Firewall Analysis by Symbolic Simulation: Advanced Optimizations

Arno Wagner
Consecom AG
Zurich, Switzerland
arno@wagner.name

***Abstract*—There are two primary tasks when doing a Layer 4 firewall security analysis. First, unifying a chain of firewalls on a given network path into a single one to efficiently determine what it allows to pass and what it drops, and second, comparing a firewall with a security policy. Both tasks are work-intensive and error-prone if performed manually and become infeasible in the presence of large firewall rule sets. To automate the process of unifying a chain of firewalls, we have created the Consecom Network Analyzer that uses symbolic simulation with an interval representation to generate a unified equivalent firewall in a normalized, simple and flat form. The unification process is also suitable to implement comparison with a policy, by representing the policy in a special way in the form of a firewall rule set. We show the suitability of this approach for firewalls with large configurations by giving benchmarks based on deployed rule sets. In addition, we demonstrate the effects of different optimization techniques on runtime and memory footprint, including the use of an advanced optimization technique that builds on ideas from geometrical search to reduce unnecessary rule applications by means of interval search trees. The Consecom Network Analyzer has been used successfully for a number of industrial security reviews.**

Keywords-Network Security; Firewall Analysis; Symbolic Simulation; Interval Search Trees.

I. Introduction

This work describes the *Consecom Network Analyzer* (CNA), which is the result of a collaboration between academia and industry. It is an invited extension of results previously published in [1]. The main improvement is the use of *Interval Search Trees* as additional optimization technique, as described in Section VII.

The CNA is a tool-set that greatly reduces the effort, and thereby cost, for practical firewall security analysis in the presence of firewalls with large rule sets. A firewall security analysis is one type of network security review. It is often done on network Layer 4, for example for TCP and UDP traffic. Figure 1 shows the basic scenario. The typical steps to be done include:

1) Normalize firewall configurations
2) Identify critical network paths
3) Identify firewalls along each critical path
4) Determine network reachability on each critical path
5) Compare reachability and security requirements
6) Identify non-compliant firewall rules

The primary motivation for creating the CNA lies in steps 4, 5 and 6. In step 4, the CNA calculates the reachability in a unified simple format. Each element of the combined reachability is annotated with the firewall rules that give raise to it. If a formalized or easy to formalize security policy is available, it can be compared automatically to the actual network reachability using the CNA. As such a security policy is often not available in practice, step 5 may still need to be done manually or can be only partially automatized.

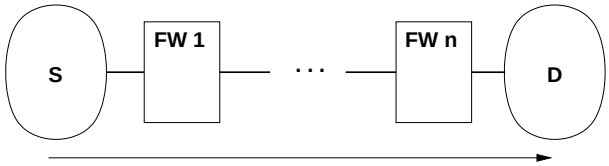


Fig. 1. Unidirectional reachability along a critical network path.

Figure 2 shows the typical data flow for a firewall analysis task. The Rule-Set Converter is not part of the core CNA system and has to be adapted for each different firewall description format. The CNA uses a normalized symbolic Layer 4 format internally that is based on intervals. As core contribution of this paper, we show this representation is suitable for calculating reachability even in the presence of large firewall configurations. To this end, we present benchmark calculations on deployed rule-sets. The CNA has been used successfully in several industrial firewall security reviews.

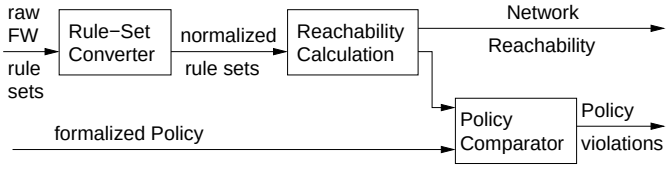


Fig. 2. Typical analysis data-flow with the CNA.

The rest of the paper is organized as follows: Section II introduces our network and firewall model, and the

symbolic representation used. Section III gives the operations used for single firewalls. Section IV explains how to calculate unidirectional reachability. A complexity analysis is sketched briefly in Section V. Section VI describes the implementation, while Section VI states benchmark results and the effects of different optimization techniques. Section VII explains how interval search trees can be used to speed up the CNA core loop and justifies their effectiveness with a separate set of benchmarks. Section VIII explains how to extend the approach to two-sided reachability and to automated comparison with a policy. The paper finishes with a discussion of related work in Section X and a conclusion in Section XI.

II. Approach

The reachability calculation process starts with a representation of the initial reachability (disregarding firewalls), which will often be unconstrained. This initial reachability is then successively reduced by applying firewall rules. The end-result is a flat, unified representation of the firewall-chain, restricted by the initial reachability.

A. Network Model

We are primarily interested in network reachability as restricted by firewalls. Given a source network S, sequence of firewalls FW_1, ... , FW_n and a destination network D (see also Figure 1), we say that D is *reachable* from S if there are network packets that can traverse FW_1, ..., FW_n without being dropped by any FW_i. Note that some attacks will need *two-sided reachability*. For example services used over TCP can usually only be attacked if response packets can traverse the firewall sequence in reverse order. See Section VIII-A for a discussion on how to check for two-sided reachability.

We restrict the packet information visible to firewalls to IP addresses and ports, which results in a Layer 4 model. Each protocol is treated separately, although it is possible to mix protocols, for example by doing a forward analysis with TCP and a backward analysis with ICMP in order to determine whether an ICMP response to a TCP packet would get through. This situation arises, for example, when determining whether a firewall configuration allows port scanning. Routing is out of scope for this work, as we do not see it as a security mechanism; see Section IV-A for a brief discussion.

B. Subspaces, Boxes and Intervals

Reachability is represented by subspaces of

$$M = \{\text{src IPs}\} \times \{\text{src ports}\} \times \{\text{dst IPs}\} \times \{\text{dst ports}\}$$

with the four fields representing the corresponding IP v4 layer 4 header address fields for TCP and UDP, and the port fields being misused to represent ICMP Type and Code for ICMP. Other layer 4 protocols that fit this scheme can also be represented.

We organize these subspaces into sets of axis-aligned hyperrectangles in M, also called *axis aligned boxes* [2], [3]. In this paper, boxes will always be axis-aligned, hence we will simply call them *boxes* for short.

Note that any non-empty subspace of M that has an interval for each of its 4 components trivially is a box. At the same time, *any* subspace of M can be represented as the union of a set of boxes. A subspace A of M can hence be represented by

$$\begin{aligned} A &\subseteq M \text{ and} \\ A &= \{b_1, \ldots, b_n\} \text{ with } b_i \in M \text{ and } b_i \text{ is a box.} \end{aligned}$$

The matching expressions of a firewall rule can be represented by a single box. Security policies can also be represented this way, by giving a set of boxes that specifies forbidden reachability. If the intersection between network reachability and a policy represented this way is non-empty, then the policy is violated. In the implementation, boxes can have attached information. In particular, trace information can be attached in order to document which firewall rules were applied to a box. Trace information is critical to determine why a specific box is in the final reachability or why it was dropped.

A box can be represented as a 4-tuple of intervals, which allows symbolic computations. As far as we know, Eronen and Zitting [4] were the first to use intervals in this context.

Box example:
$b = (10.0.0.0 - 10.0.0.255, 1024 - 65535, 10.1.1.1, 80)$

We use intervals with wrap-around, where IP and port number spaces are regarded as circles. This facilitates representing complements and reduces the number of elements in the complement of a box, see below. Figures 3 and 4 gives graphical examples of three boxes in two dimensions represented this way. Some textual examples for intervals with wrap-around are:

- Port interval $[81, 80)$ represents all ports except port 80, i.e., port 81-65535 and port 0-79. Without wrap-around this complemented interval would need to be represented as $[0, 80)$ and $[81, 65535)$
- IP interval $[127.0.0.256, 127.0.0.0)$ represents all IP addresses except $127.0.0.0 - 127.0.0.255$.

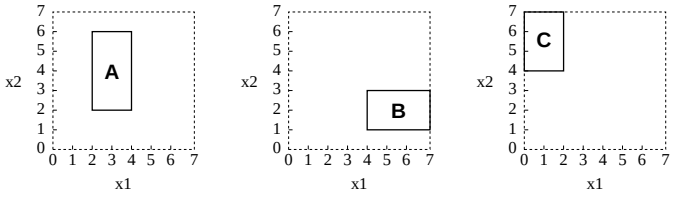


Fig. 3. Boxes in two dimensions.

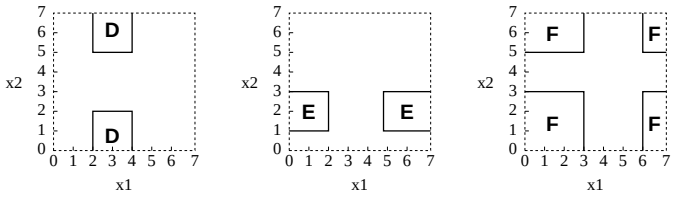


Fig. 4. Boxes with wrap-around in two dimensions.

C. Firewall Model

The CNA uses a simple firewall model, where each firewall consists of a linear sequence of rules r that each have a box describing their applicability and one of the target actions *accept* or *drop*, with a default *drop* at the end of sequence. This corresponds to the "simple" model used in [5].

D. Rule Application and Set Operations

In order to apply a firewall rule $r = (b, <\text{action}>)$ to a subspace $A = \{b_1, \ldots, b_n\} \subseteq M$, we intersect b with the different b_i in turn and apply the action to the result $A \cap \{b\} = \{b \cap b_1, \ldots, b \cap b_n\}$.

The usual set operations are defined on boxes and, by extension, on subspaces of M. Some deserve additional comments.

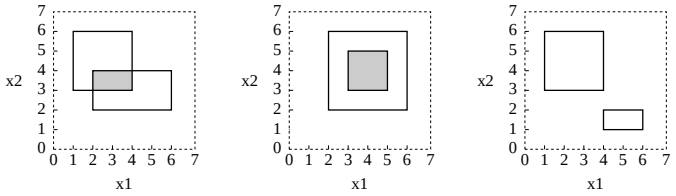


Fig. 5. Box intersections in two dimensions.

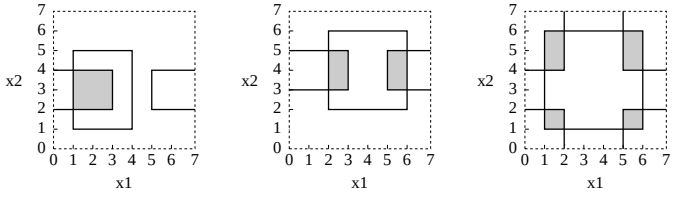


Fig. 6. Box intersection in two dimensions with wrap-around.

Intersection: Intersecting two boxes in d dimensions can have up to 2^d result boxes. Figures 5 and 6 illustrates this in two dimensions. For $b_1, b_2 \in M$, the intersection $b_1 \cap b_2$ may consist of up to 16 boxes as M has 4 dimensions.

Box complement: The complement of an interval is derived by adjusting the boundaries. The complement of a box is derived by complementing each interval in turn and setting all other intervals to full range. Hence, a 4-dimensional box has up to four boxes as its complement. Without wrap-around, the complement of a box could have up to 8 elements.

Subtraction: Calculating $a - b$ for boxes a and b is done by using the relation $a - b = a \cap \bar{b}$ from set calculus.

III. Restricting Reachability by a Single Firewall

The core operations used in determining reachability through a single firewall are `apply_firewall()` and `apply_rule()`, shown in Figure 7 in simplified form. The task of `apply_firewall()` is to take a given reachability description, stated as a set of boxes, called here a *Work Set* (WS) and, using the rules of the firewall, determine both an *Accept Set* (AS), which is the part of the WS that can pass the firewall, and a *Drop Set* (DS) that is the part of the WS that cannot pass the firewall. AS and DS are represented as sets of boxes. The function `apply_rule()` forms the basis of `apply_firewall()` and implements calculation of the intersection I between a given rule and WS. The intersection I is then added to the AS for an *accept* rule or to the DS for a *drop* rule.

```
apply_firewall(WS, FW):
  AS := ∅      /* Accept Set */
  DS := ∅      /* Drop Set   */
  for r ∈ in FW:     /* r: box of a rule */
    I := apply_rule(WS, r)
    WS := WS - I     /* reduce Work Set  */
    if r is accept:  AS := AS ∪ I
    if r is drop:    DS := DS ∪ I
  return(AS, DS)

apply_rule(WS, r):
  I := ∅
  for b ∈ WS:        /* b is a box */
    i := b ∩ r
    I := I ∪ i
  return(I)
```

Fig. 7. Pseudo-code for `apply_firewall()` and `apply_rule()` (simplified).

Building on these two operations, more complex operations can be constructed. Note that `apply_rule()` may attach trace information to boxes, for example to document rule application. If desired, the full history of each box can be recorded in the trace. This allows to determine the specific firewall rules that are responsible for a box being in the final reachability and represents information needed in any report about firewall configuration issues.

IV. Unidirectional Reachability Computation

Pseudo-code for the calculation of unidirection reachability through a *sequence* of firewalls is given in Figure 8.

We will typically choose the initial reachability as unrestricted. This is a sound practice, as network routing can usually not be regarded as a security feature and quite a few customers cannot specify source network `S` and destination network `D` with the required exactness. Starting with full, unconstrained reachability will ensure the final results only rely on the given firewall configurations. A more restricted initial reachability can still be used when appropriate. Ports are unconstrained in the initial reachability.

A. Comments on Routing

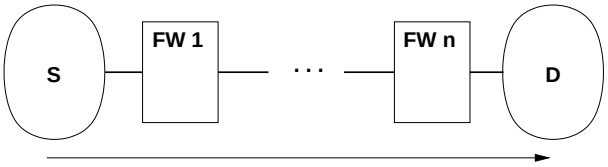
A frequent issue that crops up when doing a firewall security analysis in the field is that often routing is mixed with firewalling. This view gives flawed results. There are several reasons:

- The primary task of routing is to get packets to a specific destination, while the primary task of a firewall is to prevent packets reaching a specific destination. Routing configuration and firewall configuration hence have diametrically opposed primary tasks and this is reflected in procedures and mind-sets.
- Due to the different primary tasks, often the teams responsible for routing and for firewalls are different.
- While firewall configurations are handled securely and all updates are done with the security model in mind, routing configurations are typically changed with the network model in mind and handled in a less secure fashion. Routing is hence easier to compromise.
- Sometimes customers cannot even specify the IP ranges of `S` and `D` precisely, but have precise firewall information. This may sound surprising, but if routing delivers more to a physical target network than expected, this is not necessarily a problem. For firewalls, it is a critical error.
- Routing works on Layer 3, while firewalls work on Layer 4. Mixing the two complicates things and increases maintenance effort.
- Firewall configurations often do not include information about physical or virtual interfaces, but solely refer to layer 4 information. If routing were regarded as a security feature, interface information would be needed in addition and would be critical for security. This would also complicate firewall configuration and make network security critically dependent on the details of physical or virtual network cabling.

Overall, it is far more practical to separate routing and firewalls and to require that all restrictions on reachability must be implemented by firewalls placed into the critical network paths. This is especially true for customers with complex firewall configurations.

It should be noted that with this approach, the question arises whether a specific firewall actually is on the critical network paths it is supposed to be on. Answering this question requires a network topology analysis and is outside of the scope of this work.

It should also be noted that network scanning always takes routing into account and is restricted by it. This is a fundamental limitation of network scanning that is not present in firewall simulation approaches.




```
in:  S, D  /* Source, Destination networks */
     FW1, ..., FWn   /* firewalls          */
out: ASn             /* final reachability */
     DS1,...,DSn     /* Drop Sets          */

WS1 := S × <all> × D × <all>
(AS1, DS1) := apply_firewall(WS1, FW1)
WS2 := AS1
(AS2, DS2) := apply_firewall(WS2, FW2)
WS3 := AS2
...
(ASn, DSn) := apply_firewall(FWn − 1, WSn − 1)
```

Fig. 8. Pseudo-code for calculating unidirectional reachability with `apply_firewall()` for the scenario shown in Figure 1.

V. Algorithmic Complexity

We briefly sketch the complexity analysis idea. For a worst-case scenario, start with one box and a single firewall with n drop rules. Each drop rule can split (asymptotically) at most one element of the Work Set into a maximum of 2^d (with dimension $d = 4$) non-overlapping parts that are kept in the working set. Hence, each rule increases the size of the working set by a maximum of 16, giving an overall space complexity of the result of $16 * n \in O(n)$ for n firewall rules. As each successive rule application has to work on 16 more boxes, time complexity is $1 * 16 + 2 * 16 + \ldots + n * 16 = 16 * (1 + 2 + \ldots + n) = \frac{16}{2} n(n-1) \in O(n^2)$. A very similar argument applies to accept rules and mixed rule-sets.

In comparison, in [6], the authors need worst case effort $O(n^4)$ to build a Firewall Decision Diagram (FDD) for n firewall rules with the same firewall model as we use. It is reasonable to expect that this worst-case is extremely unlikely to happen in practice.

In [5], the authors claim a worst case complexity of $O(n)$ for processing a firewall with n rules in their "simple model". However, they wrongly assume constant effort for set operations on their accept (A) and drop (D) sets. While the BDDs used in [5] are often very efficient in practice, they do not have constant worst case effort for set

TABLE I
BENCHMARKS

No	Firewall or Firewall sequence	rule-set size raw	nor-malized	opt.	benchmark results Python baseline		input opt.		trace reduction		core-loop ported to C	
1	S	27	2'000	180	4:52min	12MB	3.9s	6MB	3.2s	6MB	0.07s	6MB
2	M	67	23'000	8300	1752 min	184MB	346min	84MB	148min	48MB	29s	18MB
3	L	170	27'000	3100	2392min	292MB	21:54min	26MB	12:45min	18MB	2.8s	10MB
4	S, M				64min	34MB	191s	14MB	156s	13MB	1.8s	13MB
5	M, S				1870min	186MB	347min	84MB	146min	48MB	29s	19MB
6	M, L				5000min	187MB	660min	77MB	250min	56MB	38s	21MB
7	S, M, L				205min	58MB	370s	16MB	305s	16MB	4s	16MB

operations and the stated complexity analysis is therefore incorrect.

VI. Implementation, Optimization and Benchmarks

The CNA is implemented in Python 3 [7] with C extensions. This allows a clean and flexible OO design and facilitates targeted optimization. IP addresses and port numbers are represented directly by Python integers. Boxes are represented as Python 8-tuples (representing 4 intervals) and encapsulated into class objects in order to allow attachment of traces, annotations and firewall rule actions. Subspaces are represented as Python lists. The pure-Python prototype is relatively slow and has high memory consumption, but can already be used for security reviews involving firewalls with small and medium-sized rule-sets.

First, note that in the absence of Network Address Translation (NAT), which is rarely deployed in security critical networks, firewalls can be arbitrarily reordered, as exactly those packets that make it through *all* of them are part of the final reachability space. In particular, a good selection of the first firewall to be processed can have significant performance benefits. Benchmarks must therefore always be seen together not only with the relevant firewall configurations, but also their processing order.

A. Benchmarks

In order to determine performance and to examine the performance impact of different optimizations, we give a selection of benchmark results[1] in Table I. Times are CPU times including input data parsing and result output. Memory sizes are the whole process memory footprint, excluding shared areas (libraries). The calculations were done using Linux (Debian Squeeze 32bit) on an AMD Phenom II X4 970 CPU with 3.5GHz, using only one CPU at a time. Memory was set to the 4GB memory model and the machine was running kernel 3.4.7 from kernel.org without any special optimizations. Python version used was 3.1.

Lines $1, 2, 3$ of Table I describe the firewall configurations used. These are firewall configurations deployed in the real world. They have a flat form (no sub-chains) and a default-drop policy.

Line 4 and following lines of Table I give benchmarks for different firewall combinations. The order of the firewalls is important as the first one has to be completely represented in memory, which causes effort $O(|\mathrm{FW}_1|^2)$ (where $|\mathrm{FW}_k|$ is the number of rules in firewall FW_k). The effort for each additional firewall in the chain is $O((|\mathrm{WS}_i|+|\mathrm{FW}_i|)\cdot|\mathrm{FW}_i|)$ and hence higher in the worst case. But when starting with a firewall with small rule-set, we observed that a later combination with a firewall with a large rule-set does often not increase the WS size significantly, as most rules of the larger firewall do not apply. For that case, the complexity goes effectively down to $O(|\mathrm{WS}_i| \cdot |\mathrm{FW}_i|)$, which is a lot smaller than $O(|\mathrm{FW}_i|^2)$ if $|\mathrm{FW}_i|$ is large but $|\mathrm{WS}_i|$ is small. If the firewall processed first has a much larger rule-set than the others, we have observed that processing it will often dominate the runtime.

The columns "rule-set size" give the number of rules in the raw input in vendor format (including groupings, lists, etc.), the normalized number of rules without optimization and the optimized rule-set size. Benchmarks are given only for TCP for brevity, UDP and ICMP analysis have comparable results. We do not have benchmarks for comparison against a policy, as we do not have a sufficiently formalized policy and hence looking directly at reachability was more efficient. Comparison with a policy would incur effort comparable to adding one more firewall configuration in the size of the negated policy specification. The idea is that nothing must be able to pass through the given firewall chain *and* an additional firewall representing the negated policy, with the negated policy representing all forbidden traffic.

As can be seen in Table I, each evaluated optimization step has significant impact on observed run-time. The final implementation with all optimizations included has very reasonable performance even in the presence of firewalls

[1] While in theory there is no difference between theory and practice, in practice there is and benchmark results are very much subject to this limitation. Hence the stated benchmark results only give a rough idea about runtime, memory footprint and effects of different optimizations.

with large rule-sets.

B. Firewall Evaluation Sequence Optimization

The benchmarks demonstrate that the selection of the first firewall to be processed has a huge impact on performance. For the first firewall, the Work Set can grow for each rule application as it has to be completely represented in memory, while for later firewalls only rules that have a non-empty intersection with the Work Set can increase Work Set size by splitting elements already contained in it.

If the first firewall contains a large number of rules that allow traffic through that is later dropped by the other firewalls, then all these irrelevant rules will cause significant load on the memory allocator that can be avoided with a different selection for the first firewall to be processed. Our experiences show that the most restrictive firewall configuration should be processed first. In many scenarios, this will be the smallest firewall configuration, measured in number of rules.

C. Rule-Set Representation Optimization

Firewall configurations in a vendor-format often allow more complex specifications, such as lists or groupings of multiple sources, destinations or services. Decomposing such input rules into rules using a single box each can results in a number of normalized rules that is a lot higher than needed. The reason is that many resulting rules will be overlapping or adjacent in such a way that they can be combined. The column "opt." under "rule-set size" in Table I states the reduced number of rules after optimization and the column "input opt." gives the improved run times and memory footprints. The runtime for the input optimization itself is small, as it only works with a focus of one raw input rule at a time.

Note that global box combination would be possible, but combining boxes from different raw rules has two problems: First, if both *accept* and *drop* rules are present, the combination algorithm has to take rule sequence into account. And second, in this approach a box cannot be labeled with the single raw firewall rule it originated from. This makes the identification of policy-violating rules in the end-result difficult.

D. Trace Reduction

While the original prototype retained traces for all operations that changed a box, it turns out these full traces are only beneficial for debugging. In a security analysis, only *accept* and *drop* actions are relevant and hence it is enough to add trace information to a box when it is added to the Accept Set or Drop Set. It is not necessary to trace when boxes are reduced or split in the Work Set. Hence, traces were reduced accordingly. This also means that there can be at most one trace entry per firewall in each box contained in the result. The column "trace reduction" in Table I states the additional performance gains. Note that trace reduction was benchmarked with input optimization applied as well.

E. Core-Loop Ported to C

In a last step, the core loop function `apply_rule()` was ported to C and embedded into the Python code. Contrary to Figure 7, WS, AS and DS are passed to `apply_rule()` and are manipulated in-place according to the rule action. This puts expensive operations, such as data-structure manipulations, into the C code. No other special optimizations were done for the C code and in particular the standard GNU libc memory allocator was used. The column "core-loop ported to C" in Table I states final performance figures. Note that trace reduction and rule-set representation optimization was applied as well.

In addition, we performed a benchmark calculation for deployed firewall configuration "XL". It has a normalized rule-set size of 2.8 million rules, which reduces to 300'000 rules after input optimization. Raw rule number is 95. Representing configuration XL in memory took 20h of CPU time and resulted in a memory footprint of 900MB. This shows that firewall configurations of this size can still be processed with the CNA with reasonable effort.

The C code can keep box description efficiently in structs and does not need any wrapping and unwrapping of tuple elements and can therefore speed up execution massively, while at the same time reducing memory footprint significantly. However, the unit tests written in Python can still be applied by exposing the interval and box operations implemented in C to Python via the class interface. This helped significantly in the optimization effort.

VII. Advanced Optimization

The algorithm described so far compares each working set element against each rule. This leads to effort linear in the size of the Work Set and linear in the size of the rule set. This is problematic for large inputs. At the same time, for typical firewall rule sets, most elements of any given Work Set do not intersect most rules and hence a large part of the effort is wasted. If it were possible determine a subset of the Work Set that has a higher likelihood of intersecting a given rule r efficiently, a significant speed-up could be obtained. One such possibility is represented by interval search trees.

A. Interval Search Trees

Different types of interval search trees are known. They include trees that support searching with a point, where the result consists of all intervals in a given set that include the point, and searching with an interval, where the result includes all intervals that intersect the given search interval. We need the second variant.

TABLE II
BENCHMARKS: WORK SET AS ARRAY VS. WORK SET AS INTERVAL SEARCH TREE

No	Firewall or Firewall sequence	rule-set size(s) (opt.)	array	interval search tree
1	A	100	0.06s, 8.5MB	0.06s, 8.5MB
2	B	7.5k	4.8s, 27MB	1.2s, 28MB
3	C	1.3M	163h, 15GB	96min, 15GB
4	A,B	100, 7.5k	1s, 22MB	1s, 22MB
5	B,C	7.5k, 1.3M	16:05min, 2.6GB	2:47min, 2.6GB
6	A,B,C	100, 7.5k, 1.3M	2:49min, 2.6GB	2:45min, 2.6GB

As we want to represent the Work Set in an interval search tree, we also need efficient insertion and deletion of intervals from an already constructed tree. Unfortunately, many interval search tree variant do not support these operations efficiently and to the best or our knowledge, no multi-dimensional interval search tree variant can support insertion, deletion and searching with an interval, efficiently.

Due to these restrictions, we selected the interval trees from [8], Section 14.3. These are one-dimensional interval search trees constructed from balanced trees and support all operations we need efficiently. In [8], they are constructed on top of red-black trees as they are claimed to be simpler to implement than alternatives. As an implementation using AVL trees generally gives a smaller tree-height, we adapted the idea from [8] to AVL trees and used them as basis for our implementation.

The complexity of performing an interval search on an interval search tree with n elements is $O(k \cdot log(n))$, with k the number of results. For large k, the overall effort is bound by n, as each tree element is at most inspected once. For example, when the search result includes the full set of tree elements, the effort is only $O(n)$ and not $O(n \cdot log(n))$.

As one-dimensional interval search trees can only handle one component of the 4 different dimensions represented in a box, the idea is to use the most selective dimension of the set of multi-dimensional sets in the interval search, and then iterate linearly over the results as before. For typical large firewall rule sets, the most selective interval is the destination IP address interval. It is possible to use a different dimension. It would also be possible to use several interval search trees for the different dimensions, and then, for a given rule, perform the interval search in each dimension and then continue processing with the smallest result. It should be noted that using one-dimensional interval trees does not decrease the theoretical worst-case complexity of the algorithm and hence effectiveness has to be demonstrated by benchmark calculations.

B. Adjusting the Implementation

The core loop modified to use an interval search tree is shown in Figure 9. The `WS`, `AS` and `DS` are now kept as elements of an interval search tree, different from the linear array that was used before. The key effort reduction lies in reducing the Work Set size in `apply_rule()` by performing an interval search on the complete Work Set with the destination IP interval of the rule r. Only elements of the `WS` that intersect this interval in their destination IP component are added to the `WS_reduced` and have the complex box intersection algorithm applied to them.

```
apply_firewall(WS, FW):
  AS := ∅      /* Accept Set */
  DS := ∅      /* Drop Set   */
  for r ∈ in FW:     /* r: box of a rule */
    I := apply_rule(WS, r)
    WS := WS - I     /* reduce Work Set  */
    if r is accept:  AS := AS ∪ I
    if r is drop:    DS := DS ∪ I
  return(AS, DS)

apply_rule(WS, r):
  I := ∅
  WS_reduced := interval_search(WS, r)
  for b ∈ WS_reduced:        /* b is a box */
    i := b ∩ r
    I := I ∪ i
  return(I)
```

Fig. 9. Pseudo code from Figure 7 modified for interval search trees

C. Rules in an Interval Search Tree

An alternative to putting the Work Set elements into an interval search tree is putting the rules into one. The core loop in `apply_rule()` of Figure 9 would then have to be changed to select an element of the Work Set and then apply all rules to it in turn. The set of all rules would first be restricted using the interval search tree to those rules that intersect, for example, the destination IP interval of the Work Set element being processed.

At a first glance, this looks attractive: the rule-set does not change and hence tree construction does only happen once and no additions or deletions are performed on the tree. Unfortunately, the use of the interval search tree for the rules changes the application order of the rules. Rule sets with accept and drop rules can change their semantics whenever an accept and a drop rule are switched with regard to application order.

This means that while it is possible to apply the idea of using interval search trees for the rule sets, it only works correctly for rule sets that are all accept or all drop rules, with a possible final drop or accept, respectively. While many rule sets observed in practice have this form, some of the largest ones we have encountered do not and hence we are unwilling to accept this limitation.

A second, less problematic, limitation is that if rule application order is changed, it becomes more difficult to determine which rule actually accepted or dropped a specific packet. This ambiguity arises when a specific packet could have been accepted (dropped) by a rule R1 or a rule R2, but rule order determines which one actually does it. This becomes meaningful if it is necessary to determine which rule exactly processed a packet, for example if the packet is to be tagged for policy-based routing or a similar application.

D. Benchmarks for the Interval Search Tree Optimization

As the CNA is subject to on-going optimization, the experimental setup and base-line have changed. In particular redundant element copying and inefficient handling of element traces has been eliminated, resulting in a different baseline than thr one given in Table I. At the same time an updated benchmark firewall set was used that is similar in nature to the older one used for Table I, but changes all firewalls to some degree and includes one much larger firewall rule set. To prevent accidental confusion of the benchmark rule sets in the two tables, the firewalls in Table II have been named differently.

The Benchmarks in Table II were performed on an AMD Phenom II core with 3.4GHz core clock and 32GB available memory. The benchmarks were compiled and run in 64 bit mode, using gcc 4.7.2, Python 3.1.3 on Linux kernel 3.10.11. The characteristics of this setup are very similar to the one used for Table I, except for the 64 bit memory model.

The second column of Table II gives the firewall or firewall sequence processed left-to-right. Single firewalls are given as the process of representing a single firewall in memory is the same as processing it as the first element of a chain. The 3rd column lists the optimized rule-set sizes, similar to the 5th column of Table I. For a sequence of firewalls, the individual sizes are stated. The 4th column of Table II gives the runtimes and memory footprint with the classical array-based Work Set representation. These numbers include the full process including input parsing and result output. Finally, the last column of Table II lists execution time and memory footprint with the Work Set placed into an interval search tree.

E. Discussion

As can be seen, for some benchmarks, the advantage of using interval search trees is significant. In particular for computations with large reachabilities and hence large Work Set sizes, a massive speed improvement can be observed.

For computations with small Work Sets, like the firewall sequences ABC or AB, the speed-up is small or non-existent. The main reason is that storing the Work Set in an interval search tree is slower than storing it in an array. At the same time we do not observe any measurable slow-down due to the use of interval search trees and the memory footprint remains nearly the same.

The benchmark results support the claim that representing the Work Set in an interval search tree is superior, as the overhead created by the tree is compensated by smaller box intersection effort even in cases where restrictive firewalls are processed first and small Work Sets ensue. Tests with a synthetic, tiny first firewall that generates a Work Set of only 4 elements combined with firewalls B and C from Table II confirm that even in this extreme case, use of interval search trees does not slow down the computation to any measurable degree. Hence there is no need to retain the old, array-based Work Set representation.

As the optimization using interval search tree retains the full flexibility and expressiveness of the original CNA implementation, and does not increase memory consumption or CPU load even in the worst cases examined, use of interval search trees represents a significant improvement in the usefulness of the CNA for the processing and analysis of large firewall rule sets.

VIII. Performing Advanced Analysis Tasks

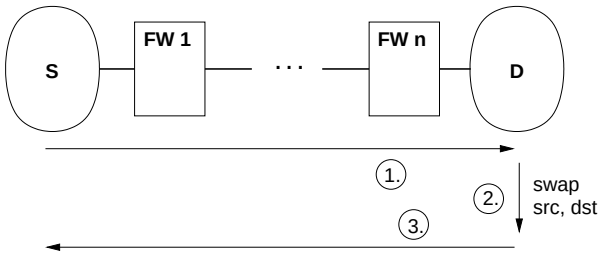
There are two common analysis tasks we have not yet described in detail. One is checking for presence or absence of bidirectional reachability. This answers the question whether a *connection* can be established through a series of firewalls. The second one is checking a chain of firewalls for compliance with a formalized policy. While we have anticipated this task earlier, we now describe how to perform it.

A. Computing Two-Sided Reachability

Two-sided Reachability allows determining whether an agent in the source network S can use a service offered in the destination network D that needs a *connection*, for example any service offered over TCP, or a *response*, as for example TCP port scanning, where TCP SYN packets are sent and potentially answered by ICMP packets. It allows limited comparison with scan results (for example from `nmap` [9]), which are sometimes used to verify a firewall deployment. Figure 10 gives the idea on how to obtain a two-sided reachability result.

B. Verifying Policy Compliance

Policies can be represented as an undesired reachability U, with the meaning that if anything in $U \subseteq M$ is actually reachable through the firewalls, then the policy is violated.




```
in: S, D, FW_0, ..., FW_n
out: AS2n (final reachability), DS0,...,DS2n

WS0 := S × <all> × D × <all>
(AS0, DS0) := apply_firewall(FW0, WS0)
WS1 := AS0
...
(ASn, DSn) := apply_firewall(FWn − 1, WSn − 1)
WSn + 1 := swap_src_and_dst(ASn)
(ASn + 1, DSn + 1) :=
          apply_firewall(FWn, WSn + 1)
WSn + 2 := ASn + 1
...
(AS2n, DS2n) :=
          apply_firewall(FW2n − 1, WS2n − 1)
```

Fig. 10. Calculating bidirectional reachability.

A firewall representing U is constructed by adding accept rules for all traffic components that are undesirable and a final drop rule that drops everything else. In a sense, this firewall acts as a filter that only leaves the undesired components of the actual reachability through a sequence of firewalls.

To test policy compliance, the actual network reachability A on each critical network path is calculated. Let V be the policy-violating reachability. Then $V = A \cap U$. If V is non-empty, all elements of V represent violations. The non-compliant firewall rules can be identified by looking at the trace information attached to elements of V, which they inherit from A.

A rarer compliance test is whether desired reachability is actually present. It can be used to determine which firewall of a firewall chain blocks desired traffic. Here, the desired reachability R is intersected with the subspace D that represents all dropped packets. If the intersection $V = D \cap R$ is nonempty, then parts of R will be dropped by some firewall drop rule and will not be part of the network reachability. As above, the problematic firewall rules can be identified from the traces attached to elements (boxes) of V, which they inherit from D.

Other compliance tests are possible and can be implemented when needed.

IX. Lessons Learned

Input Data: When converting firewall configuration data from customers, we found that significant effort may be needed to account for deviations from expected format convention and outright errors. We expect that for large firewall configurations some manual adaption may be hard to avoid. In seems that in their desire to accommodate customer requests, firewall vendors sometimes allow their customers to do things that are not advisable with regard to clean structuring and consistency, such as overlapping network groups, empty network groups and increasingly more action keywords in new versions. Some of these require manual intervention in order to map them to a unified firewall model. In addition, the right mapping may depend on the actual analysis task to be performed.

Software Engineering: Both, prototyping in Python and providing full, meaningful unit-tests provided hugely beneficial in creating a correctly working prototype and in making sure optimizations did not introduce additional errors. As the same time, keeping the Python-layer as "glue" on top of the implementation of the core loop in C allows for very efficient configuration and scripting of arbitrary analyses. The chosen implementation approach can be qualified as a success and is highly recommended for similar projects.

Performance: We found that run-time and memory footprint allow analysis of large and very large firewalls on standard hardware. This result is unexpected, as the underlying problems are algorithmically not efficient. We theorize that the reason lies in the fact that real-world firewall deployments only sparingly use most of the possibilities that firewalls offer (for example, mixing *accept* and *drop* rules excessively) as the firewall configuration still has to be created and maintainable by human beings.

X. Related Work

Reachability Analysis: One alternative to using the CNA is network scanning, for example with nmap [9]. It should be noted however that this suffers from the limitations that routing affects scanning and that normal scanning cannot find undesired *unidirectional* reachability.

Algorithmic Firewall Analysis: It is possible to formalize firewall functionality with a suitable logic and then use approaches from automated theorem proving to derive properties and check against violation of conditions. Work in this area includes FIREMAN [5] by Yuan, Mai, Su, Chen, Chuah and Mohapatra, which uses a BDD (Binary Decision Diagram) representation. The idea of using BDDs is developed further by Liu and Gouda [6], [10], with the introduction of Firewall Decision Diagrams (FDDs).

A different approach based on Decision Diagrams is described by Liu in [11]. It allows the checking of properties given a specific firewall rule set. The properties are formalized as firewall rules with wildcards, e.g., that no traffic must flow to or from IP address `1.2.*.*`. This formalization has a close relation to our policy checking approach where we formalize a policy as an additional firewall. Unfortunately, [11] only tested performance for

small real-world firewall rule-sets up to 661 rules and hence a meaningful performance comparison with our approach is not possible.

Firewall Models: Leporati and Ferretti [12] use Tissue-like P Systems to model connected sets of firewalls. In [13], Bourdier and Cirstea employ rewrite systems to model firewall filtering and translation rules. Bera, Dasgupta and Ghosh [14] is an example for the use of a Boolean SAT solver to verify firewall ruleset properties.

Firewall Redundancy Analysis: Firewall redundancy analysis is aimed at identifying and removing redundancies in a firewall ruleset, such as rules that have overlapping boxes. While a prolific theoretical field, its relevance to practice is minor. For example, [15], [16] and [17] deal with this aspect of firewall analysis.

Query Engines: The query-engine of Mayer, Wool and Ziskind [18] answers questions on whether a specific packet would traverse a set of firewalls by using a rule-based simulator. This is mostly useful to determine the impact of specific firewall configuration changes. Its value in a complete firewall security analysis is limited. The Margrave Tool [19] uses a similar approach.

Commercial Tools: A commercial firewall analyzer is offered by AlgoSec [20]. This tool seems to be targeted at maintenance and administration of large numbers (up to 1000) of firewalls. Commercial firewall maintenance tools with limited audit capabilities are also offered by Tufin [21] and FireMon [22].

XI. Conclusion and Future Work

We have designed and implemented the CNA (Consecom Network Analyzer), a tool that calculates network reachability through a series of firewalls given as a Layer 4 abstraction by symbolic simulation. The primary use is for real-world security audits that examine firewalls with large rule-sets. While using set operations to model firewalls is simple, to the best of our knowledge we are the first to demonstrate that an abstraction based on intervals is efficient enough to calculate reachability through large deployed firewall configurations in practically useful time and with moderate memory footprint, while at the same time retaining the capability to annotate each result sub-set with a full trace of the applied firewall rules. Automated result annotation is essential when analyzing firewall chains that include firewalls with a large number of rules.

We also have demonstrated the effect of a series of implementational and algorithmic optimizations on execution time and memory-footprint. The last step is the application of ideas from geometrical search to use one-dimensional interval search trees for reduction of ineffective rule applications to Work Set elements. The benchmarks given include performance on large firewall rule sets actually deployed in real applications.

One possible direction for future work is further investigation into how multi-dimensional geometric search structures could be used to improve efficiency even more. Primary issues are that most known multi-dimensional search structures do not handle updates (additions and deletions) efficiently. Using these structures for the CNA would mean finding design and implementation trade-offs that work well for real problems, even if their theoretical worst-case performance is bad.

A second possibility for future work is the adaption of the CNA *IPv6* addresses. With the current system, this can be done by swapping out 32 bit unsigned integers for 128 bit unsigned integers in the C code. Python already handles all integers as long-numbers and no change in the Python code would be needed. However, input-parsing and result output would have to be adapted. However, the larger memory footprint may have significant impact on the actual implementation and may require specific additional optimizations to retain efficiency.

Finally, the CNA could be extended to handle subchains in firewall rule sets. At this time, subchains can be handled by a preprocessing step. A native implementation of subchains into the CNA core code by adding suitable rule actions could speed up processing of subchains significantly.

Acknowledgments: We thank the Swiss KTI and Consecom AG for funding parts of this work and the anonymous reviwers for their helpful suggestions.

References

[1] A. Wagner and U. Fiedler, "Firewall Analysis by Symbolic Simulation," in *The Seventh International Conference on Internet Monitoring and Protection (ICIMP 2012)*, 2012, pp. 95–100.

[2] "Wikipedia: Hyperrectangle," http://en.wikipedia.org/wiki/Hyperrectangle, last visited December 2013.

[3] H. S. M. Coxeter, *Regular Polytopes*, 3rd ed. New York: Dover, 1973.

[4] P. Eronen and J. Zitting, "An Expert System for Analyzing Firewall Rules," in *Proc. 6th Nordic Worksh. Secure IT Systems*, 2001, pp. 100–107.

[5] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra, "FIREMAN: A Toolkit for FIREwall Modeling and ANalysis," in *IEEE Symposium on Security and Privacy*, 2006, pp. 199–213.

[6] A. X. Liu and M. G. Gouda, "Diverse Direwall Design," in *IEEE Transactions on Parallel and Distributed Systems, 19(8)*, August 2008.

[7] "The Python Homepage," http://python.org/, last visited December 2013.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Mit Press, 2009.

[9] "Nmap Security Scanner," http://nmap.org/, last visited December 2013.

[10] A. X. Liu and M. G. Gouda, "Firewall Policy Queries," in *IEEE Transactions on Parallel and Distributed Systems, 20(6)*, June 2009.

[11] A. X. Liu, "Formal Verification of Firewall Policies," in *2008 IEEE International Conference on Communications, ICC '08*, 2008, pp. 1494 – 1498.

[12] A. Leporati and C. Ferretti, "Modelling and Analysis of Firewalls by (Tissue-like) P Systems," in *Romanian Journal of Information Science and Technology, Vol. 13, No 2*, 2010, pp. 169–180.

[13] T. Bourdier and H. Cirstea, "Symbolic Analysis of Network Security Policies Using Rewrite Systems," in *Symposium on Principles and Practices of Declarative Programming*, 2011, pp. 77–88.

[14] P. Bera, P. Dasgupta, and S. Ghosh, "Formal Analysis of Security Policy Implementations in Enterprise Networks," in *International Journal of Computer Networks and Communications (IJCNC), Vol. 1, No. 2*, 2009, pp. 56–73.

[15] S. Pozo, A. Varela-Vaca, and R. Gasca, "A Quadratic, Complete, and Minimal Consistency Diagnosis Process for Firewall ACLs," in *24th IEEE International Conference on Advanced Information Networking and Applications*, 2010.

[16] K. Karoui, F. B. Ftima, and H. B. Ghezala, "Formal Specification, Verification and Correction of Security Policies Based on the Decision Tree Approach," in *International Journal of Data and Network Security 08/2013; 3(3):92-111*, 2013.

[17] P. Rajkhowa, S. M. Hazarika, and G. R. Simari, "An Application of Defeasible Logic Programming for Firewall Verification and Reconfiguration," in *Quality, Reliability, Security and Robustness in Heterogeneous Networks, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Volume 115*, 2013, pp. 526–542.

[18] A. J. Mayer, A. Wool, and E. Ziskind, "Offline firewall analysis," *Int. J. Inf. Sec.*, vol. 5, no. 3, pp. 125–144, 2006.

[19] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisher, and S. Krishnamurthi, "The Margrave Tool for Firewall Analysis," in *USENIX Large Installation System Administration Conference (LISA)*, 2010.

[20] "Algosec Homepage," http://www.algosec.com/, last visited December 2013.

[21] "tufin Homepage," http://www.tufin.com/, last visited December 2013.

[22] "FireMon Homepage," http://www.firemon.com/, last visited December 2013.