort's default set is the presence of overlapping signatures for the same exploit. For example, signatures #1536, #1537, #1455, and #1456 (the latter is disabled by default) all search for `CVE-2000-0432`, but their patterns differ in the amount of detail. In addition, the vulnerability IDs given in Snort's signatures are not always correct. For example, signature #884 references `CVE-1999-0172` and Buqtraq [6] ID #1187. But the latter corresponds to `CVE-2000-0411`.

As already noted, we cannot expect to avoid these limitations of Snort's signatures by just using them semantically unmodified in Bro. For example, although we convert the Snort's fixed strings into Bro's regular expressions, naturally they still represent fixed sets of characters. Only manual editing would give us the additional power of regular expressions. We give an example for such an improvement in §3.5.1.

3.5 The Power of Bro Signatures

In this section, we show several examples to convey the power provided by our signatures. First, we demonstrate how to define more "tight" signatures by using regular expressions. Then, we show how to identify failed attack attempts by considering the set of software a particular server is runnning (we call this its *vulnerability profile* and incorporate some ideas from [22] here) as well as the response of the server. We next demonstrate modelling an attack in multiple steps to avoid false positives, and finally show how to use alert-counting for identifying *exploit scans*. We note that none of the presented examples are supported by Snort without extending its core significantly (e.g. by writing new plug-ins).

3.5.1 *Using Regular Expressions*

Regular expressions allow far more flexibility than fixed strings. Figure 3 (a) shows a Snort signature for `CVE-1999-0172` that generates a large number of false positives at Saarland University's border router. (See §4.1 for a description of the university.) Figure 3 (b) shows a corresponding Bro signature that uses a regular expression to identify the exploit more reliably. `CVE-1999-0172` describes a vulnerability of the `formmail` CGI script. If an attacker constructs a string of the form "`...; <shell-cmds>`" (a | instead of the `;` works as well), and passes it on as argument of the `recipient` CGI parameter, vulnerable formmails will execute the included shell commands. Because CGI parameters can be given in arbitrary order, the Snort signature has to rely on identifying the `formmail` access by its own. But by using a regular expression, we can explicitly define that the `recipient` parameter has to contain a particular character.

Figure 3: Two signatures for CVE-1999-0172

```
alert tcp any any -> a.b.0.0/16 80
  (msg:"WEB-CGI formmail access";
   uricontent:"/formmail";
   flow:to_server,established;
   nocase; sid:884; [...])
```

(a) Snort using a fixed string

```
signature formmail-cve-1999-0172 {
  ip-proto == tcp
  dst-ip == a.b.0.0/16
  dst-port = 80
  # Again, actually expressed in a
  # case-insensitive manner.
  http /.*formmail.*\?.*recipient=[^&]*[;|]/
  event "formmail shell command"
}
```

(b) Bro using a regular expression

3.5.2 *Vulnerability Profiles*

Most exploits are aimed at particular software, and usually only some versions of the software are actually vulnerable. Given the overwhelming number of alerts a signature-matching NIDS can generate, we may well take the view that the only attacks of interest are those that actually have a chance of succeeding. If, for example, an `IIS` exploit is tried on a Web server running `Apache`, one may not even care. [23] proposes to prioritize alerts based on this kind of vulnerability information. We call the set of software versions that a host is running its *vulnerability profile*. We have implemented this concept in Bro. By protocol analysis, it collects the profiles of hosts on the network, using version/implementation information that the analyzer observes. Signatures can then be restricted to certain versions of particular software.

As a proof of principle, we have implemented vulnerability profiles for `HTTP` servers (which usually characterize themselves via the `Server` header), and for `SSH` clients and servers (which identify their specific versions in the clear during the initial protocol handshake). We intend to extend the software identification to other protocols.

We aim in future work to extend the notion of developing a profile beyond just using protocol analysis. We can *passively fingerprint* hosts to determine their operating system version information by observing specific idiosyncrasies of the header fields in the traffic they generate, similar to the probing techniques described in [13], or we can separately or in addition employ *active* techniques to explicitly map the properties of the site's hosts and servers [31]. Finally, in addition to automated techniques, we can implement a configuration mechanism for manually entering vulnerability profiles.

3.5.3 *Request/Reply Signatures*

Further pursuing the idea to avoid alerts for failed attack attempts, we can define signatures that take into account both directions of a connection. Figure 4 shows an example. In operational use, we see a lot of attempts to exploit `CVE-2001-0333` to execute the Windows command interpreter `cmd.exe`. For a failed attempt, the server typically answers with a $4xx$ `HTTP` reply code, indicating an error.[2] To ignore these failed attempts, we first define one signature, `http-error`, that recognizes such replies. Then we define a second signature, `cmdexe-success`, that matches only if `cmd.exe` is contained in the requested URI (case-insensitive) and the server does *not* reply with an error. It's not possible to define this kind of signature in Snort, as it lacks the notion of associating both directions of a connection.

Figure 4: Request/reply signature

```
signature cmdexe-success {
  ip-proto == tcp
  dst-port == 80
  http /.*[cC][mM][dD]\.[eE][xX][eE]/
  event "WEB-IIS cmd.exe success"
  requires-signature-opposite ! http-error
  tcp-state established
}
signature http-error {
 ip-proto == tcp
 src-port == 80
 payload /.*HTTP\/1\.. *4[0-9][0-9]/
 event "HTTP error reply"
 tcp-state established
 }
```

[2] There are other reply codes that reflect additional types of errors, too, which we omit for clarity.

3.5.4 *Attacks with Multiple Steps*

An example of an attack executed in two steps is the infection by the `Apache/mod_ssl` worm [7] (also known as `Slapper`), released in September 2002. The worm first probes a target for its potential vulnerability by sending a simple `HTTP` request and inspecting the response. It turns out that the request it sends is in fact in violation of the `HTTP` 1.1 standard [11] (because it does not include a `Host` header), and this idiosyncracy provides a somewhat "tight" signature for detecting a `Slapper` probe.

If the server identifies itself as `Apache`, the worm then tries to exploit an `OpenSSL` vulnerability on `TCP` port 443. Figure 5 shows two signatures that only report an alert if these steps are performed for a destination that runs a vulnerable `OpenSSL` version. The first signature, `slapper-probe`, checks the payload for the illegal request. If found, the script function `is_vulnerable_to_slapper` (omitted here due to limited space, see [2]) is called. Using the vulnerability profile described above, the function evaluates to true if the destination is known to run `Apache` as well as a vulnerable `OpenSSL` version.[3] If so, the signature matches (depending on the configuration this may or may not generate an alert by itself). The header conditions of the second signature, `slapper-exploit`, match for any `SSL` connection into the specified network. For each, the signature calls the script function `has_slapper_probed`. This function generates a signature match if `slapper-probe` has already matched for the same source/destination pair. Thus, Bro alerts if the combination of probing for a vulnerable server, plus a potential follow-on exploit of the vulnerability, has been seen.

Figure 5: Signature for `Apache/mod_ssl` worm

```
signature slapper-probe {
  ip-proto == tcp
  dst-ip == x.y.0.0/16 # sent to local net
  dst-port == 80
  payload /.*GET \/ HTTP\/1\.1\x0d\x0a\x0d\x0a/
  eval is_vulnerable_to_slapper # call policy fct.
  event "Vulner. host possibly probed by Slapper"
}
signature slapper-exploit {
  ip-proto == tcp
  dst-ip == x.y.0.0/16
  dst-port == 443 # 443/tcp = SSL/TLS
  eval has_slapper_probed # test: already probed?
  event "Slapper tried to exploit vulnerable host"
}
```

3.5.5 *Exploit Scanning*

Often attackers do not target a particular system on the Internet, but probe a large number of hosts for vulnerabilities (*exploit scanning*). Such a scan can be executed either *horizontally* (several hosts are probed for a particular exploit), *vertically* (one host is probed for several exploits), or both. While, by their own, most of these probes are usually low-priority failed attempts, the scan itself is an important event. By simply counting the number signature alerts per source address (horizontal) or per source/destination pair (vertical), Bro can readily identify such scans. We have implemented this with a policy script which generates alerts like:

```
a.b.c.d triggered 10 signatures on host e.f.g.h
i.j.k.l triggered signature sid-1287 on 100 hosts
m.n.o.p triggered signature worm-probe on 500 hosts
q.r.s.t triggered 5 signatures on host u.v.x.y
```

[3] Note that it could instead implement a more conservative policy, and return true *unless* the destination is known to not run a vulnerable version of OpenSSL/Apache.

4. EVALUATION

Our approach for evaluating the effectiveness of the signature engine is to compare it to Snort in terms of run-time performance and generated alerts, using semantically equivalent signature sets. We note that we do not evaluate the concept of conceptual signatures by itself. Instead, as a first step, we validate that our implementation is capable of acting as an effective substitute for the most-widely deployed NIDS even when we do not use any of the advanced features it provides. Building further on this base by thoroughly evaluating the actual power of contextual signatures when deployed operationally is part of our ongoing work.

During our comparision of Bro and Snort, we found several peculiarities that we believe are of more general interest. Our results stress that the performance of a NIDS can be very sensitive to semantics, configuration, input, and even underlying hardware. Therefore, after discussing our test data, we delve into these in some detail. Keeping these limitations in mind, we then assess the overall performance of the Bro signature engine.

4.1 Test Data

For our testing, we use two traces:

`USB-Full` A 30-minute trace collected at Saarland University, Germany (`USB-Full`), consisting of all traffic (including packet contents) except for three high-volume peer-to-peer applications (to reduce the volume). The university has 5,500 internal hosts, and the trace was gathered on its 155 Mbps access link to the Internet. The trace totals 9.8 GB, 15.3M packets, and 220K connections. 35% of the trace packets belong to `HTTP` on port 80, 19% to eDonkey on port 4662, and 4% to `ssh` on port 22, with other individual ports being less common than these three (and the high-volume peer-to-peer that was removed).

`LBL-Web` A two-hour trace of HTTP client-side traffic, including packet contents, gathered at the Lawrence Berkeley National Laboratory (LBL), Berkeley, USA (`LBL-Web`). The laboratory has 13,000 internal hosts, and the trace was gathered on its Gbps access link to the Internet. The trace totals 667MB, 5.5M packets, and 596K connections.

Unless stated otherwise, we performed all measurements on 550MHz Pentium-3 systems containing ample memory (512MB or more). For both Snort and Bro's signature engine, we used Snort's default signature set. We disabled Snort's "experimental" set of signatures as some of the latest signatures use new options which are not yet implemented in our conversion program. In addition, we disabled Snort signature #526, `BAD TRAFFIC data in TCP SYN packet`. Due to Bro matching stream-wise instead of packet-wise, it generates thousands of false positives. We discuss this in §4.2. In total, 1,118 signatures are enabled. They contain 1,107 distinct patterns and cover 89 different service ports. 60% of the signatures cover `HTTP` traffic. For `LBL-Web`, only these were activated.

For Snort, we enabled the preprocessors for IP defragmentation, `TCP` stream reassembling on its default ports, and HTTP decoding. For Bro, we have turned on `TCP` reassembling for the same ports (even if otherwise Bro would not reassemble them because none of the usual event handlers indicated interest in traffic for those ports), enabled its memory-saving configuration ("`@load reduce-memory`"), and used an `inactivity_timeout` of 30 seconds (in correspondence with Snort's default session timeout). We configured both systems to consider all packets contained in the traces. We used the version 1.9 branch of Snort, and version 0.8a1 of Bro.

4.2 Difficulties of Evaluating NIDSs

The evaluation of a NIDS is a challenging undertaking, both in terms of assessing attack recognition and in terms of assessing performance. Several efforts to develop objective measures have been made in the past (e.g., [21, 22, 15]), while others stress the difficulties with such approaches [24]. During our evaluation, we encountered several additional problems that we discuss here. While these arose in the specific context of comparing Snort and Bro, their applicability is more general.

When comparing two NIDSs, differing internal semantics can present a major problem. Even if both systems basically perform the same task—capturing network packets, rebuilding payload, decoding protocols—that task is sufficiently complex that it is almost inevitable that the systems will do it somewhat differently. When coupled with the need to evaluate a NIDS over a *large* traffic trace (millions of packets), which presents ample opportunity for the differing semantics to manifest, the result is that understanding the significance of the disagreement between the two systems can entail significant manual effort.

One example is the particular way in which `TCP` streams are reassembled. Due to state-holding time-outs, ambiguities (see [27, 16] and [25] for discussion of how these occur for benign reasons in practice) and non-analyzed packets (which can be caused by packet filter drops, or by internal sanity checks), TCP stream analyzers will generally wind up with slightly differing answers for corner cases.

Snort, for example, uses a preprocessor that collects a number of packets belonging to the same session until certain thresholds are reached and then combines them into "virtual" packets. The rest of Snort is not aware of the reassembling and still only sees packets. Bro, on the other hand, has an intrinsic notion of a data stream. It collects as much payload as needed to correctly reconstruct the next in-sequence chunk of a stream and passes these data chunks on as soon as it is able to. The analyzers are aware of the fact that they get their data chunk-wise, and track their state across chunks. They are *not* aware of the underlying packetization that lead to those chunks. While Bro's approach allows true stream-wise signatures, it also means that the signature engine loses the notion of "packet size": packets and session payload are decoupled for most of Bro's analyzers. However, Snort's signature format includes a way of specifying the packet size. Our signature engine must fake up an equivalent by using the size of the first matched payload chunk for each connection, which can lead to differing results.

Another example of differing semantics comes from the behavior of protocol analyzers. Even when two NIDS both decode the same protocol, they will differ in the level-of-detail and their interpretation of protocol corner cases and violations (which, as mentioned above, are in fact seen in non-attack traffic [25]). For example, both Bro and Snort extract URIs from `HTTP` sessions, but they do not interpret them equally in all situations. Character encodings within URIs are sometimes decoded differently, and neither contains a full Unicode decoder. The anti-IDS tool Whisker [37] can actively exploit these kinds of deficiencies. Similarly, Bro decodes pipelined HTTP sessions; Snort does not (it only processes the first URI in a series of pipelined HTTP requests).

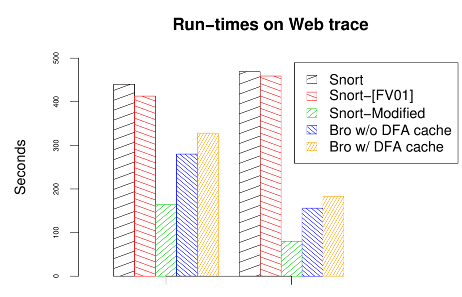
Usually, the details of a NIDS can be controlled by a number of options. But frequently for a Bro option there is no equivalent Snort option, and vice versa. For example, the amount of memory used by Snort's `TCP` reassembler can be bounded to a fixed value. If this limit is reached, old data is expired aggressively. Bro relies solely on time-outs. Options like these often involve time-memory trade-offs. The more memory we have, the more we can spend for Snort's reassembler, and the larger we can make Bro's time-outs. But how to choose the values, so that both will utilize the same amount of memory? And even if we do, how to arrange that both expire the same old data? The hooks to do so simply aren't there.

The result of these differences is differing views of the same network data. If one NIDS reports an alert while the other does not, it may take a surprisingly large amount of effort to tell which one of them is indeed correct. More fundamentally, this depends on the definition of "correct," as generally both are correct within their own semantics. From a user's point of the view, this leads to different alerts even when both systems seem to use the same signatures. From an evaluator's point of view, we have to *(i)* grit our teeth and be ready to spend substantial effort in tracking down the root cause when validating the output of one tool versus another, and *(ii)* be very careful in how we frame our assessment of the differences, because there is to some degree a fundamental problem of "comparing apples and oranges".

The same applies for measuring performance in terms of efficiency. If two systems do different things, it is hard to compare them fairly. Again, the `HTTP` analyzers of Snort and Bro illustrate this well. While Snort only extracts the first URI from each packet, Bro decodes the full `HTTP` session, including tracking multiple requests and replies (which entails processing the numerous ways in which HTTP delimits data entities, including "multipart MIME" and "chunking"). Similarly, Bro provides much more information at various other points than the corresponding parts of Snort.

But there are still more factors that influence performance. Even if one system seems to be significantly faster than another, this can change by modifying the input or even the underlying hardware. One of our main observations along these lines is that the performance of NIDSs can depend heavily on the particular input trace. On a Pentium-3 system, Snort needs 440 CPU seconds for the trace `LBL-Web` (see Figure 6). This only decreases by 6% when using the set-wise pattern matcher of [12]. In addition, we devised a small modification to Snort that, compared to the original version, speeds it up by factor of 2.6 for this particular trace. (The modification is an enhancement to the set-wise matcher: the original implementation first performs a set-wise search for all of the possible strings, caching the results, and then iterates through the lists of signatures, looking up for each in turn whether its particular strings were matched. Our modification uses the result of the set-wise match to identify potential matching signatures directly if the corresponding list is large, avoiding the iteration.)

Figure 6: Run-times on different hardware



Using the trace `USB-Full`, however, the improvement realized by our modified set-wise matcher for Snort is only a factor of 1.2. Even more surprisingly, on a trace from another environment (a research laboratory with 1,500 workstations and supercomputers), the original version of Snort is *twice as fast* as the set-wise implementation of [12] (148 CPU secs vs. 311 CPU secs), while our patched version lies in between (291 CPU secs). While the reasons remain to be discovered in Snort's internals, this demonstrates the difficulty of finding representative traffic as proposed, for example, in [15].

Furthermore, relative performance does not only depend on the input but even on the underlying hardware. As described above, the original Snort needs 440 CPU seconds for LBL-Web on a Pentium-3 based system. Using exactly the same configuration and input on a Pentium-4 based system (1.5GHz), it actually takes 29 CPU seconds *more*. But now the difference between stock Snort and our modified version is a factor of 5.8! On the same system, Bro's run-time *decreases* from 280 to 156 CPU seconds.[4]

Without detailed hardware-level analysis, we can only guess why Snort suffers from the upgrade. To do so, we ran valgrind's [34] cache simulation on Snort. For the second-level data cache, it shows a miss-rate of roughly 10%. The corresponding value for Bro is below 1%. While we do not know if valgrind's values are airtight, they could at least be the start of an explanation. We have heard other anecdotal comments that the Pentium-4 performs quite poorly for applications with lots of cache-misses. On the other hand, by building Bro's regular expression matcher incrementally, as a side effect the DFA tables will wind up having memory locality that somewhat reflects the dynamic patterns of the state accesses, which will tend to decrease cache misses.

4.3 Performance Evaluation

We now present measurements of the performance of the Bro signature engine compared with Snort, keeping in mind the difficulties described above. Figure 7 shows run-times on trace subsets of different length for the USB-Full trace. We show CPU times for the original implementation of Snort, for Snort using [12] (virtually no difference in performance), for Snort modified by us as described in the previous section, for Bro with a limited DFA state cache, and for Bro without a limited DFA state cache. We see that our modified Snort runs 18% faster than the original one, while the cache-less Bro takes about the same amount of time. Bro with a limited state cache needs roughly a factor of 2.2 more time.

We might think that the discrepancy between Bro operating with a limited DFA state cache and it operating with unlimited DFA state memory is due to it having to spend considerable time recomputing states previously expired from the limited cache. This, however, turns out not to be the case. Additional experiments with essentially infinite cache sizes indicate that the performance decrease is due to the additional overhead of maintaining the cache.

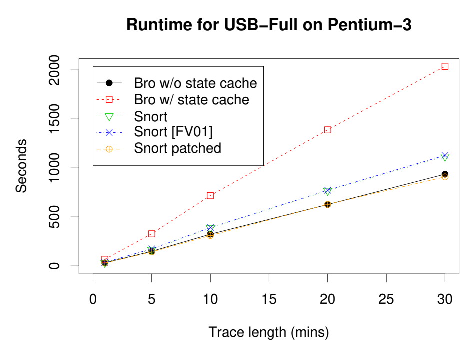
While this looks like a significant impact, we note that it is not clear whether the space savings of a cache is in fact needed in operational use. For this trace, only 2,669 DFA states had to be computed, totaling roughly 10MB. When running Bro operationally for a day at the university's gateway, the number of states rapidly climbs to about 2,500 in the first hour, but then from that point on only slowly rises to a bit over 4,000 by the end of the day.

A remaining question, however, is whether an attacker could create traffic specifically tailored to enlarge the DFAs (a "state-holding" attack on the IDS), perhaps by sending a stream of packets that nearly trigger each of the different patterns. Additional research is needed to further evaluate this threat.

Comparing for USB-Full the alerts generated by Snort to the signature matches reported by Bro, all in all we find very good agreement. The main difference is the way they report a match. By design, Bro reports all matching signatures, but each one only once per connection. This is similar to the approach suggested in [10]. Snort, on the other hand, reports the first matching signature for each packet, independently of the connection it belongs to. This makes it difficult to compare the matches. We account for these difference by comparing connections for which at least one match is generated by either system. With USB-Full, we get 2,065 matches by Bro in total on 1,313 connections. Snort reports 4,147 alerts. When counting each alert only once per connection, Snort produces 1,320 on 1,305 connections.[5] There are 1,296 connections for which both generate at least one alert, and 17 (9) for which Bro (Snort) reports a match but not Snort (Bro).

Looking at individual signatures, we see that Bro misses 10 matches of Snort. 5 of them are caused by Snort ID #1013 (WEB-IIS fpcount access). The corresponding connections contain several requests, but an idle time larger than the defined inactivity_timeout of 30 seconds. Therefore, Bro flushes the state before it can encounter the match which would happen later in the session. On the other hand, Bro reports 41 signature matches for connections for which Snort does not report anything. 37 of them are Web signatures. The discrepancy is due to different TCP stream semantics. Bro and Snort have slightly different definitions of when a session is established. In addition, the semantic differences between stream-wise and packet-wise matching discussed in §4.2 cause some of the additional alerts.

Figure 7: Run-time comparison on 550MHz Pentium-3



We have done similar measurements with LBL-Web. Due to limited space, we omit the corresponding plot here. While the original Snort takes 440 CPU seconds for the trace, Bro without (with) a limited state cache needs 280 (328) CPU seconds, and Snort as modified by us needs only 164 CPU seconds. While this suggests room for improvement in some of Bro's internal data structures, Bro's matcher still compares quite well to the typical Snort configuration.

For this trace, Bro (Snort) reports 2,764 (2,049) matches in total. If we count Snort's alerts only once per connection, there are 1,472 of them. There are 1,395 connections for which both report at least one alert. For 133 (69) connections, Bro (Snort) reports a match but Snort (Bro) does not. Again, looking at individual signatures, Bro misses 73 of Snort's alerts. 25 of them are matches of Snort signature #1287 (WEB-IIS scripts access). These are all caused by the same host. The reason is packets missing from the trace, which, due to a lack of in-order sequencing, prevent the TCP stream from being reassembled by Bro. Another 19 are due to signature #1287 (CodeRed v2 root.exe access). The ones of these we inspected further were due to premature server-side resets, which Bro correctly identifies as the end of the corresponding connections, while Snort keeps matching on the traffic still being send by the client. Bro reports 186 signature matches for connections for which Snort does not report a match at all. 68 of these connections simultaneously trigger three signatures (#1002, #1113, #1287). 46

[4] This latter figure corresponds to about 35,000 packets per second, though we strongly argue that measuring performance in PPS rates implies undue generality, since, as developed above, the specifics of the packets make a great difference in the results.

[5] Most of the duplicates are ICMP Destination Unreachable messages. Using Bro's terminology, we define all ICMP packets between two hosts as belonging to one "connection."

are due to simultaneous matches of signatures #1087 and #1242. Looking at some of them, one reason is SYN-packets missing from the trace. Their absence leads to different interpretations of established sessions by Snort and Bro, and therefore to different matches.

5. CONCLUSIONS

In this work, we develop the general notion of *contextual signatures* as an improvement on the traditional form of string-based signature-matching used by NIDS. Rather than matching fixed strings in isolation, contextual signatures augment the matching process with both low-level context, by using regular expressions for matching rather than simply fixed strings, and high-level context, by taking advantage of the rich, additional semantic context made available by Bro's protocol analysis and scripting language.

By tightly integrating the new signature engine into Bro's event-based architecture, we achieve several major improvements over other signature-based NIDSs such as Snort, which frequently suffer from generating a huge number of alerts. By interpreting a signature-match only as an event, rather than as an alert by itself, we are able to leverage Bro's context and state-management mechanisms to improve the quality of alerts. We showed several examples of the power of this approach: matching requests with replies, recognizing exploit scans, making use of vulnerabilty profiles, and defining dependencies between signatures to model attacks that span multiple connections. In addition, by converting the freely available signature set of Snort into Bro's language, we are able to build upon existing community efforts.

As a baseline, we evaluated our signature engine using Snort as a reference, comparing the two systems in terms of both run-time performance and generated alerts using the signature set archived at [2]. But in the process of doing so, we encountered several general problems when comparing NIDSs: differing internal semantics, incompatible tuning options, the difficulty of devising "representative" input, and extreme sensitivity to hardware particulars. The last two are particularly challenging, because there are no *a priori* indications when comparing performance on one particular trace and hardware platform that we might obtain very different results using a different trace or hardware platform. Thus, we must exercise great caution in interpreting comparisons between NIDSs.

Based on this work, we are now in the process of deploying Bro's contextual signatures operationally in several educational, research and commercial enviroments.

Finally, we have integrated our work into version 0.8 of the Bro distribution, freely available at [5].

6. ACKNOWLEDGMENTS

We would like to thank the Lawrence Berkeley National Laboratory (LBL), Berkeley, USA; the National Energy Research Scientific Computing Center (NERSC), Berkeley, USA; and the Saarland University, Germany. We are in debt to Anja Feldmann for making this work possible. Finally, we would like to thank the anonymous reviewers for their valuable suggestions.

7. REFERENCES

[1] arachNIDS. http://whitehats.com/ids/.

[2] Web archive of versions of software and signatures used in this paper. http://www.net.in.tum.de/~robin/ccs03.

[3] S. Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Transactions on Information and System Security*, 3(3):186–205, August 2000.

[4] R. G. Bace. *Intrusion Detection*. Macmillan Technical Publishing, Indianapolis, IN, USA, 2000.

[5] Bro: A System for Detecting Network Intruders in Real-Time. http://www.icir.org/vern/bro-info.html.

[6] Bugtraq. http://www.securityfocus.com/bid/1187.

[7] CERT Advisory CA-2002-27 Apache/mod_ssl Worm. http://www.cert.org/advisories/CA-2002-27.html.

[8] C. J. Coit, S. Staniford, and J. McAlerney. Towards Faster Pattern Matching for Intrusion Detection or Exceeding the Speed of Snort. In *Proc. 2nd DARPA Information Survivability Conference and Exposition*, June 2001.

[9] Common Vulnerabilities and Exposures. http://www.cve.mitre.org.

[10] H. Debar and B. Morin. Evaluation of the Diagnostic Capabilities of Commercial Intrusion Detection Systems. In *Proc. Recent Advances in Intrusion Detection*, number 2516 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[11] R. F. et. al. Hypertext transfer protocol – http/1.1. Request for Comments 2616, June 1999.

[12] M. Fisk and G. Varghese. Fast Content-Based Packet Handling for Intrusion Detection. Technical Report CS2001-0670, UC San Diego, May 2001.

[13] Fyodor. Remote OS detection via TCP/IP Stack Finger Printing. *Phrack Magazine*, 8(54), 1998.

[14] J. Haines, L. Rossey, R. Lippmann, and R. Cunnigham. Extending the 1999 Evaluation. In *Proc. 2nd DARPA Information Survivability Conference and Exposition*, June 2001.

[15] M. Hall and K. Wiley. Capacity Verification for High Speed Network Intrusion Detection Systems. In *Proc. Recent Advances in Intrusion Detection*, number 2516 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[16] M. Handley, C. Kreibich, and V. Paxson. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proc. 10th USENIX Security Symposium*, Washington, D.C., August 2001.

[17] J. Heering, P. Klint, and J. Rekers. Incremental generation of lexical scanners. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(4):490–520, 1992.

[18] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[19] K. Jackson. Intrusion detection system product survey. Technical Report LA-UR-99-3883, Los Alamos National Laboratory, June 1999.

[20] U. Lindqvist and P. A. Porras. Detecting computer and network misuse through the production-based expert system toolset (P-BEST). In *Proc. IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 1999.

[21] R. Lippmann, R. K. Cunningham, D. J. Fried, I. Graf, K. R. Kendall, S. E. Webster, and M. A. Zissman. Results of the 1998 DARPA Offline Intrusion Detection Evaluation. In *Proc. Recent Advances in Intrusion Detection*, 1999.

[22] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das. The 1999 DARPA off-line intrusion detection evaluation. *Computer Networks*, 34(4):579–595, October 2000.

[23] R. Lippmann, S. Webster, and D. Stetson. The Effect of Identifying Vulnerabilities and Patching Software on the Utility of Network Intrusion Detection. In *Proc. Recent Advances in Intrusion Detection*, number 2516 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[24] J. McHugh. Testing Intrusion detection systems: A critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by Lincoln Laboratory. *ACM Transactions on Information and System Security*, 3(4):262–294, November 2000.

[25] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–2463, 1999.

[26] P. A. Porras and P. G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *National Information Systems Security Conference*, Baltimore, MD, October 1997.

[27] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., January 1998.

[28] M. J. Ranum, K. Landfield, M. Stolarchuk, M. Sienkiewicz, A. Lambeth, and E. Wall. Implementing a generalized tool for network monitoring. In *Proc. 11th Systems Administration Conference (LISA)*, 1997.

[29] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proc. 13th Systems Administration Conference (LISA)*, pages 229–238. USENIX Association, November 1999.

[30] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *Proc. 8th USENIX Security Symposium*. USENIX Association, August 1999.

[31] U. Shankar and V. Paxson. Active Mapping: Resisting NIDS Evasion Without Altering Traffic. In *Proc. IEEE Symposium on Security and Privacy*, 2003.

[32] Steven T. Eckmann. Translating Snort rules to STATL scenarios. In *Proc. Recent Advances in Intrusion Detection*, October 2001.

[33] tcpdump. http://www.tcpdump.org.

[34] Valgrind. http://developer.kde.org/~sewardj.

[35] G. Vigna, S. Eckmann, and R. Kemmerer. The STAT Tool Suite. In *Proc. 1st DARPA Information Survivability Conference and Exposition*, Hilton Head, South Carolina, January 2000. IEEE Computer Society Press.

[36] G. Vigna and R. A. Kemmerer. Netstat: A network-based intrusion detection system. *Journal of Computer Security*, 7(1):37–71, 1999.

[37] Whisker. http://www.wiretrip.net/rfp.