

Systematic Signature Engineering by Re-use of Snort Signatures

Sebastian Schmerl¹, Hartmut Koenig², Ulrich Flegel^{3,i}, Michael Meier⁴, René Rietz⁵

^{1,2,5} Brandenburg University
of Technology Cottbus
03013 Cottbus, Germany
{sbs¹, koenig², rrietz⁵}@informatik.tu-cottbus.de

³ SAP Research
Vincenz-Prießnitz-Str. 1
76131 Karlsruhe, Germany
ulrich.flegel@sap.com

⁴ University of Dortmund
44221 Dortmund,
Germany
michael.meier@udo.edu

Abstract

Most intrusion detection systems apply the misuse detection approach. Misuse detection compares recorded audit data with predefined patterns denoted as signatures. A signature is usually empirically engineered based on experience and expert knowledge. This induces relatively long development times for novel signatures causing inappropriate long vulnerability windows. Methods for a systematic engineering have been scarcely reported so far. Approaches for an automated re-use of design and modeling decisions of available signatures also do not exist. In this paper we present an approach for systematic engineering of signatures which is based on the re-use of existing signatures.

1 Motivation

The growing dependencies of social processes on IT infrastructures as well as their increasing complexity imply a large potential of threats. Intrusion detection systems (IDS) are a fundamental reactive measure to counter threats that could not be prevented. They provide means to automatically detect occurred security violations and to trigger appropriate countermeasures. IDSs apply two complementary approaches: anomaly and misuse detection. Misuse detection focuses on the detection of attacks in audit trails described by patterns of known security violations, i.e. so-called signatures. Since misuse detection systems take a key position in productive systems we are focusing on optimizing the process of providing signatures for these systems. The effectiveness of misuse detection strongly depends on the conciseness and the timeliness of the signatures used. Imprecise signatures heavily limit the detection capability of the IDSs and lead to false positives. The reasons for this detection inaccuracy must be sought rather in the signature engineering process itself than in qualitative restrictions of the audit functions. In particular, the engineering of signatures starting from given exploits often is a sore spot. The engineering of signa-

tures relies mostly on expert knowledge and experience. Methods for systematic signature engineering have not been reported yet.

In this paper we present a general method for deriving new signatures from a given set of signatures. The principle of the approach consists of generalizing a given signature by removing selected parts. Although thus the detection accuracy is reduced the number of attacks which can be detected is increasing. This process is repeated until traces of the attack under investigation are detected. The result of this process is a tree of abstracted signatures. By assigning weights to the different abstractions the signature engineer can select those signatures or fragments which are most appropriate for detecting the new attack. A specific starting point is valuable because it provides the signature engineer with information about similar attacks and related detection rationale as well as design decisions. Ideally the new signature can be adopted from the tree by adding constraints to one of the abstracted signatures that already detect the attack.

In the field of misuse detection two classes of attacks can be distinguished related to their operation sequence and the expense required for their detection: *multi step attacks* and *single step attacks*. The detection of multi step attacks requires a correlation among several log entries, therefore multi step signatures are typically specified in highly expressive languages (cf. [8]). However, today many IDSs, and especially network IDSs, are still geared towards *single step attack signatures*, i.e. they use signature specification languages with a lower expressiveness. In order to prove the feasibility of our approach for single step attacks we choose the signature specification language of the *Snort* IDS as an example. The reason for using *Snort* is mainly due to its widespread deployment and the huge signature knowledge base maintained by the user community. This provides a good basis for validating the approach and ensures that the results are relevant in practice and widely applicable.

The paper is organized as follows. In Section 2 we describe the general principle of the approach and related work. Next in Section 3, we adapt the procedure to

ⁱThe work of this author was funded by means of the German Federal Ministry of Economy and Technology under the promotional reference "01MQ07012". The author takes the responsibility for his contribution.

the signature description language of *Snort*. Section 4 describes the evaluation method using an official set of the *Snort-VRT Certified Rules*. We present results of applying the approach in two examples in Sect. 5 and conclude with final remarks and an overview of future work in Sect. 6.

2 Deriving Signatures by Re-use

The engineering of a signature for a novel attack can be divided into the following steps: (1) execution of the attack on a dedicated system to record its traces in an audit trail. Traces represent security relevant events (basic events) which are observed by sensors. (2) The signature engineer investigates the basic events, identifies the relevant events, and (3) derives step by step the new signature considering the way how the attack exploits the vulnerability. An important aspect in this context is the detection of patterns which allow the IDS to identify attack traces. (4) After specifying the signature its correctness and conciseness must be validated. If required the signature has to be corrected or changed.

The signature engineering process is time-consuming, in particular for the phases (3) and (4). If knowledge of existing signature designs can be re-used the time exposure for the signature engineering will be reduced significantly and the quality of the new signature will be improved. In addition, the re-use of approved, i.e. validated, signature fragments will also considerably shorten the expensive validation phase.

The idea of re-use is already known from software engineering. There it is applied to reduce the time for software developments by re-using design and implementation decisions of existing software. Only a few approaches have been published that deal with this subject in the context of signature engineering. Cheung et al. propose a method for simplifying the signature design by applying attack models [3]. This approach corresponds to the design patterns of software engineering [4]. It allows the re-use of architectural design decisions. The re-use of specified signatures or signature fragments is not supported. Rubin et al. describe how mutants can be generated for a given attack [5]. Mutants exploit the same vulnerabilities as the basic attack but without performing the same security relevant actions. If a signature for an attack mutant is supposed to be engineered, the signature of the basic attack can be re-used, if available. The approach of Rubin et al. could be re-used for this purpose by deriving an abstracted attack. The required transformations though (except simple transformations like IP fragmentation) strongly depend on the specific attack. A universally valid procedure for all kinds of attacks is not implementable with this approach. Rubin et al. further describe the refinement of signatures based on formal languages [6]. This

approach may help the signature engineer to remove triggers of false positives caused by imprecise signatures. The procedure, however, assumes an almost error-free reference signature. Larson et al. present a tool for extracting the significant events of an attack from the audit trail [7]. It executes the attack and records the respective audit data. Then the differences between this audit data and attack free audit data are determined. The problem of deriving a signature from the differences, however, remains open.

Our approach is based on the assumption that the signatures of related attacks look similarly. After selecting the relevant signatures with regard to the given new attack, the signature engineer can look for similarities and adapt the selected signatures. The identification of relevant signatures can be accomplished by an iterative abstraction of existing signatures until traces of the given attack are covered. Abstraction means a generalization of signatures. Whether an abstracted signature detects the traces of the new attack can be easily decided by matching the signature to the traces. Abstractions are accomplished by iteratively applying transformations to the given signatures. The abstraction procedure results in an abstraction tree. In order to evaluate the abstractions each transformation is weighted by a metric which defines a similarity measure related to the original signature. Re-usable signature fragments should be derived from the least abstracted signatures. The signature engineer should try to understand the attack rationale by considering the original signature. In the following we introduce the transformation process, the resulting abstraction trees, and the applied similarity measures of our approach in detail.

2.1 Signature Transformations

A signature describes the set of identifiable manifestations of an attack. The latter is characterized by the events occurring during the attack execution. These events leave traces in the audit trail. When new signatures are engineered by re-use, signatures need to be identified that detect similar attack manifestations. Although the signatures may be similar, each signature is specialized to detect a specific attack. Signature transformations strongly depend on the signature model underlying the signature specification language in use. To identify re-usable signatures it is necessary to abstract from the attack-specific elements. This is done by iteratively removing semantic features with regard to the applied signature specification language. The semantic features (cf. [9]) determine the set of attack manifestations that a signature detects. New signature abstractions can be obtained by removing constraining features from a signature or by broadening their applicability. Transformations enlarge the set of attack manifestations M_{AS} that can be detected by an

M_{AS} that can be detected by an abstracted signature AS . A transformation T abstracts a signature S to AS only if $M_S \subseteq M_{AS}$ holds for the related sets of manifestations M_S and M_{AS} .

The concrete transformations strongly depend on the signature modeling framework. Hence, they need to be determined based on the signature specification language applied. In order to perform appropriate transformations the following three rules should be applied.

- (1) No two transformations should be defined which produce the same abstraction. This rule prevents redundant transformations.
- (2) Transformations should always be selected such that they semantically abstract the signature only slightly, i.e. the difference of the manifestation sets M_{AS}/M_S of the original signature S and the abstracted AS should be small. Such transformations lead to fine-grained abstractions of signatures.
- (3) A transformation which generates an abstraction AS from a signature S should preferably fulfill the condition that the intersection of the associated manifestation sets M_S and M_{AS} is not empty.

2.2 Signature Abstraction Tree

The iterative application of transformations to a given signature S from the set K of existing signatures produces all abstracted signatures for S . There exists a relation between the derived signatures concerning their abstraction. Two signatures A and B are related, if B is abstracted from signature A . The relations between a basic signature S and all of its abstractions can be represented by a signature abstraction tree, where S represents the root node. Direct children of S are those abstractions that can be generated by applying a single transformation to S . The signature abstraction tree and the related abstracted signatures can be generated successively. The tree structure determines the abstraction degree of S .

An abstracted signature associated to a node of the abstraction tree matches all attack manifestations that are matched by the signature associated to the parent node, i.e. an abstraction tree defines a *contained-in* relation over all sets of attack manifestations. Each abstracted signature accepts at least the manifestations that are accepted by S .

Fig. 1 depicts an example abstraction tree of S . Abstraction AS_1 is, for instance, directly generated from S by transformation 1, whilst signature AS_7 is derived from AS_1 using transformation 3. The respective manifestation sets M_S, M_{AS_1}, M_{AS_7} must meet the condition $M_S \subseteq M_{AS_1} \subseteq M_{AS_7}$.

The signature abstraction tree may contain identical nodes. Two nodes are identical if the associated signatures are identical. The sub-trees of these nodes are

equal. Therefore one of the identical nodes and its subtree can be removed.

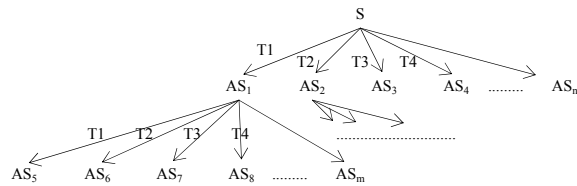


Fig. 1: Abstraction tree and manifestation sets

After deriving the abstraction tree, the abstracted signatures need to be tested in order to check to what extent they identify the traces T of the given attack. These tests can be supported by the use of an IDS. The test applies a breadth-first search. It stops, when an abstracted signature is found that detects the traces T and if there is no other untested signature which has a closer similarity to the root signature. As a result we get the abstracted signature(s) of lowest abstraction degree that match(es) the given traces.

2.3 Similarity Measures of Signatures

In order to estimate the similarities of different abstracted signatures the edges of the abstraction tree are weighted using a metric δ . This metric maps the type of the transformation onto real numbers. If a signature AS is generated from signature S by transformation X then the related edge between nodes S and AS is weighted with $\delta(X)$. The metric expresses the degree of the semantic abstraction of each transformation. After weighting the edges of the tree the similarities of different abstracted signatures related to the basic signature S can be assessed. This is done by cumulating the edge weights on the path from the root node to the abstracted signature, i.e. the abstraction degree of a signature is determined by the sum of the edge weights.

2.4 Signature Selection

Now we describe how signatures are selected from a set K of known signatures for a new attack. First, the traces T of the given attack, which are received by executing the exploit on a system, are logged. Next, the abstraction degree of each signature S in K is determined. It measures the abstractions which need to be applied to S to match the traces T . The procedure comprises five steps: (1) Successive application of transformations to S in order to derive all possible signature abstractions. (2) Generation of the abstraction tree. (3) Weighting the edges in the tree using a metric δ . (4) Applying the abstracted signatures of S to the traces T by using an IDS and identifying all signatures matching T . (5) Selecting the abstracted signature with the lowest abstraction degree. This abstraction degree is assigned to the signature S . After accomplishing this procedure for all signa-

tures in K , the abstraction degree of each signature is known. The signatures in K with the lowest abstraction degrees are suggested to the signature engineer for re-use. The signature engineer may use these signatures as well as the original signatures to understand the nature of the attack and to determine the differences to the current variant of the attack. The selection steps may be optimized as indicated in Section 2.2 and executed in parallel.

3 Application to Snort

Here we consider the application of our approach to the *Snort* signature specification language. *Snort* is a network-based IDS that is deployed by over 100,000 active users world-wide. It is the most widely used IDS and provides an extensive signature knowledge base. We start with a brief overview of the signature specification language of *Snort* and proceed with defining suitable transformations.

3.1 Modeling Signatures for Snort

Snort allows modeling single-step signatures by rules (see [1] for a detailed manual). In *Snort* parlance, signatures are also denoted as rules. *Snort* rules comprise a header and option conditions. The *rule header* specifies the basic parameters of the signature such as the inspected protocol (IP, UDP, TCP, or ICMP), the observed communication directions (uni- or bi-directional) as well as source and destination IP addresses and port numbers. In addition, actions may be optionally specified that are executed when the rule matches, e.g. generating alarms (*msg*) or dropping datagrams (*drop*). The *rule option* part specifies the identifier of the signature and the constraints for matching payload and header fields of the datagrams. *Payload constraints* may be used to match values or patterns in datagrams or transport streams. *Header constraints* are used to match datagram header fields, where the available fields depend on the protocol selected in the rule header. For example, sequence number fields can only be matched for TCP segments.

If *Snort* identifies a datagram or transport stream that matches all constraints of a rule header and of the payload and header constraints, the action defined in the rule header will be executed. Since actions are not related to rule abstraction, we refrain from going into the details here. An example rule is given in Fig. 2, pointing out the rule header and the header and payload constraints in the option part of the rule.

```
//rule header
alert tcp any any -> 141.43.3.0/24 445 (
  //header options
  tos: 1;
  flow: to_server, established;
```

```
//payload options
content: "|FF|SMB%"; depth: 5; offset: 4;
content: "&|00|"; within: 2; distance: 56;
content: "|05|"; within: 1; distance: 2;
content: "|0B|"; within: 1; distance: 1;
byte_test: 1,&,1,0,relative;
content: "|00|"; within: 1; distance: 21;
//actions
msg: " Netbios access";
//rule ID
sid: 2191;)
```

Fig. 2: Example of a Snort signature

3.2 Transformations for Snort Signatures

For the construction of the signature abstraction tree for *Snort* rules we use nine primary transformations. These transformations can be classified as follows: abstraction of the *rule header*, abstraction of the rule option part, i.e. the datagram *header* and *payload constraints*. In the following we describe the transformations for transformable rule elements in detail, also giving the possible pre-conditions. These pre-conditions have a two-fold function. First, they ensure that the given transformation produces a syntactically correct rule. Second, they ensure that the abstracted rules AS accept a super-set M_{AS} of the manifestations M_S of the original rule S , i.e. $M_S \subseteq M_{AS}$ (see Sect. 2.1).

Further we specify hints for each of the transformations that will be offered to the signature engineer together with the proposed rules. If a rule is proposed to the engineer due to the fact that after applying transformations t to this rule the resulting abstracted signature matches the given attack trace, then the hints specified for all transformations t are given to the engineer. These hints identify the elements of a rule that were abstracted and that need adjustment or extension. Hints do not offer advice towards entirely new elements that need to be added to the rule in order to match the new attack; they are rather intended to support the engineer in refining re-used rule fragments.

The following options are frequently used in the *Snort-VRT Certified Rules* (dating back to May 15, 2007) rule base (in % of the total number of rules): *content* (98%), *flow* (90%), *pcre* (66%), *byte_test* (50%), *byte_jump* (44%), *uricontent* (26%), and *isdataat* (2.5%). Each of the remaining options is used in less than 2% of the rules. Due to space limitations we present the transformations for a selection of these frequently used *Snort* options.

3.2.1 Transformations of the Rule Header. The *Snort* rule header can be used to restrict the network traffic being matched against the rule option part. This is done by specifying concrete source and/or destination IP addresses, network masks and/or port numbers. The option part of the rule will then only be matched against datagrams that meet the constraints of the rule header.

Transformation: Concrete IP addresses, network masks, and/or port numbers are replaced by the keyword *any* in the rule header. When applying this transformation to the rule given in Fig. 2, the constraint for port number 445 is removed, resulting in the rule depicted in Fig. 3. Alternatively, it is feasible to take a more fine-grained approach by dropping only parts of an IP address in a subnet-wise fashion, e.g. 141.43.3.0/24 → 141.43.0.0/16.

```
alert tcp any any -> 141.43.3.0/24 any (
  tos: 1; flow: to_server, established;
  content: "|FF|SMB%"; depth: 5; offset: 4;
  ...
  msg: "Netbios access";
  sid: 2191;)
```

Fig. 3: Transformation of the rule header

Note that if there do not exist any constraints in the rule option part then such a rule will just activate when any traffic is observed between the specified addresses and ports. Such rules in general are not part of generic rule bases as distributed to the community. They are rather of site-specific interest. Since they do not specify a specific kind of attack, they would be excluded from the process of rule abstraction.

Transformation hints: The signature engineer may restrict the rule to specific ports or IP addresses in order to reduce false alarms and improve the run-time efficiency of *Snort*.

3.2.2 Transformations of the IP Options. In the option part of the *Snort* rules we define constraints for matching the datagram header fields of the network traffic selected by the rule header. Exactly speaking, constraints can be defined for the fields *TTL*, *TOS*, *ID*, *Fragbits*, *Fragoffset*, *datagram size*, *protocol*, and *IP options*. If a given rule specifies constraints for one or more of these fields, all traffic that does not meet the constraints will not be considered any further by the rule. As an example, requiring the *Fragbits* option will focus the rule to fragmented datagrams only. Removing the option will widen the focus of the rule to also consider non-fragmented datagrams.

Transformation: Simple constraints with regard to datagram header fields such as *TTL*, *TOS*, *ID*, *Fragbits*, *Fragoffset*, *datagram size*, and *transport protocol*, will be removed from the rule one by one. Alternatively, it is feasible to take a more fine-grained approach by reducing the *TTL* and *datagram size* in a step-wise fashion. Fig. 4 exemplarily depicts the abstracted rule after removing the *TOS* option from the original rule in Fig. 2. In contrast to the original rule, the abstracted rule will also consider datagrams that are not tagged as urgent by the sender.

Transformation hints: If the transformation removed *Fragbits* or *Fragoffset* options, the signature engineer might look into fragmentation issues that the given at-

tack may exploit. The other options are usually merely used to corroborate other constraints by looking for known anomalies in the context of the given attack, e.g. *TTL > 64*.

```
alert tcp any any -> 141.43.3.0/24 any (
  flow: to_server, established;
  content: "|FF|SMB%"; depth: 5; offset: 4;
  ...
  msg: "Netbios access";
  sid: 2191;)
```

Fig. 4: Transformation of an IP option

3.2.3 Transformations of the TCP Options. Similar to *Snort* IP options for datagram IP header fields we can use TCP options for TCP header fields in TCP segments. *Snort* offers the following TCP options. The option *flags* is used for testing the TCP header control flags *FIN*, *SYN*, *RST*, *PUSH*, *ACK*, and *URG*. The *flow* option focuses the rule on open, established or stateless TCP connections. Additionally, the rule can be constrained to only one direction of a TCP connection by using the *to_client* or the *to_server* option. The options *seq*, *ack*, or *window* restrict the rule to segments with a specific sequence number, acknowledgement number, or window size, respectively. Like IP options, TCP options may also restrict the rule to traffic that meets the respective constraints. Hence, if such an option is removed the rule focus is widened to traffic that does not meet the respective constraint.

Transformation: TCP options are removed from the rule, one by one. Alternatively, it is feasible to take a more fine-grained approach by adapting the sequence numbers or the window size in a step-wise fashion. Fig. 5 depicts the abstracted rule after removing the *flow* option *to_server* from the original rule in Fig. 2. In contrast to the original rule, the abstracted rule will also consider datagrams that are sent by a machine that acts as a server in the context of a TCP connection.

```
alert tcp any any -> 141.43.3.0/24 any (
  flow: established;
  content: "|FF|SMB%"; depth: 5; offset: 4;
  ...
  msg: "Netbios access";
  sid: 2191;)
```

Fig. 5: Transformation of a TCP option

Transformation hints: If constraints on the communication direction have been removed, the signature engineer can investigate the server and the client for being vulnerable to the given attack, and then restrict the rule accordingly to look only for traffic directed towards the vulnerable machine, thereby improving the run-time efficiency of *Snort*. If the transformation removed constraints on sequence numbers or window sizes, the signature engineer can look into the TCP segment re-assembly issues that the given attack may exploit.

3.2.4 Transformations of the Payload Options. Payload options in the rule option part are used to match

patterns in the payload of datagrams. This allows for matching application layer specific byte string manifestations of attacks. Depending on the rule header the rule will match the specified patterns to the payload of single IP datagrams or to re-assembled TCP streams. We present the payload options and suitable transformations in the following.

Content option: This option matches a specified byte string in the payload. It is frequently used to detect protocol messages or control commands of application layer protocols, which exploit a vulnerability or control bots. As an example, the *content* option may be used to detect buffer overflow attacks.

Transformation: Content options may be abstracted by three transformations: (1) The *content* keyword is complemented with the *nocase* option, such that the rule also matches the byte string with varying case of characters. (2) The byte string is split into sub-strings that are delimited by freely specified separation symbols introducing a separate *content* option for each of the sub-strings. The abstracted rule thus detects the sub-strings independently of the separation symbols and with arbitrary characters in between. This is handy for detecting variants of an exploit, where arbitrary functionally neutral symbols have been introduced in order to thwart overly specific rules. (3) The *content* option is discarded.

Pre-conditions: In rules containing several payload options, the options may refer to each other. A *content* option must not be removed if it is referred to by other payload options. Referral to other payload options is expressed using the relative options *distance* and *within*. The respective informal semantics and suitable transformations are described in the following.

Distance x specifies the initial search position for matching a subsequent *content* option. Suppose that for two given *content* options for the second one there is defined a distance of x , then it will only be tried to be matched after the first x bytes following the first *content* option. **Within x** refines *content* options with a bounded search depth. Suppose that for two given *content* options for the second one there is defined a within of x , then it will only be tried to be matched within the first x bytes following the first *content* option. It is also possible to restrict the matching of a content option by referring the beginning of a packet. Therefore the relative options *offset* and *depth* can be used in the same manner like *distance* and *within*, respectively, except they are relative to the beginning of the packet.

Rules containing refined *content* options are abstracted by removing the refining options. A more fine-grained abstraction would step-wise decrease the *distance* and *offset* values and increase the *within* and *depth* values, respectively.

Transformation hints: If the referring options have been removed the signature engineer should check whether the remaining payload options still search for a similar phenomenon. In that case, the signature engineer should look for further constraints restricting the search space and search depth using the *distance*, *within*, *depth* or *offset* options thus reducing false alarms and improving the run-time efficiency of *Snort*. If a *content* option has been split, the original search string can be re-used when replacing certain blank characters by other blank characters. If a *content* option has been removed entirely, the signature engineer might check, whether the manifestations contain substrings of the original content option that could be used, instead.

Uricontent option: This option matches a normalized URI string in the payload, where normalized means that also semantically equivalent variants are matched, if they syntactically differ only in semantically equivalent symbols or codes. Additionally, URI directory paths are rewritten into a reduced form before matching, e.g. `<dir1>../../../../<dir2>` becomes `../../../../<dir2>`.

Transformation: The option is discarded in order to abstract the rule. More fine-grained transformation is possible using selective removal of sub-directories.

Transformation hints: If the *uricontent* option has been modified the signature engineer should check, whether the new string conforms to the actual configuration of the service, e.g. the web content and module directories of Apache.

Isdataat option: This option tests whether the payload contains data at the specified position, which is basically used to check packet lengths and for detecting buffer overflow attacks.

Transformation: The option is discarded in order to abstract the rule. Stepwise decrementing the position parameter allows for more fine-grained transformation.

Transformation hints: If the *isdataat* option has been modified the signature engineer can determine the actual position in the given manifestations.

Pcre option: Similarly to the *content* option the *pcre* option can be used to match byte strings or patterns in the payload. The *pcre* option, however, allows a more expressive specification of the pattern to be matched using Perl compatible regular expressions (see Fig. 6a for the syntax of the option). At first, we introduce the most important switches i , A , E , and R of the *pcre* option before we describe suitable transformations.

Switch i allows for case insensitive matching. Switch A limits matches to the starting position of the payload. Switch E modifies the behavior of the '\$' symbol, such that it matches the end of the payload instead of any end-of-line symbol. Switch R works just as the *relative* option. The search for the pattern then starts relative to the last match identified by the *content* or *pcre* option. Further switches are s , m , x , G , U , and B .

We omit the description of these infrequently used switches for the sake of brevity.

Transformation: A given *pcr* option may be abstracted by modifying or entirely removing the option. The option may be modified in two ways: (a) the option switches may be modified in order to widen the search focus. We give three examples: setting the *i* switch will ignore the case during matching, effectively accepting a wider range of byte strings. Removing the *A* or the *E* switch will remove the restriction to matching the starting position or end of any line, respectively, allowing for a potentially larger number of matches. (b) The regular expression is suitably modified. We provide several examples. Line break symbols, such as `\r\n`, `\n\r`, `\r` and `\n`, may be replaced with `\R`, such that the search is not restricted to a fixed line break symbol, but it accepts diverse variants of line break symbols. Removing trailing `$` symbols allow matching the expression over the whole payload instead of merely at end of lines or the payload. The respective rationale applies to the `^` symbol as well, which restricts matches to the starting position of lines or the payload. The replacement of `\s` by `\s+` allows an arbitrary number of blank symbols between sub-strings, instead of merely one blank symbol. As for the *content* option we can also split the regular expression into sub-expressions that are delimited by arbitrary separation symbols and accordingly use as many *pcr* options. Each option then matches the respective sub-strings in the payload, allowing for additional strings between the sub-strings and for an arbitrary order of the sub-strings in the payload.

Pre-conditions: In rules containing several payload options, other options may refer to a *pcr* option (see relative matching for *content* options). A *pcr* option must not be removed if it is referred to by payload options.

Transformation hints: If the referring options have been removed the signature engineer should check, whether the remaining payload options still search for a similar phenomenon. In that case, the signature engineer can look for further constraints reducing the search space and search depth using the *distance*, *within*, *depth* or *offset* options, thereby reducing false alarms and improving the run-time efficiency of *Snort*. If a *pcr* option has been split the original search string can be used when replacing certain blank characters with other blank characters. If a *pcr* option has been removed entirely the signature engineer can check, whether the manifestations contain substrings of the original content option that can be used, instead. If `\s` has been replaced with `\s+` the signature engineer can determine the actual number of blank characters in the manifestations. If the `\R` switch has been inserted the signature engineer might determine the actual line break coding.

Byte_test option: This option extracts a given number `<num>` of bytes from a specified position `<offset>` of the payload, interprets them as a number given in a specified format and compares that number against a specified number `<value>` using a specified comparison operator (see Fig. 6b for the syntax of the *byte_test* option). The switch *relative* interprets `<offset>` relatively to the last match.

Transformation: We assume that testing for a given `<value>` is highly specific for the given attack, such that a general fine-grained abstraction strategy cannot be provided. The rule can still be abstracted by removing the entire option.

Pre-conditions: A *byte_test* option must not be removed if it is referred to by *content*, *pcr*, *byte_test* or *byte_jump* options.

Transformation hints: Since `<value>` is highly specific for the given attack, general advice concerning the option value cannot be provided, if the *byte_test* option has been removed.

Byte_jump option: This option extracts a given number `<num>` of bytes from a specified position `<offset>` of the payload, and after multiplying the composite number with `<factor>` interprets it as a position in the payload prior to using a *byte_test*, *pcr*, or *content* option (see Fig. 6c for the syntax of the *byte_jump* option). The option *relative* works as described for *byte_test*.

a)	<code>pcr: "<regex> [/{i s m x A E G R U B}];"</code>
b)	<code>byte_test:"<num>",< > = ^ &","<value>","<offset>":["relative"]"</code>
c)	<code>byte_jump:"<num>","<offset>","multiplier"<factor>":["relative"];</code>

Fig. 6 a,b,c: Syntax of the payload options

Transformation: We assume that a given tuple `<num, offset factor>` is highly specific for the given attack, such that a general fine-grained abstraction strategy cannot be provided. The rule can still be abstracted by removing the entire option.

Pre-conditions: A *byte_jump* option must not be removed, if it is referred to by a *byte_test* option.

Transformation hints: Since the *byte_jump* option values are highly specific for the given attack, general advice concerning these values cannot be provided, if the *byte_jump* option has been removed.

4 Evaluation Method

A rigorous evaluation of the suitability of the proposed approach requires cross-testing of several *Snort* rule engineering cycles of several experienced signature engineers with and without the proposed process, and comparing relevant process parameters such as the time for engineering new rules and the usefulness of the ru-

les. Note that assessing the usefulness of a given rule implies the evaluation of false negatives and positives generated by the rule. Such a personnel intensive evaluation has been out of the scope of this project. In order to get an impression of the suitability of the approach, we implemented the process for *Snort* rules and applied it to selected examples. The prototype automates the process beginning with abstracting signatures, testing signatures using *Snort* on the given attack trace, and selecting signatures for consideration of re-use or adaptation. Due to the promising results we are confident that further investigation is justified.

For our experiments we used the 7544 *Snort-VRT Certified Rules* dating back to May 15, 2007. The evaluation of the approach is divided into four phases. (1) An arbitrary rule is picked out randomly and removed from the rule base. (2) The attack manifestations are manually inferred and synthesized from the rule. (3) Our prototype generates all abstracted signatures from the rule base and tests each abstracted signature against the synthetic manifestations. (4) The tool selects the abstracted signatures that match the manifestations and ranks them by their degree of abstraction.

For a number of transformations described in Section 3.2, more fine-grained alternatives exist. Whether these refinements allow better re-usable signatures or just result in the creation of additional abstractions primarily depends on the used signature base. Further alternative transformations are feasible. The transformations defined above provide a starting point that is iteratively improved with each deployment cycle of the process. In the following we present an approach for tailoring the metric δ and the granularity of transformations to the given signature base.

Determining Weights and Granularity. For any new attack for which we deploy the re-usage approach, we save the x proposed signatures that were ranked best and the transformations applied in order to match the new attack. The quality of each proposed signature is manually marked with values of the interval $[0,1]$ (real numbers, 0 best to 1 worst suitable for re-use). Thus, each deployment cycle of the re-usage approach results in x tuples $\langle S, T, Q \rangle$, where S represents the proposed signature, T specifies the set of applied transformations and Q describes the quality of S with respect to its re-use for the given attack.

Based on these experiences (set of tuples E) we optimize δ in order to associate well re-usable signatures with a low abstraction degree. To this end δ is optimized for each transformation such that the respective abstraction degree (cumulated $\delta(t)$ for all t in T , $\sum_{t \in T} \delta(t)$) of as much as possible tuples in E correlates with quality Q of the proposed signature. This is a typi-

cal optimization problem, where the objective function given by the distance between abstraction degree $\sum_{t \in T} \delta(t)$ and quality Q is minimized for all tuples in E .

The parameters to be optimized are given by the metric δ , while the sum of the metric values of all types of transformations is constant.

We proposed a number of fine-grained transformations in Sec. 3.2. In order to determine a suitable granularity and the transformations resulting in the best proposed signatures we again use the tuples E of (ranked signature proposals of) previous abstraction processes. Numerous occurrences of a transformation type in set E (e.g. frequent relaxation of constraints regarding IP addresses or the search depth of content options) for high quality signatures identifies transformations that are too fine-grained. Such transformations should be coarsened to avoid generating unnecessary signature abstractions. Situations in that the metric value of a transformation is difficult to optimize for tuples in E (e.g. in one case the value should be high, in another case it should be low) indicate that this transformation is too coarse-grained. To provide more freedom for optimizing the metric and to allow better adjustment of δ such transformations need to be refined. Modifications of the granularity of transformations require updating of sets T of tuples in E in order to have a valid base for evaluation. In case of coarsened transitions an update simply replaces the substituted fine-grained transformations with respective coarse-grained transformations. In contrast, refinements of transformations call for redefining the sets T in tuples of E . However, this can be done automated for the abstraction and test process.

To go smoothly the proposed approach requires an initial set of marked signatures. Right now we are in the process of evaluating signatures and creating such a set, but do not yet have a sufficiently large set of marked signatures to allow an automated deployment of the approach. Therefore we use a constant value of 1 as an initial metric δ in the examples described in the following section.

5 Examples

In the following we demonstrate the approach and the evaluation method using two selected examples. The first example shows how a new signature is nearly completely derived from the signature base. The second example demonstrates how re-using signature fragments can significantly reduce the engineering effort.

Example 1: First we use a rule from the *Snort* rule base which detects a buffer overflow attack on the FTP service of the Oracle XML DBMS (see Fig. 7a). We chose this rule as a typical example for a buffer over-

flow attack. Manifestations of this attack were captured while the attack was executed. The rule focuses on TCP connections to the destination port number 2100 and looks for a string in the payload starting with “USER” and followed by one or more blank characters and a number of arbitrary other bytes, where the complete string is longer than 100 bytes. The over-sized user ID string provokes a buffer overflow during login to the Oracle FTP service.

Our tool identifies a series of abstracted rules for this rule. We investigate here the four rules from the *Snort* rule base with the lowest abstraction degree that after abstraction detect the above attack (see Fig. 7 b,c,d,e). The rule options that were transformed are grey.

The first rule (see Fig. 7b) proposed by our tool is too specific concerning the destination port number and the search string. The rule matches the given manifestations after: (i) removing the port number, (ii) replacing ‘\s’ with ‘\s+’ in the pcre option. Hence, the abstraction degree of the rule is two. In our example this rule exhibits the lowest abstraction degree. The second rule (see Fig. 7c) needs three transformations: (i) removing the port number, (ii) splitting the search string, and (iii) subsequently discarding the second search sub-string. The third rule (see Fig. 7d) proposed by our tool has an abstraction degree of four. In addition to (i) removing the port number and (ii) splitting the search string: (iii) removing the second search sub-string from the pcre option and (iv) the search for the string “y049575046” of the content option needed to be removed. For the fourth rule (see Fig. 7e) four transformations have been applied: (i) removing the port number, (ii) removing the search string for the content option, (iii) splitting the search string for pcre option, and (iv) removing the second search sub-string.

From the abstracted rules described above the first rule is the closest one to the target rule that we removed from the rules base. Merely replacing ‘\s’ with ‘\s+’, and removing ‘(?!\\n)’ are necessary to obtain the target rule. The characterizing element of the abstracted rule, namely detecting oversized user ID strings, can be directly re-used for the target rule.

For re-using this rule the following hints are provided. (1) In order to avoid false positives and to improve the run-time efficiency of *Snort* the port number should be limited, if possible. Following this hint the engineer restricts the port number to 2100. (2) Due to the replacement of \s by \s+ the engineer tries to determine the actual number of blank characters. The exact number can be determined by reviewing the source code of the FTP service of Oracle’s XML DBMS. Following these hints the engineer can easily re-create a suitable signature. Adopting the *Isdataat* option of the proposed rule is not necessary, but it represents a possi-

ble optimization compared to the original rule for the given attack (see Fig. 7a).

a)	<pre> alert tcp \$EXTERNAL any -> \$HOME 2100 (flow: to_server, established; content: " user"; nocase; pcre: "/^USER\s+[\n]{100,}/smi"; sid: 3631;) </pre>
b)	<pre> alert tcp any any -> any 21 (flow : established, from_client; content: "USER"; nocase; isdataat: 100, relative; pcre:"/^USER(?:\n)\s+[\n]{100}/smi"; sid: 1734;) </pre>
c)	<pre> alert tcp any any -> any 21 (flow:established, from_client; content:"USER"; nocase; pcre:"/^USER\s+[\n]*%[\n]*%/smi"; sid:2178;) </pre>
d)	<pre> alert tcp any any -> any 3535(flow:established, from_client; content:"USER"; nocase; content:"y049575046"; nocase; pcre:"/^USER\s+y049575046/smi"; sid:2334;) </pre>
e)	<pre> alert tcp any any -> any 21 (flow : established, from_client; content: "USER" ; nocase; content: "w0rm"; nocase; distance: 1; pcre:"/^USER\s+w0rm/smi"; sid: 144;) </pre>

Fig. 7 a,b,c,d,e: Suggested original rules for re-use

Example 2: In this example we chose a *Snort* rule for detecting injection and execution of arbitrary code using buffer overruns in RPCSS service (see Fig. 8). Traces of this attack were captured, while the attack was executed.

In this case, all rules of the signature basis need to be strongly abstracted to match the attack trace. Therefore, we were not able to derive a complete signature by simply re-using the proposed signatures and implementing the given hints. However, all of the proposed best-ranked rules contain the grey lines shown in Fig. 8. An analysis of these lines shows that this rule fragment is responsible for detecting (a) Netbios packets containing (b) a WIN2K/XP special header and (c) a remote administration protocol header that contains (d) a SMB command. Further this rule fragment allows analyzing the particular SMB command by appending distinct payload options (jump into the payload of the SMB command). The specific characteristics of the attack still need to be modeled, but the re-use of this approved signature fragment for protocol testing and header traversal clearly reduces the development time. Fig. 8 shows that the original *Snort* rule for this attack uses this rule fragment too (grey area).

```

alert tcp $EXTERNAL_NET any -> $HOME_NET 139
(msg:" NETBIOS SMB ISystemActivator
CoGetInstanceFromFile WriteAndX unicode little
endian object call attempt";

```

```

flow:established,to_server; flowbits:isset,
dce.bind.ISystemActivator;
content:"|00|"; depth:1; content:"|FF|SMB/";
within:5; distance:3; byte_test: 1,
&,128,6,relative; pcre:"/^.{27}/sR";
byte_jump: 2, 23, relative, from beginning,
little; pcre:"/^.{4}/sR"; content:"|05|";
within:1; byte_test:1,&,16,3,relative;
content:"|00|"; within:1; distance:1;
byte_test:1,&,128,0,relative; content:"|01
00|"; within:2; distance:19; pcre:"/^.{16}
/sR"; content:"|01 10 08 00 CC CC CC CC|";
distance:0; content:"|5C 00 5C 00|";
distance:0; byte_test:4,>,256,-8, relative,
little; reference:cve,2003-0715;
reference:url,www.microsoft.com /technet/
security/ bulletin/ms03-039. msp; classtype:
protocol-command-decode; sid:3178; rev:4;)

```

Fig. 8: Rule for buffer overruns in the RPCSS service

The examples demonstrate that our approach, which was originally for multi-step signatures, can also be applied to single-step signatures. It works well for the attack description language of the most prominent single-step IDS. Currently, we do not have enough experience tuples E of deployment cycles yet to optimize the initial metric and the granularity of transformations as described in Sec. 4. However, based on the initial parameters we already achieved promising results for all deployment cycles of the approach realized so far.

Resource Requirements of the Tool. In order to estimate the tool’s run-time and memory requirements we capture some performance numbers. The following discussion considers the execution of the second example above. In this example 7543 (number of Snort-VRT Certified Rules - 1) *Snort* signatures were used as basis. Each signature was abstracted using transformations until one abstraction was generated that matches the given audit trace. In total about 641,453,983 abstractions were generated, of which only 7543 match the given audit trace (one abstraction of each basis signature). Thus on average about 80,000 (85039) were generated per signature. Generating and matching would require a total amount of time of about 122 hours. Actually only about 17 hours were needed because we parallelized the process using seven typical desktop machines (Intel Xeon CPU 2.66GHz). The memory consumption of the tool was 500 MB on average and 1.9 GB on peak. Many further optimizations are possible to minimize the runtime and memory requirements.

6 Final Remarks

Signature engineering can be supported by re-using signature design decisions and/or fragments of existing signatures. The re-use of already approved structures may not only reduce the effort of the signature engineering process, but it can also considerably shorten the costly test and correction phase. Moreover, the pro-

posed procedure allows the signature engineer to exploit experience encoded in existing signatures. In this paper we showed that our general approach is applicable for single-step signatures. We systematically analyzed the elements of a single-step signature specification language and identified suitable transformations for signature abstraction. The approach was implemented for the most prominent representative of single-step IDSs, namely *Snort*. For a given attack manifestation the tool computes and selects abstracted signatures that are suitable for further refinement and for understanding the given attack manifestation. We have also developed methods for selecting the set of most effective transformations as well as for evaluating the quality of the generated abstracted signatures. We have demonstrated the approach and exemplarily evaluated.

Directions for future work include the implementation of an automated method for selecting the most effective transformations for a given set of signatures in a given specification language. As another working direction we consider identifying design patterns for signature engineering from a given signature base.

7 References

- [1] Baker, A.; Beale J.; Caswell B.; Poore M.: *Snort 2.1 Intrusion Detection*. Syngress Publishing, 2004.
- [3] Cheung S.; Lindqvist U.; Fong M.: Modeling Multi-step Cyber Attacks for Scenario Recognition. In: Proc. of the 3rd DARPA Information Survivability Conf., IEEE Computer Society Press, 2003, pp. 284-292.
- [4] Gamma E., Helm R., Johnson E. R.: *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1997.
- [5] Rubin S., Jha S., Miller B.: Automatic Generation and Analysis of NIDS Attacks. In: Proc. of the 20th Annual Computer Security Applications Conf., IEEE Computer Society Press, 2004, pp. 28-38.
- [6] Rubin S.; Jha S.; Miller P. B.: Language-based generation and evaluation of NIDS signatures. In: Proc. of the IEEE Symposium on Security and Privacy, IEEE Computer Society Press, 2005, pp. 3-17.
- [7] Larson U., Lundin B. E., Jonsson E.: METAL - A Tool for Extracting Attack Manifestations. In: Proc. of the 2nd Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment, 2005, LNCS 3548, Springer, pp. 85-102.
- [8] Meier M.; Schmerl S.: Improving the Efficiency of Misuse Detection. In: Proc. of the 2nd Conf. on Detection of Intrusions, Malware, and Vulnerability Assessment, LNCS 3548, Springer, 2005, pp. 188-205.
- [9] Meier, M.: A Model for the Semantics of Attack Signatures in Misuse Detection Systems. In: Proc. of the 7th International Information Security Conference (ISC 2004), LNCS 3225, Springer, 2004, pp. 158 – 169.