ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Information Security
www.infsec.ethz.ch

Adrian Schüpbach

Firewall Testing with NAT

Semesterarbeit
Wintersemester 2005/2006
ETH Zürich, 07. Februar 2006

Betreuerin: Diana Senn
Professor: David Basin

Zusammenfassung

Firewalls sind ein wichtiger Bestandteil von Netzwerken. Um sicher zu sein, dass eine Firewall sich so verhält, wie man das möchte, muss man sie testen. Es wird getestet, ob die Regeln das aussagen, was man möchte und ob sich die Firewall den vorgegebenen Regeln nach verhält.

In einer frühreren Arbeit wurde von Gerhard Zaugg mit `fwtest` ein Tool geschaffen, das es ermöglicht, Firewalls anhand vorgegebener Testfälle zu prüfen. In der vorliegenden Arbeit wurde dieses Tool um NAT-Funktionalität erweitert. Der Tester hat neu die Möglichkeit, jedes Testpaket in zwei Varianten zu spezifizieren: so wie es gesendet werden soll und so wie er es erwartet. Dabei ist es auch möglich, Variablen zu verwenden, um gewisse Werte zu kennzeichnen, die in mehreren Testpaketen gleich sein sollen, bei denen man im Voraus aber nicht weiss, welchen Wert sie haben.

Zusätzlich wurde `fwtest` so angepasst, dass es als eine Instanz gestartet wird.

Abstract

Firewalls are important components in networks. To be sure, that a firewall behaves as specified, it must be tested. The aim of the test is to show that the firewall does what we expect from it. It is that the firewall rules conform to the security policy, and that the firewall implements the rules correctly.

In a previous thesis, Gerhard Zaugg created a tool, called `fwtest`, which can test firewalls with specified testpackets. In the thesis at hand fwtest was enhanced with NAT support. The tester has now the possibility to specify each packet in two ways: how it has to be sent and how he expects it. It is also possible to use variables to mark values which have to be the same in some packets, but whose value is not known a priori.

Further `fwtest` was modiefied so, that it has now to be started as a single instance.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

1.1. Firewalls testen

Firewalls spielen eine zentrale Rolle in Netzwerken. Sie sind für den Schutz in Netzwerken zuständig. Eine Firewall besitzt eine Reihe von Regeln, die definieren, welche Verbindungen zugelassen und welche abgeblockt werden sollen.

Es ist wichtig, dass die Regeln das aussagen, was man vorgegeben hat und dass sich die Firewall genau wie die spezifizierten Regeln verhält. Um sicher sein zu können, muss die Firewall getestet werden.

fwtest testet eine Firewall, indem Pakete auf der einen Seite der Firewall ins Netz geschleust werden und auf der anderen Seite der Firewall gesnifft werden. Dies geschieht in beide Richtungen. In einer Testpaket-Datei (TP-File) sind die zu sendenden Pakete definiert. fwtest prüft anhand dieser Datei und anhand der gesnifften Pakete, ob sich die Firewall wie erwartet verhält. Unregelmässigkeiten werden in eine log-Datei gespeichert.

1.2. NAT

Viele Firewalls benützen NAT (Network Address Translation). Beim Senden eines Paketes vom internen ins externe Netzwerk über die Firewall, sendet die Firewall das Paket mit ihrer eigenen IP-Nummer und einem von ihr gewählten Port ins externe Netzwerk. In der Firewall gibt es ein Mapping von originaler IP und originalem Port zur Firewall-IP und dem gewählten Port. Ein Paket, das vom externen Netzwerk über die Firewall ins interne Netzwerk gesendet werden soll, wird an die IP-Nummer der Firewall und an den von ihr gewählten Source-Port gesendet, wie eine ganz normale Antwort auf ein Paket. Die Firewall sendet nun das empfangene Paket ins interne Netzwerk an die originale IP-Nummer und den originalen Port. Diese beiden Werte kennt sie wegen des Mappings. Ein Beispiel ist in Bild 1 gegeben.

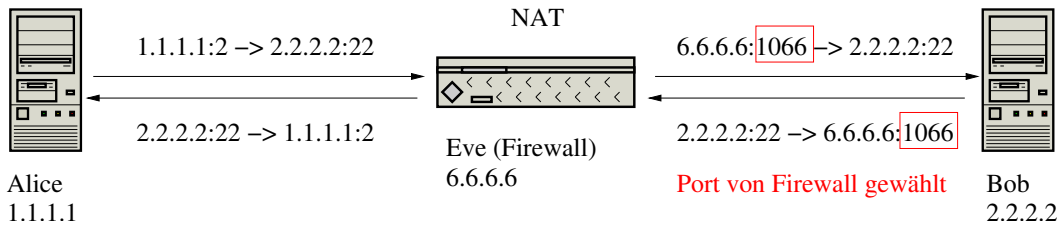

Abbildung 1: Verbindung über eine Firewall mit NAT

Das ursprünglich von einem internen Rechner gesendete Paket hat nach der Firewall also eine andere Form. Es hat eine andere Sende-IP-Nummer und/oder einen anderen Sende-Port. Bisher wurden die Pakete durch die Gleicheit des empfangenen und des gesendeten Paketes über die IP-Nummern und die Port-Nummern identifiziert. Mit NAT ist das nicht mehr möglich. Eine eindeutige ID für jedes Paket ist nötig, um ein Paket eindeutig zu identifizieren. Mit NAT muss es die Möglichkeit geben zu spezifizieren, wie ein Paket nach der Firewall aussehen muss.

Da man den von der Firewall gewählten Source-Port nicht kennt, muss man eine Variable dafür benützen können. Diese soll für die ganze Verbindung den gleichen Wert behalten. Deshalb wird ihr der Wert des Ports des ersten empfangenen Paketes zugewiesen und dieser

Wert bleibt für den ganzen Testfall gleich. Die Variable ist also innerhalb eines Testfalles gültig.
Variablen können auch für IP-Nummern benützt werden, da die Source-IP-Nummer auch verändert ist.

1.3. Synchronisation

Um eine Firewall zu testen, mussten zwei Instanzen von fwtest gestartet werden. Die Synchronisation der Verbindung geschah über die absolute Zeit.
Bei Tests von fwtest wurde festgestellt, dass diese Synchronisation mit der absoluten Zeit unzuverlässig ist. Deshalb soll fwtest jetzt nur noch als eine Instanz gestartet werden müssen. Dafür wird es so erweitert, dass es auf zwei Netzwerk-Schnittstellen sowohl Pakete ins Netzwerk einschleusen kann, wie auch von beiden Netzwerk-Schnittstellen sniffen kann. Somit entfällt die Synchronisation und die einzige Instanz von fwtest hat eine globale Sicht über die Verbindungen. Abbildung 2 zeigt dieses Szenario.

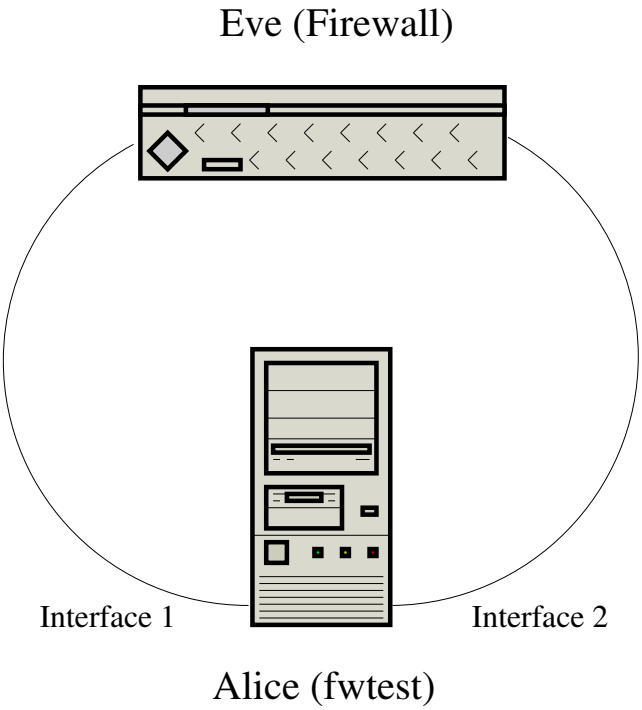

Abbildung 2: fwtest als eine Instanz auf einem Rechner verbunden mit zwei Schnittstellen, je eine mit der Innen- und Aussenseite der Firewall

1.4. Ziel

Das Programm fwtest wurde bereits von [Zau05] programmiert. Es soll jetzt so erweitert werden, dass auch Firewalls mit NAT, wie oben beschrieben, getestet werden können.
Die zweite grosse Änderung ist das Benützen einer zweiten Netzwerk-Schnittstelle, damit fwtest als eine Instanz gestartet werden kann. Wichtig ist die Elimination der Zeit. Die Synchronisation zwischen dem Senden und Empfangen auf den zwei Netzwerk-Schnittstellen soll nicht zeitabhängig sein.

1.5. Voraussetzungen für die Änderungen

1.5.1. TP-File und Parser

Da fwtest neu NAT unterstützen soll, müssen sowohl die zu sendenden Pakete, wie auch die erwarteten Pakete in einer Datei definiert werden können. Ausserdem muss es die Möglichkeit geben, Variablen benützen zu können. Das Bedingt ein neues TP-File-Format und auch einen neuen Parser für dieses neue Format.

1.5.2. Identifikation der Pakete

Damit fwtest auch Firewalls mit NAT testen kann, muss die Paketidentifikation unabhängig von IP-Nummern und Port-Nummern sein. Eine Identifikation über eine eindeutige ID, die im Paket mitgesendet wird, ist notwendig.

1.5.3. Sniffen von zwei Schnittstellen

fwtest muss jetzt von zwei Schnittstellen sniffen. Es muss unterscheiden können, von welcher Schnittstelle ein Paket gerade empfangen wurde. Damit es von beiden Schnittstellen sniffen kann, darf dies nicht mehr in einem blockierenden Modus geschehen. Trotzdem soll fwtest so gebaut werden, dass möglichst wenig Prozessorzeit für die Ausführung benötigt wird.

Viele Funktionen, die für die Netzwerk-Schnittstelle benützt werden, benutzten global Variablen. Das ist für mehr als eine Schnittstelle nicht mehr möglich. Deshalb müssen sie so angepasst werden, dass Variablen als Parameter übergeben werden können. Die Funktionen werden flexibler und sind für beliebige Schnittstellen brauchbar.

1.5.4. Timeout

Da nicht immer alle gesendeten Pakete ankommen, weil sie zum Beispiel von der Firewall geblockt werden, ist ein Timeout notwendig. Dieser Wert soll leicht geändert werden können.

1.6. Speicherbedarf während der Ausführung von fwtest

fwtest soll sehr grosse Testläufe, die aus sehr vielen Paketen bestehen, ausführen können. Damit nicht zu viel Speicher während der Ausführung verwendet wird, sollen die Pakete erst dann generiert werden, wenn sie wirklich benötigt werden, das heisst, kurz vor dem senden. Sobald sie gesendet wurden, können sie wieder gelöscht werden. Wichtig ist nur, dass man die Paketdefinition im Speicher behält, damit man empfangene Pakete identifizieren kann.

2. Design

2.1. Sniffen von zwei Schnittstellen

Bisher wurde nur von einer Schnittstelle gesnifft. Deshalb konnte im Hauptprogramm die Funktion `pcap_loop` mit dem Parameter `INFINITE` Aufgerufen werden. Die Funktion kehrt so nie zur Funktion `main` zurück. Das geht für zwei Schnittstellen nicht, denn wenn die Funktion nicht mehr zum Hauptprogramm zurückkehrt, gibt es für das Hauptprogramm keine Möglichkeit, von der anderen Schnittstelle zu sniffen. Der Aufruf der Funktion `pcap_loop` erfolgt im Hauptprogramm.

Es muss also eine andere Lösung gefunden werden. In den nächsten 3 Unterkapiteln wird eine Lösung für dieses Problem vorgestellt und beschrieben.

2.1.1. pcap_dispatch

Eine Möglichkeit ist das Verwenden der Funktion `pcap_dispatch`. Diese Funktion kehrt zurück, sobald von der übergebenen Schnittstelle etwas gelesen werden konnte. Wenn man diese Funktion einmal für die erste Schnittstelle, dann für die zweite Schnittstelle aufruft und diese beiden Aufrufe innerhalb einer endlosen while-Schleife hat, wird abwechslungsweise von beiden Schnittstellen gelesen. Das kommt einem `pcap_loop` mit`INFINITE` schon Nahe für zwei Schnittstellen. Die Deskriptoren der offenen Schnittstellen sind in einem blockierenden Modus sind. Das heisst, dass das Lesen eines Paketes den Funktionsaufruf solange blockiert, bis tatsächlich ein Paket gelesen werden konnte. Pakete der zweiten Schnittstelle werden so aber erst gelesen, wenn der erste Aufruf zurückkehrt, das heisst, wenn ein Paket von der ersten Schnittstelle gelesen werden konnte.

2.1.2. nonblocking-mode

Die beiden Deskriptoren der offenen Schnittstelle sollen in den nicht-blockierenden Modus versetzt werden. Dies kann mit `pcap_setnonblock` erreicht werden. Jetzt kehrt die Funktion `pcap_dispatch` sofort zurück, auch wenn kein Paket gelesen werden konnte. Die while-Schleife wird jetzt dauernd ausgeführt. Das braucht allerdings quasi 100% der Prozessorzeit. Dieses Problem muss behoben werden, denn wenn kein Paket angekommen ist, soll eine while-Schleife, die nichts macht, nicht unnötig den Prozessor belasten.

2.1.3. "any"-Device

Um die while-Schleifen-Ausführung anzuhalten, wenn kein Paket angekommen ist, muss eine Schnittstelle im blockierenden Modus sein. Wenn eine der beiden effektiv benützten Schnittstelle in diesem Modus ist und die andere nicht, bleibt wieder das Problem, dass die eine Schnittstelle nur gelesen wird, wenn ein Paket bei der blockierenden Schnittstelle ankommt. Es soll aber egal sein, von welcher Schnittstelle zuerst das Paket kommt.

Die Lösung ist das Benützen der Pseudo-Schnittstelle "any". Diese Schnittstelle liest von allen effektiv vorhandenen Schnittstellen gleichzeitig. Wenn der Deskriptor dieser Schnittstelle im blockierenden Modus bleibt, kann unter den beiden Leseaufrufen der effektiven Schnittstellen ein Leseaufruf für die "any"-Schnittstelle stehen. Somit wird die while-Schleife an dieser stelle blockiert, bis auf irgendeiner Schnittstelle ein Paket ankommt. Die "any"-Schnittstelle liest das Paket aus, es ist aber nicht verloren damit, weil es auch für eine reelle Schnittstelle

angekommen ist und dort noch nicht gelesen wurde. Die while-Schleife macht jetzt eine Iteration. Sie kommt zu den beiden Leseaufrufe der effektiven Schnittstellen. Eine der beiden liest jetzt das angekommene Paket.

Nur von der "any"-Schnittstelle lesen ist keine gute Idee, denn falls noch mehr als die beiden von fwtest benützen Schnittstellen auf dem Testrechner vorhanden sind, würde die "any"-Schnittstelle diese Pakete auch einlesen und fwtest übergeben. Interessiert ist man aber nur an Paketen, die von einer der beiden von fwtest benützen Schnittstellen kommen.

2.2. Timeout

Gesendete Pakete müssen nicht unbedingt ankommen. Es gibt zwei Gründe, warum ein Paket nicht ankommt:

- Das Paket ist auf dem Weg zum Ziel verlorengegangen
- Das Paket wird von der Firewall geblockt

Da nicht jedes gesendete Paket bei der anderen Schnittstelle von fwtest auch wieder ankommt, weil es zum Beispiel auf dem Weg dorthin von der Firewall geblockt wird, muss ein Timeout eingeführt werden, damit die Programmausführung weitergehen kann. Jedesmal, wenn ein Paket gesendet wird, wird der Timeout gestartet. Falls das Paket innerhalb der Zeit des Timeoutes nicht ankommt, wird zum nächsten Zeitpunkt übergegangen.

Dieser Timeoutwert ist im Moment fix in der Headerdatei `main.h` festgesetzt. In einer späteren Semesterarbeit soll dieser Wert variabel werden. Es gibt verschiedene Gründe, das Timeout variabel zu wählen. Es gibt Netzwerke, die eine lange Verzögerung haben, weil sie zum Beispiel stark belastet sind. Hier wäre ein langer Timeout nötig, damit nicht die meisten Pakete erst nach dem Ablaufen des Timeouts ankommen. Das Ergebnis würde so verfälscht werden. Man sieht aber im log-File, dass das Ergebnis aufgrund eines zu kurz gewählten Timeouts so herausgekommen ist daran, dass Pakete im erwarteten Zeitpunkt fehlen, aber unerwartete Pakete später ankommen und diese Pakete genau die ID der vorher erwarteten Pakete haben. Der Timeout sollte aber nicht zu lange sein, da sonst ein Testlauf unnötig in die Länge gezogen wird, v.a. wenn viele Pakete von der Firewall geblockt werden und somit für fast jedes gesendete Paket der volle Timeout abgewartet werden muss.

Interessant ist auch eine Erweiterung von fwtest, so dass fwtest selber erkennt, wenn der Timeoutwert schlecht gewählt ist und ihn anpasst und die Tests wiederholt. Erkennen kann fwtest einen schlecht gewählten Timeout daran, dass viele Pakete erst nach dem Ablaufen des Timeouts ankommen.

2.3. Globale Zeit

Die Zeit des alten TP-Files muss eliminiert werden, da das zu grosse Synchronisationsprobleme gibt. fwtest soll als eine einzige Instanz auf einem Rechner ausgeführt werden. Für die Sendereihenfolge der Pakete und für Timeouts muss es aber dennoch eine Art Zeit geben. Diese Zeit soll neu eine relative Zeit zwischen den Paketen sein.

Gewisse Verbindungen können nicht gleichzeitig stattfinden. Die Firewall identifiziert eine verbindung anhand der IP- und Portnummern. Wenn zwei im TP-File spezifizierte Verbindungen die gleiche IP- und Portnummern verwenden, dürfen sie nicht gleichzeitig ausgeführt werden. Die Paket-ID, die auch als relative Zeit dient (siehe 2.7) muss so gewählt werden, dass

die Pakete zur richtigen Zeit gesendet werden, so dass es keine Konflikte gibt. Somit muss die relative Zeit in fwtest global sein, das heisst, sie muss für alle Testfälle gleich sein. Es darf nicht sein, dass ein Testlauf schneller als ein anderer läuft und er eine eigene Zeit hat, obwohl das für jeden Testfall einzeln gehen würde. Aber so könnten Konflikte nicht verhindert werden. Mit der globalen Zeit kann fwtest warten, ein Paket eines Testfalles zu senden, bis alle Pakete auch aus anderen Testfällen gesendet wurden, die effektiv vorher gesendet werden müssen.

Mit jedem Ablaufen des Timeouts wird die globale Zeit um einen Zeitpunkt weitergeschaltet. Somit können die Pakete des neuen Zeitpunktes gesendet werden und müssen innerhalb des Timeouts empfangen werden. Ein Timer im Hauptprogramm stellt die globale Zeit jeweils zum nächsten Zeitpunkt.

2.4. NAT

fwtest soll so erweitert werden, dass NAT unterstützt wird. Mit NAT ändert die Firewall gewisse Felder der Pakete. Zum Beispiel wird die Sendeportnummer geändert und das Paket wird mit der IP-Nummer der Firewall gesendet. Die Portnummer wird beim ersten Paket von der Firewall gewählt, muss dann aber für die ganze Verbindung gleich bleiben. Das heisst, dass in diesem Fall die Portnummer variabel sein muss und diese Variable für jedes weitere Paket derselben Verbindung eingesetzt werden muss.

2.5. Paket-ID

Jedes Paket muss eindeutig identifizierbar sein. Da fwtest auch NAT unterstützen soll, geht eine Identifikation über IP- und Port-Nummern nicht gut. Die Firewall kann die Portnummern ändern und das Paket mit ihrer eigenen IP-Nummer und einer neuen Portnummer senden.

Eine Lösung ist die Paket-ID und die Testfall-ID mitzuschicken. Somit ist jedes Paket eindeutig identifizierbar, ohne die IP- und die Port-Nummern vergleichen zu müssen.

2.5.1. ID als Daten mitschicken

Für TCP und UDP-Pakete kann die Testfallnummer und die Paket-ID als Daten mitgeschickt werden. Beide Felder können so 32Bits gross sein. Der Zahlenbereich ist somit sehr gross. Es ist einfach, Pakete anhand dieser beiden Nummern in den Daten eindeutig identifizieren zu können. Das Problem mit dieser Methode ist, dass es nicht ICMP-konform ist. In ICMP-Paketen kann man nicht frei Daten anhängen. Ein weiteres Problem kann in der Firewall auftreten, falls sie auch die Daten überprüft oder manipuliert.

2.5.2. ID im IP-ID-Feld mitschicken

Das IP-ID-Feld ist eine Möglichkeit, eine ID mitzuschicken. Dieses Feld ist nur 16Bits gross. Der Zahlenbereich wird somit massiv kleiner. In diesem IP-ID-Feld muss dann eine Kombination der Testfallnummer und der Paket-ID geschickt werden. Diese Variante geht auch für ICMP. Mit der kleinen Grösse dieses Feldes können aber nur noch wenige Pakete gesendet werden. Die 16Bits müssen in zwei Bereiche aufgeteilt werden, damit man die Testfallnummer und die Paket-ID mitsenden kann.

2.5.3. Mapping von Testfallnummer und Paket-ID auf 16Bit-ID

Um eine flexible Aufteilung der 16 verfügbaren Bits aus dem IP-ID-Feld zu erhalten, kann dieses Feld als fortlaufender Zähler verwendet werden. Das erste Paket aus dem ersten Testfall erhält die Zahl 0000. Das nächste Paket, egal ob aus dem gleichen Testfall oder aus einem neuen Testfall, erhälte die nächste Nummer. Somit können insgesamt 65535 Pakete gesendet werden. Die Aufteilung, in Testfälle und in Anzahl Pakete pro Testfall ist für die IP-ID somit nicht mehr relevant. Für ein ankommendes Paket wird das IP-ID-Feld gelesen und daraus, mit dem Mapping, die Testfallnummer und die Paket-Nummer gewonnen. Das ankommende Paket kann so eindeutig identifiziert werden.

2.5.4. ID-Mix aus Daten und IP-ID-Feld

Um mehr Pakete senden zu können, kann ein Mix aus den beiden Verfahren (2.5.1 und 2.5.3) gemacht werden. Für TCP- und UDP-Pakete wird die Testfallnummer und die Paket-ID als Daten mitgesendet und der IP-ID-Zähler wird nicht benützt und auch nicht verändert. Für ICMP-Pakete wird ein Mapping von der Testfallnummer und der Paket-ID auf einen eindeutigen 16Bit-Zähler gemacht. Dieser Zählerwert wird im IP-ID-Feld mitgesendet. Somit können 65535 ICMP-Pakete und $2^{32} * 2^{32}$ TCP- und UDP-Pakete gesendet werden.

2.5.5. Kommandozeilen-Schalter: Benützer entscheidet

Mit einem Kommandozeilen-Schalter kann der Benützer entscheiden, ob für TCP- und UDP-Pakete die ID als Daten oder im IP-ID-Feld mitgesendet werden soll. Wenn der Benutzer keinen Schalter angibt, wird die Standardeinstellung "IP-ID-Feld" angewendet.

2.5.6. Mapping

Das Mapping von Testfallnummern und Paket-ID's ist kein Problem, da fwtest jetzt als eine Instanz ausgeführt wird. Die gleiche Instanz führt das Mapping von Testfallnummern und Paket-ID's in IP-ID und umgekehrt aus.

2.6. Symboltabelle

Damit Variablen im TP-File benützt werden können, muss fwtest eine Symboltabelle haben. Es ist wichtig, dass die Symboltabelle eine Art Scope hat, da jede Variable innerhalb eines Testfalles gültig sind. Das heisst, dass der gleiche Variablenname in verschiedenen Testfällen vorkommen kann, aber es sich dann nicht um die gleiche Variable handelt. Somit müssen die Werte dieser Variablen auch nicht gleich sein.

Innerhalb des gleichen Testfalles wird der Wert einer Variable nur einmal zugewiesen und bleibt dann für den ganzen Testfall gültig. Wenn ein Variablenname innerhalb des gleichen Testfalles mehrfach vorkommt, heisst das also, dass es sich um die gleiche Variable handelt und dass somit ein bestimmter Wert an diesen Verschiedenen Orten vorkommen muss.

Variablen können auch in der Sendespezifikation von Paketen benützt werden. Das Paket wird dann mit dem zugewiesenen Wert dieser Variable gesendet. Die Variable muss vor dem Generieren und Senden des Paketes also zugewiesen werden. Falls das nicht der Fall ist, wird eine Warnung ausgegeben und das Paket wird nicht gesendet.

Die Symboltabelle muss initialisiert werden können. Nach der Initialisierung ist sie leer. Es müssen Variablen ohne Probleme eingefügt werden können, indem eine Funktion der Symboltabelle aufgerufen wird, der man den Variablenname und den Scope übergeben kann. Variablen müssen auch leicht gefunden werden können. Ein einziger Zeiger auf die Symboltabelle wird benötigt um Zugriff auf die ganze Tabelle zu haben.

Während des Parsens werden die Variablen in die Symboltabelle in den Scope des Testfalles eingefügt. Dieser Scope wird durch die Testfallnummer repräsentiert.

2.6.1. Datenstruktur

Die Symboltabelle besteht aus eine Hauptdatenstruktur, die die Scopes speichert und eine Datenstruktur pro Scope, die alle Variablen dieses Scopes speichert. Die Hauptdatenstruktur der Scopes ist ein binärer Baum. Jeder Knoten besitzt die Testfallnummer als Schlüssel für den Scope und hat einen Zeiger auf das erste Element der linearen Liste der Variablen des Scopes. Die Datenstruktur, die alle Variablen eines bestimmten Scopes speichert, ist eine einfach verkettete lineare Liste. Es gibt eine solche Liste pro Knoten im Baum. Jeder Schlüssel darf nur einmal im Baum vorkommen. Das geht ohne Probleme, da jede Testfallnummer eindeutig ist.

In jeder Variablen-Liste darf jeder Variablenname nur einmal vorkommen. Das bedeutet, dass jeder Variablenname, der mehrfach im gleichen Scope, das heisst im gleichen Testfall, vorkommt, auch die gleiche Variable meint. Da pro Testfall eine neue Liste erzeugt wird und im entsprechenden Konten gespeichert wird, können in verschiedenen Testfällen die gleichen Variablennamen vorkommen, ohne dass damit die gleiche Variable gemeint ist.

2.6.2. Einfügen von Variablen

Während des Parsens kann jede Variable, die gerade geparst wird, in die Symboltabelle eingefügt werden, indem der Name und die Testfallnummer übergeben wird. Falls die Variable noch nicht existiert, wird ein neues Element erzeugt, initialisiert und in die Variablen-Liste eingefügt. Falls der Knoten mit der angegebenen Testfallnummer noch nicht existiert, wird er erzeugt und in den Baum eingefügt. Anschliessend wird der Zeiger auf den Anfang der Variablen-Liste gesetzt. Zurückgegeben wird der Zeigr auf das neu kreierte Element.

Falls die Variable schon existiert, wird kein neues Element erzeugt, sondern es wird der Zeiger auf das bereits vorhandene Element zurückgegeben. Somit ist auch gewährleistet, dass keine Variable zweimal eingefügt werden kann und dass gleiche Variablennamen innerhalb eines Testfalles sich auch wirklich auf die gleiche Variable beziehen.

2.6.3. Suchen nach Variablen

Es kann nach einer bestimmten Variable in der Symboltabelle gesucht werden. Dazu muss der Name und die Testfallnummer übergeben werden. Falls die Variable existiert, wird der Zeiger auf das Element, das sie beschreibt, zurückgegeben, anderenfalls wird der Null-Zeiger zurückgegeben.

2.6.4. Initialisierung der Symboltabelle

Mit einer einfachen Funktion kann die Symboltabelle initialisiert werden. Sie ist dann leer.

2.6.5. Löschen der Symboltabelle

Zwei Funktionen sind für das Löschen der Symboltabelle verantwortlich. Eine Funktion löscht die übergebene Liste von Variablen-Elementen. Sie wird nur intern benützt. Die andere Funktion löscht den ganzen Baum, indem sie ihn postorder traversiert und in jedem Knoten mit der internen Funktion die Variablenliste und danach den Knoten selber löscht.

2.7. TP-File Design

Fwtest unterstützt jetzt auch das Senden von UDP- und ICMP-Paketen (siehe [Str06]). Die effektive Sendezeit der Pakete soll nicht mehr benützt werden, stattdessen soll es eine relative Zeit zwischen den Paketen geben. Ausserdem unterstützt fwtest neu auch das Testen von Firewalls, die NAT benützen. Für die Verwendung von fwtest für eine Firewall mit NAT, müssen im TP-File Variablen anstelle der Werte in den Paketen definiert werden können. Die Werte dieser Variablen werden von der Firewall im Betrieb festgesetzt und somit von fwtest während des Testlaufs zugewiesen. Daher ist ein neues TP-File-Format notwendig. Neu soll nicht das File das Protokoll für alle Pakete vorgeben, sondern das Protokoll, mit dem ein Paket gesendet wird, muss für jedes Paket einzeln spezifizierbar sein. Damit können TCP-, UDP- und ICMP-Testcases im gleichen TP-File gespeichert sein und von der gleichen fwtest-Instanz abgearbeitet werden.

Die Erweiterung von fwtest erlaubt es, nicht nur zu sendende Pakete zu spezifizieren, sondern auch, wie das entsprechende Paket aussehen soll, wenn es empfangen wird. Somit kann auch eine Firewall mit NAT getestet werden. Das bedingt auch Änderungen im TP-File. Erfahrungen mit dem alten TP-File-Format, die Elimination der Zeit und die grosse Anzahl Pakete, die in einem TP-File gespeichert werden können müssen, haben zu einigen Kriterien geführt. Die nachfolgenden Kriterien muss das neue Format erfüllen:

1. Pakete gehören zu einem Testfall
2. Pakete haben eine relative Zeit
3. Pakete haben eine eindeutige ID
4. Testfälle haben eine eindeutige ID
5. Darstellung soll übersichtlich sein
6. TP-File muss schnell geparst werden können
7. Datenstruktur darf nicht zu viel Speicher benützen, auch für viele Pakete
8. Der Kontext jedes Paketes darf nicht velorengehen
9. Pakete, die nichts miteinander zu tun haben, müssen gleichzeitig gesendet werden dürfen
10. Abhängigkeiten zwischen Paketen von verschiedenen Testfällen müssen berücksichtigt werden können

Aus diesen Kriterien entstanden Ideen, wie das neue TP-File und die dazugehörige Datenstruktur aufgebaut werden könnte. Diese werden in den folgenden Unterkapiteln (2.10 - 2.13) vorgestellt. Das definitive TP-File-Format und die definitive Datenstruktur werden in Kapitel 2.14 beschrieben.

2.8. Paketspezifikation – Idee 1

Jedes Paket kann als zu sendendes und als erwartetes Paket auf einer Zeile spezifiziert werden. Das ist eine Erweiterung des alten TP-File-Formates um ein "zweites" Paket. Die Synchronisation über die Zeit bereitet einige Probleme. Deshalb soll fwtest als eine Instanz auf einem Rechner laufen. Somit ist die Zeit im TP-File überflüssig und kann eliminiert werden.
Jedes Paket muss eindeutig identifiziert werden können. Deshalb kommt an Stelle der Zeit im neuen TP-File-Format eine ID.
Eine Zeile im TP-File für TCP sähe dann so aus:

```
<id> <Sendepaket> <erwartetes Paket>
```

Ein Paket besteht aus den Feldern:
`<srcip> <dstip> <srcport> <dstport> <flags> <seqnr> <acknr>`

fwtest soll neu auch NAT unterstützen. Gewisse Felder in einem Paket, zum Beispiel eine Portnummer, können von der Firewall geändert werden, müssen dann aber, solange die Verbindung bestehen bleibt, immer gleich bleiben. Das macht es nötig, dass diese Felder variabel sein können.

In der Spalte des erwarteten Pakets könnte auch ein "NOK" stehen, für Pakete, von denen man erwartet, dass sie von der Firewall geblockt werden. In dieser Spalte könnte auch ein "?" stehen, falls es egal ist, wie die Firewall reagiert.

Erfüllte Kriterien

2, 3, 6, 10

Nicht-erfüllte Kriterien

1, 5, 8

Über die Kriterien 4, 7, 9 kann nichts ausgesagt werden, da die Datei keine Testfall-Struktur hat, d.h. keine Testfälle speichert und es somit auch keine Testfall-ID's gibt. Somit sind die Pakete auch nicht in einem Kontext.

Am Ende des Kapitels 2.7 sind die Kriterien aufgelistet, die Erfüllt werden müssen. Anhand dieser Liste kann eine Evaluation der erfüllten und nicht-erfüllten Kriterien stattfinden. In Kapitel 2.14 wird die definitive Paketspezifikation gewählt und nochmals kurz zusammengefasst.

2.9. Paketspezifikation – Idee 2

Eine weitere Möglichkeit, Pakete zu speichern, ist eine strukturierte Darstellung der Pakete. Verschiedene Protokolle benötigen verschiedene Felder. Da im neuen TP-File jedes Paket mit einem anderen Protokoll gesendet werden kann und nicht mehr das File das Protokoll für alle Pakete wählt, muss das Protokoll, das ein Paket benützen soll, pro Paket angegeben werden. Die beste Möglichkeit, das zu verwendende Protokoll pro Paket anzugeben, ist, das direkt im TP-File beim Paket zu machen. So weiss der Parser auch, welche Felder dieses Paket, welches

er gerade am parsen ist, benützt. Die Erweiterung des Dateinamens des TP-Files muss somit auch nicht mehr der Protokollname sein.

Ein Paket könnte wie in Abbildung 3 dargestellt werden.

```
paket <id>{
    <protokoll>
    send{
        <veraenderliche Felder>
    }
    receive{
        <veraenderliche Felder>
    }
}
```

Abbildung 3: Darstellung eines Paketes in Paketspezifikation Idee 2

2.9.1. TCP und UDP

Für die unterstützten Protokolle TCP und UDP (siehe [Str06]) sähe das dann wie in Abbildung 4 und 5 aus.

```
paket <id>{
    TCP
    send{
        <srcip> <dstip> <srcport> <dstport> <flags> <seqnr> <acknr>
    }
    receive{
        <srcip> <dstip> <srcport> <dstport> <flags> <seqnr> <acknr>
    }
}
```

Abbildung 4: Spezifikation eines TCP-Paketes in Paketspezifikation Idee 2

2.9.2. TCPUDP

Es gibt Verbindungen und Services, die sowohl mit TCP wie auch mit UDP funktionieren. Um beide Möglichkeiten zu testen, ist es am einfachsten, wenn der Benutzer nur einmal die Pakete im TP-File definieren muss und dann mit einem Schalter beim Programmstart auswählen kann, ob er diese so definierten Tests mit dem TCP- oder dem UDP-Protokoll testen will.
Um eine Verbindung so definieren zu können, dass beim Programmstart ausgewählt werden kann, welches Protokoll benützt werden soll, kann der Benutzer im TP-File das Protokoll für ein Paket als TCPUDP definieren. Jenachdem, welche Kommandozeilen-Optionen der Benutzer angibt, werden Pakete, die als Protokoll `TCPUDP` haben, als TCP- oder als UDP-Paket gesendet. Pakete, die dieses Protokoll haben, werden wie normale TCP-Pakete im TP-File spezifiziert. Falls der Benutzer an der Kommandozeile UDP wählt, werden die Felder

```
paket <id>{
    UDP
    send{
        <srcip> <dstip> <srcport> <dstport>
    }
    receive{
        <srcip> <dstip> <srcport> <dstport>
    }
}
```

Abbildung 5: Spezifikation eines UDP-Paketes in Paketspezifikation Idee 2

<flags> <seqnr> <acknr> ignoriert. Falls er TCP angibt, wird dieses Paket als TCP-Paket gesendet.

```
paket <id>{
    TCPUDP
    senden{
        <srcip> <dstip> <srcport> <dstport> <flags> <seqnr> <acknr>
    }
    erwartet{
        <srcip> <dstip> <srcport> <dstport> <flags> <seqnr> <acknr>
    }
}
```

Abbildung 6: Spezifikation eines TCPUDP-Paketes in Paketspezifikation Idee 2

2.9.3. ICMP

Da ICMP für die verschiedenen ICMP-Codes, die es gibt, andere Felder benützt, gibt es für ICMP mehrere Formate (siehe [Str06]). Die Formate hängen vom jeweiligen ICMP-Code ab. Die ICMP-Codes sind für den Beützer nicht direkt ersichtlich. Sie werden implizit ausgewählt, indem der Name des ICMP-Paketes im TP-File als Paket-Typ gespeichert wird. So ist auch klar, welche Felder der Benutzer ausserdem für das Paket definieren muss und was für eine Bedeutung sie haben.

Für das unterstützte ICMP-echo sähe das wie in Abbildung 7 aus. Dieses Beispiel beschreibt ein ICMP-echo-Paket, d.h. der Code von "echo" wurde implizit ausgewählt. ICMP-echo braucht als weitere angaben die Sender- und Empfänger-IP-Nummer, einen Typ, eine ID-Nummer und eine Sequenz-Nummer. Für eine Beschreibung der einzelnen ICMP-Paketformate und aller Felder siehe [Str06].

2.10. Testfallspezifikation – Idee 1

Ein Testfall besteht aus mehreren Paketen. Jeder Testfall hat eine eindeutige Testfallnummer. Innerhalb eines Testfalles haben die Pakete eine eindeutige Nummer als ID. Die Pakete eines

```
paket <id>{
    ICMPecho
    send{
        <srcip> <dstip> <type> <idnr> <seqnr>
    }
    receive{
        <srcip> <dstip> <type> <idnr> <seqnr>
    }
}
```

Abbildung 7: Spezifikation eines ICMPecho-Paketes in Paketspezifikation Idee 2

Testfalles gehören zusammen, haben aber nichts mit Paketen anderer Testfälle zu tun. Um einen neuen Testfall zu spezifizieren, muss eine neue Nummer als Testfallnummer gewählt werden. Die Pakete im neuen Testfall können gleiche ID's haben, wie Pakete aus anderen Testfällen, weil die Paketnummern nur innerhalb des Testfalles eindeutig sein müssen. Jeder Testfall wird für sich abgearbeitet. Das heisst es gibt keine globale Zeit. Pakete eines Testfalles werden nacheinander gesendet, ohne auf Pakete anderer Testfälle zu warten. Sobald bei einem Testfall ein gesendetes Paket angekommen ist oder der Timeout abgelaufen ist, wird das nächste Paket gesendet. Jeder Testfall wird so abgearbeitet. Die Testfälle werden parallel, aber völlig unabhängig nach dieser Methode abgearbeitet. Da zwischen Paketen aus verschiedenen Testfällen keine Abhängigkeiten bestehen, kann ein ganzer Testfall gelöscht werden, ohne dass das einen direkten Einfluss auf die anderen Testfälle hat. Somit müssen aber die einzelnen Testfälle unabhängig voneinender sein. Ein TP-File sähe dann wie in Abbildung 8 aus.

```
testfall 00000001{
  1 gesendetes paket    erwartetes paket
  2 gesendetes paket    erwartetes paket
  ...                   ...
}

testfall 00000002{
  1 gesendetes paket    erwartetes paket
  2 gesendetes paket    erwartetes paket
  ...                   ...
}
```

Abbildung 8: Testfallspezifikation Idee 1

Die Pakete eines einzelnen Testfalles werden zeitlich nacheinander gesendet. Sie werden anhand ihrer ID sortiert und in dieser Reihenfolge gesendet. Die Testfälle werden gleichzeitig gestartet. Das nächste Paket (nächste Zeile) eines Testfalls wird gesendet, sobald das vorhergehende Paket ankommt oder ein bestimmter Timeout abgelaufen ist. Ein Paket muss nicht unbedingt ankommen, deshalb ist ein Timeout unbedingt notwendig. Im Paket werden die Testfallnummer und die ID als Daten mitgesendet. Die "Gleichheit" der Pakete kann somit über die Testfallnummer und die ID getestet werden. Da IP-Nummer und Port durch NAT

ev. geändert werden, ist ein Gleichheitstest über die Testfallnummer und die ID zuverlässiger als das Prüfen der Gleichheit über die IP- und Port-Nummern. Die ID's sind aufsteigende Nummern und sagen nichts über die absolute Zeit resp. über die absoluten Pausen zwischen den Paketen aus. Sie bestimmen nur eine Reihenfolge und somit eine relative Zeit. Da die Paketdaten aneinandergereihte Zahlen sind und die Beduetung erst durch die Interpretaion gegeben ist, könnte ein Typ-Fehler auftreten, nämlich jedesmal, wenn man empfangene Zahlen anders interpretiert, als sie der Sender usprünglich gemeint hat. Um das zu verhindern, kann man explizit den Typ (die Bedeutung) eines Wertes mitsenden. Es ist ratsam im Datenfeld vor jeder Nummer den Typ anzugeben, d.h. z.B. `''tsf=''` (als Integerwert) Testfallnummer, `''idn=''` (als Integerwert) id. Somit ist die Wahrscheinlichkeit eines Typfehlers kleiner. Somit hängt die Bedeutung nicht nur von der Interpretation einer Zahlenfolge ab einem festgesetzten Offset ab, sondern wird direkt durch den mitgesendeten Typ bestimmt.

Erfüllte Kriterien

1, 2, 3, 4, 5, 6, 7, 8, 9

Nicht-erfüllte Kriterien

10

Mögliche Datenstruktur:

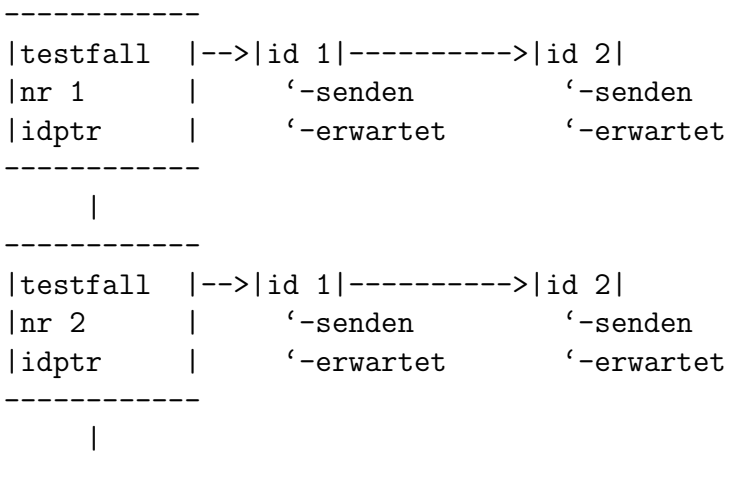

Abbildung 9: Datenstruktur für Testfallspezifikation Idee 1

Es gibt eine verkettete Liste mit allen Testfällen. Für jeden Testfall gibt es nochmals eine verkettete Liste, in der die Pakete der ID nach gespeichert sind. Ein idptr am Anfang der Liste jedes Testfalles zeigt auf die Paket-ID, die als letzte gesendet wurde.

Start:

Um die Testfälle zu starten, muss das Programm fwtest durch die Testfall-Liste gehen und bei allen Testfällen ID1 senden.

Fortsetzung:

Wenn das Paket mit der Testfallnummer x und der ID y ankommt, muss das Programm fwtest in der Testfallliste x zu dieser Testfallnummer y gehen, diesen idptr zur nächsten ID verschieben und das dazugehörende Paket senden. Dies wird solange gemacht, bis bei allen Testfällen alle ID's gesendet wurden.
Da nicht jedes Paket ankommen wird, weil es zum Beispiel von der Firewall geblockt wird oder im Netz verloren geht, muss spätestens nach einem festgesetzten Timeout zur nächsten Paket-ID gesprungen werden.

2.11. Testfallspezifikation – Idee 2

Um das Problem zu lösen, dass Testfälle nicht auf bestimmte Pakete anderer Testfälle warten können, kann man einen Zähler als eine Art globale "Uhr" benützen. Pakete dürfen erst gesendet werden, falls die ID des Paketes <= der Zähler ist. Der Zähler wird erst inkrementiert, falls die ganze Liste einer Zeit abgearbeitet wurde. Die ganze Liste einer Zeit ist abgearbeitet, wenn alle Pakete dieser Zeit gesendet wurden und diese Pakete entweder angekommen sind oder der Timeout für jedes Paket dieser Zeit abgelaufen ist. In diesem Moment kann man davon ausgehen, dass keine weiteren Pakete dieser Zeit mehr gesendet werden und auch dass keine weiteren Pakete dieser Zeit mehr ankommen. Eventuell könnte ein extrem verspätetes Paket noch ankommen, das dann als nicht-erwartetes Paket dieser Zeit geloggt wird. Ein Beispiel ist in Abbildung 10 gegeben.

Zum Zeitpunkt 1 werden die Pakete mit ID 1 der Testfälle 1 und 2 gesendet. Für die ID 1 wird ein Timeout gestartet. Der Zähler wird erhöht, sobald alle Pakete mit ID 1 ankommen oder ihr Timeout abgelaufen ist. Falls ein Paket nach dem Ablaufen des Timeouts ankommt, gilt es als zu spät angekommen. Es gibt für ein zu spät angekommenes Paket zwei log-einträge. Einmal wird es als nicht wie erwartet geloggt und einmal als zu spät angekommen. Zeitpunkt 2 ist eine leere Liste => Der Zähler geht sofort auf 3. Zum Zeitpunkt 3 werden Pakete mit der id 3 aller Testfälle gesendet. Sobald alle Pakete mit id 3 empfangen wurden, geht der Zähler auf 4 (resp. dann auf 5, weil 4 eine leere Liste ist). Zum Zeitpunkt 5 wird das Paket 5 aus Testfall 3 gesendet. Sobald es ankommt, kann Paket 6 aus Testfall 4 gesendet werden.

Die ID's sagen etwas über die Zeit und die Pausen aus. Sie stellen eine relative Zeit zwischen allen Paketen, aus dem gleichen und aus verschiedenen Testfällen, dar.

Erfüllte Kriterien

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Nicht-erfüllte Kriterien

keine

mögliche Datenstruktur:

Eine mögliche Datenstruktur ist eine verkettete Liste, die nach den Zeitpunkten sortiert ist. Von jedem Zeitpunkt aus gibt es eine neue verkettete Liste, die die Pakete enthält, die zu diesem Zeitpunkt gesendet werden sollen. Diese Liste enthält die Testfallnummer, damit Pakete

```
testfall 00000001{
  1 gesendetes paket    erwartetes paket
  3 gesendetes paket    erwartetes paket
  ...                   ...
}

testfall 00000002{
  1 gesendetes paket    erwartetes paket
  3 gesendetes paket    erwartetes paket
  ...                   ...
}

testfall 00000003{
  3 gesendetes paket    erwartetes paket
  5 gesendetes paket    erwartetes paket
  ...                   ...
}

testfall 00000004{
  1 gesendetes paket    erwartetes paket
  3 gesendetes paket    erwartetes paket
  6 gesendetes paket    erwartetes paket
  7 gesendetes paket    erwartetes paket
  ...                   ...
}
```

Abbildung 10: Beispiel TP-Datei für Spezifikation Idee 2

in der Liste eindeutig bestimmt werden können. Jeder Knoten der Liste beschreibt das Paket, wie es gesendet werden muss und wie es erwartet wird. Falls zu einem bestimmten Zeitpunkt kein Paket gesendet werden soll, ist die Paketliste, die von diesem Zeitpunkt ausgeht, eine leere Liste.

Das Programm fwtest startet beim ersten Zeitpunkt. Alle Pakete werden "gleichzeitig" (ist nur kurz nacheinander möglich) gesendet. Danach wird ein bestimmter Timeout gestartet. Pakete, die vor dem Ablaufen des Timeouts ankommen, können mit dem Erwarteten verglichen werden. Sie gelten als angekommen. Sobald der Timeout abläuft, gilt dieser Zeitpunkt als beendet. Die Liste dieses Zeitpunktes ist fertig abgearbeitet und fwtest kann zum nächsten Zeitpunkt übergehen. Wenn alle Zeitpunkte abgearbeitet sind, kann das Programm beendet werden. Diese Datenstrultur sähe dann wie in Abbildung 11 aus. Die Variable `unsigned int zähler` beschreibt den aktuellen Zeitpunkt, der bearbeitet wird.

2.12. Testfallspezifikation – Idee 3

Ein Testfall kann aus mehreren Zeilen bestehen. Jeder Testfall hat eine eindeutige Nummer. Zeilen, die mit der gleichen Nummer beginnen, gehören zum gleichen Testfall. In jeder Zeile ist ein Paket definiert und zwar als zu sendendes und als erwartetes Paket. Es gehört zum Testfall

```
unsigned int zaehler;
-------------
|zeitpunkt 1|-->|testfall 1 id 1|-->|testfall 2 id 1|-->|testfall 4 id 1|
-------------     `-senden            `-senden            `-senden
      |           `-erwartet          `-erwartet          `-erwartet
-------------
|zeitpunkt 2|-->
-------------
      |
-------------
|zeitpunkt 3|-->|testfall 1 id 3|-->|testfall 2 id 3|-->|testfall 3 id 3|-->|testfall 4 id 3|
-------------     `-senden            `-senden            `-senden            `-senden
      |           `-erwartet          `-erwartet          `-erwartet          `-erwartet
-------------
|zeitpunkt 4|-->
-------------
      |
usw.
```

Abbildung 11: Datenstruktur für Testfallspezifikation Idee 2

```
testfall        id
00000001        1       gesendetes paket        erwartetes paket
00000003        1       gesendetes paket        erwartetes paket
00000002        1       gesendetes paket        erwartetes paket
00000003        2       gesendetes paket        erwartetes paket
00000002        2       gesendetes paket        erwartetes paket
00000001        2       gesendetes paket        erwartetes paket
```

Abbildung 12: Beispiel TP-Datei für Testfallspezifikation Idee 3

mit der Nummer, die am Anfang der Zeile steht. Auch jedes Paket hat eine eindeutige Nummer. Die Pakete und die Testfälle müssen nicht sortiert in der TP-Datei gespeichert werden. Ein Beispiel ist in Abbildung 12 gegeben.

Erfüllte Kriterien

1 ,2 ,3 ,4 ,6, 7, 8 ,9, 10

Nicht-erfüllte Kriterien

5

Für diese Möglichkeit kann die gleiche Datenstruktur wie in Kapitel 2.11 verwendet werden.

2.13. Testfallspezifikation – Idee 4

Das TP-File besteht aus verschiedenen Zeitpunkten. Jeder Zeitpunkt kann mehrere Pakete enthalten. Ein Zeitpunkt ist eindeutig. Jedes Paket gehört zu einem Testfall, der eine eindeutige Testfallnummer hat. Die Testfallnummer wird am Anfang der Zeile innerhalb jedes Zeitpunktes

geschrieben. Jedes Paket hat eine eindeutige Paketnummer. Das Paket wird als zu sendendes und als erwartetes Paket spezifiziert.
fwtest arbeitet wie in 2.11.

```
unsigned int zaehler;

zeitpunkt 00000001{
    00000003    1       gesendetes paket        erwartetes paket
    00000001    1       gesendetes paket        erwartetes paket
    00000002    1       gesendetes paket        erwartetes paket
}

zeitpunkt 00000002{
    00000002    2       gesendetes paket        erwartetes paket
}

zeitpunkt 00000003{
    00000001    2       gesendetes paket        erwartetes paket
    00000003    2       gesendetes paket        erwartetes paket
    00000002    3       gesendetes paket        erwartetes paket
}
```

Abbildung 13: Beispiel TP-Datei für Testfallspezifikation Idee 4

Erfükkte Kriterien

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Nicht-erfüllte Kriterien

keine

Auch für diese Möglichkeit kann die gleiche Datenstruktur wie in Kapitel 2.11 verwendet werden.

2.14. Wahl des TP-File Designs

Wir haben uns für die Testfall-Idee 2 (siehe Kapitel 2.11) entschieden. Diese Idee hat die Vorteile, dass das TP-File nach Testfällen aufgebaut ist und dass die Pakete eine relative Zeit haben. Jedes Paket und jeder Testfall hat eine eindeutige Nummer. Der Kontext der Pakete bleibt erhalten. Mit dieser Idee können alle Kriterien, die in Kapitel 2.7 festgelegt wurden, eingehalten werden. Auch Idee 4 erfüllt alle Kriterien, aber Idee 2 entspricht eher dem "menschlichen Denken".

In den nächsten drei Unterkapiteln wird die definitiv gewählte Lösung nochmals kurz zusammengefasst.

2.14.1. TP-File Aufbau

Ein TP-File besteht aus mehreren Testfällen. Jeder Testfall hat eine eindeutige Nummer. Ein Testfall kann aus mehreren Paketen bestehen. Innerhalb des Testfalles haben Pakete eine eindeutige Nummer. Die folgenden zwei Unterkapitel zeigen einen TP-File-Aufbau und beschreiben die EBNF des TP-Files.

2.14.2. Aufbau-Beispiel des TP-Files

Ein TP-File sieht dann wie in Abbildung 14 aus.

```
testcase <nr>{
  packet <time>{
    <PROTO>
    send{
      xxx
    }
    receive{
      yyy
    }
  }
  packet...
}
testcase...
```

Abbildung 14: Aufbau des TP-Files

Je nach <PROTO> sind xxx und yyy anders. Für mehr Details siehe Kapitel 2.14.3. Wenn kein Paket erwartet wird, ist receive leer, also `receive{}`.

Die <nr> eines Testfalles ist eindeutig. <time> kann mehrmals vorkommen in der gleichen TP-Datei, aber nur einmal pro Testfall. Pakete mit der gleichen "Zeit" (<time>) können bzw. sollen gleichzeitig gesendet werden, zumindest aber vor allen Paketen mit einer späteren Zeit und nach allen Paketen mit einer früheren Zeit. Die eindeutige Identifikation eines Paketes erfolgt durch die Kombination von <nr> und <time>.

Eine globale Variable enthält den aktuellen Zeitpunkt. Erst, wenn alle Pakete dieses Zeitpunktes gesendet und wieder empfangen wurden oder der Timeout für diesen Zeitpunkt abgelaufen ist, kann zum nächsten Zeitpunkt übergegangen werden. Der Wert des Timeouts ist fix in einer Header-Datei gegeben und verändert sich während des Testens nicht (siehe 2.2).

2.14.3. EBNF des TP-Files

```
TP-File         ::=Testcase {Testcase}.
Testcase        ::="testcase" INT "{" Packet {Packet} "}".
Packet          ::="packet" INT "{" PacketSpec "}".
Packetspec      ::=TCP | UDP | TCPUDP | ICMPecho | ICMPunreach | ICMPredir
                   | ICMPtexc | ICMPtstamp.
TCP             ::="TCP" "send" "{" TCPflds "}" TCPrcv.
```

```
TCPrcv          ::="receive" "{" TCPflds "}" | "receive" "{" "}" | "?".
TCPflds         ::= SRCIP DSTIP SRCPRT DSTPRT FLAGS SEQ ACK.
UDP             ::="UDP" "send" "{" UDPflds "}" UDPrcv.
UDPrcv          ::="receive" "{" UDPflds "}" | "receive" "{" "}" | "?".
UDPflds         ::= SRCIP DSTIP SRCPRT DSTPRT.
TCPUDP          ::="TCPUDP" "send" "{" TCPflds "}" TCPrcv.
ICMPecho        ::="ICMPecho" "send" "{" ICMPechoflds "}" ICMPechorcv.
ICMPechorcv     ::="receive" "{" ICMPechoflds "}" | "receive" "{" "}" | "?".
ICMPechoflds    ::=SRCIP DSTIP INT ID SEQ.
ICMPunreach     ::="ICMPunreach" "send" "{" ICMPunreachflds "}" ICMPunreachrcv.
ICMPunreachrcv  ::="receive" "{" ICMPunreachflds "}" | "receive" "{" "}" | "?".
ICMPunreachflds ::=SRCIP DSTIP CODE ORIGID.
ICMPredir       ::="ICMPredir" "send" "{" ICMPredirflds "}" ICMPredirrcv.
ICMPredirrcv    ::="receive" "{" ICMPredirflds "}" | "receive" "{" "}" | "?".
ICMPredirflds   ::=SRCIP DSTIP CODE GWIP ORIGID.
ICMPtexc        ::="ICMPtexc" "send" "{" ICMPtexcflds "}" ICMPtexcrcv.
ICMPtexcrcv     ::="receive" "{" ICMPtexcflds "}" | "receive" "{" "}" | "?".
ICMPtexcflds    ::=SRCIP DSTIP CODE ORIGID.
ICMPTstamp      ::="ICMPtstamp" "send" "{" ICMPtstampflds "}" ICMPtstamprcv.
ICMPtstamprcv   ::="receive" "{" ICMPtstampflds "}" | "receive" "{" "}" | "?".
ICMPtstampflds  ::=SRCIP DSTIP ID SEQ OTIM TTIM RTIM.

OPTINT          ::=INT | "-".
ID              ::=OPTIND.
INT             ::=Digit {Digit}.
Digit           ::="0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".
SRCIP           ::=IP.
DSTIP           ::=IP.
IP              ::=IPnr | Var.
IPnr            ::=INT"."INT"."INT"."INT.
SRCPRT          ::=PORT.
DSTPRT          ::=PORT.
PORT            ::=INT | Var | Text.
CODE            ::=INT.
ORIGID          ::=INT"."INT.
TYPE            ::=INT.
SEQ             ::=OPTIND.
ACK             ::=OPTIND.
IDNR            ::=OPTIND.
FLAG            ::="S"|"A"|"F"|"R"|"U"|"P"|"s"|"a"|"f"|"r"|"u"|"p".
FLAGS           ::=FLAG {FLAG}.
Letter          ::="a"|"b"|..|"z".
Capitalletter   ::="A"|"B"|..|"Z".
Letters         ::=Capitalletter | Letter.
Var             ::=Capitalletter {Letters}.
Text            ::=Letter {Letters}.
```

2.14.4. Datenstruktur

Die Datenstruktur besteht aus eine verketteten Liste. Jeder Knoten beschreibt einen Zeitpunkt und hat eine weitere verkettete Liste mit Paket-Beschreibungen für diesen Zeitpunkt. In diesen Listen ist beschrieben, wie die Pakete gesendet werden sollen und wie sie erwartet werden. Die Zähler-Variable zeigt den aktuellen Zeitpunkt.

Die Zeitpunkte für die einzelnen Pakete können vom Benützer beliebig gewählt werden, sie müssen nicht der Reihe nach aufsteigen gewählt sein. Zum Beispiel ist die Zeitpunktfolge "1,3,9,7,4" für fwtest eine gültige Folge und es ist nicht nötig, für die fünf Pakete die Zeitpunkt "0,1,2,3,4" zu wählen. Die Zeitpunkt-Liste muss der Zeit nach aufsteigend sortiert sein. Die Sortierung der Liste wird im Parser von fwtest gemacht.

```
unsigned int zaehler;
-------------
|zeitpunkt t1|-->|testfall n1 id 1|-->|testfall n2 id 1|-->|testfall n3 id 1|
-------------     `-senden            `-senden            `-senden
      |           `-erwartet          `-erwartet          `-erwartet
-------------
|zeitpunkt t2|-->
-------------
      |
-------------
|zeitpunkt t3|-->|testfall n1 id 3|-->|testfall n2 id 3|-->|testfall n3 id 3|-->|testfall n4 id 3|
-------------     `-senden            `-senden            `-senden            `-senden
      |           `-erwartet          `-erwartet          `-erwartet          `-erwartet
-------------
|zeitpunkt t4|-->
-------------
      |
usw.
t1<t2<t3<t4
```

Abbildung 15: Definitive Datenstruktur

2.14.5. Parser

Der Parser liest das ganze TP-File ein. Es wird vollständig geparst. Die ganze Datenstruktur, die oben in Kapitel 2.14.4 beschrieben ist, wird vollständig durch den Parser aufgebaut. Somit muss im Hauptprogramm an der Datenstruktur nichts mehr geändert werden. In der Datenstruktur ist jetzt die vollständige Beschreibung der Pakete, wie sie gesendet werden sollen und wie sie erwartet werden. Diese Informationen genügen vollständig, um Pakete generieren zu können. Da ein durch `libnet` generiertes Paket zusätzlich viel Speicher benötigt, werden die Pakete nicht wie bis anhin alle beim Start generiert. Sie werden erst generiert, wenn sie wirklich gebraucht werden, d.h. kurz vor dem Senden. Generiert werden müssen nur zu sendende Pakete. Sobald ein Paket gesendet wurde, kann das generierte Paket gelöscht werden, da die Beschreibung in der Datenstruktur ausreicht, um das Paket eventuell nochmals zu generieren. Das empfangene Paket kann mit dem erwarteten Paket verglichen werden, ohne dass das erwartete Paket generiert werden muss.

2.15. Control Flow Graph

Abbildung 16 zeigt den Control Flow Graph von fwtest.

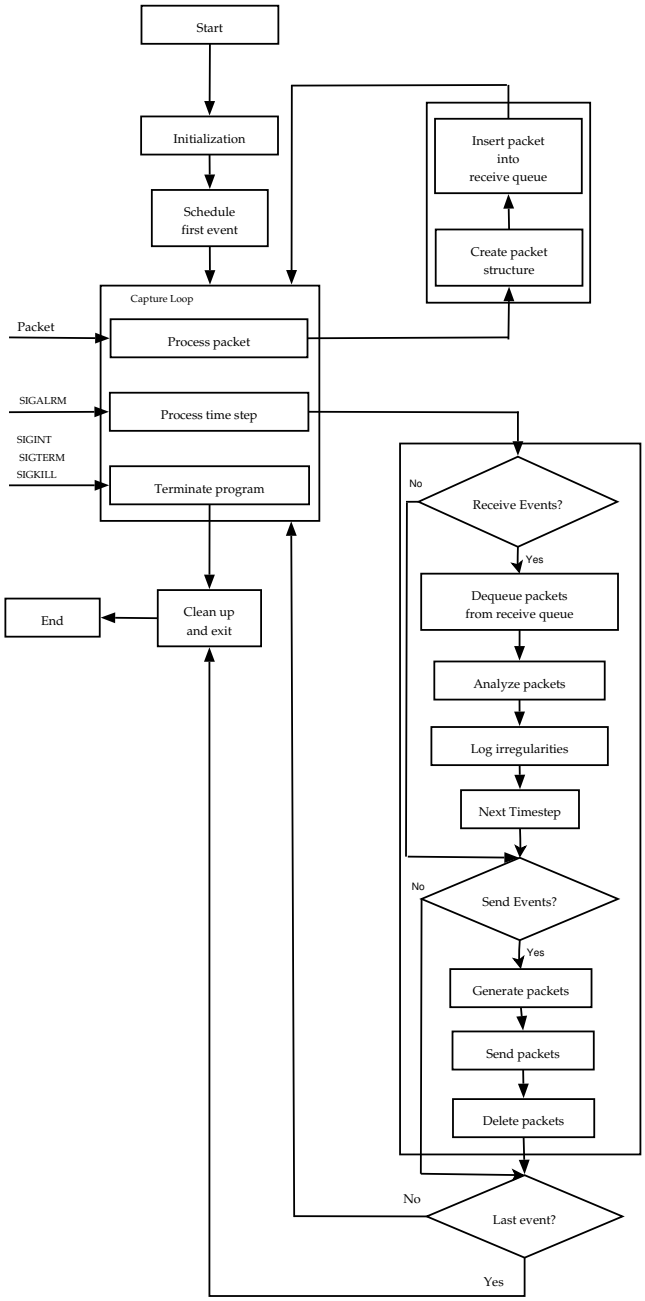

Abbildung 16: Control Flow Graph

3. Implementation

3.1. Lexer

Für den neuen Parser und das neue TP-File-Format ist ein neuer Lexer notwendig. Es müssen Schlüsselwörter erkannt werden, die im alten TP-File nicht vorgekommen sind. Für die Programmierung des Lexers eignet sich der Lexer-Generator `flex`. Aus einer strukturierten Eingabedatei generiert `flex` einen Lexer in C. Die Funktion `int yylex()` wird vom Parser aufgerufen und gibt jeweils das nächste Token zurück, d.h. das nächste Schlüsselwort.

Der aktuelle Text, den der Lexer gerade scannt, steht in der Variablen `yytext`. Diese Variable kann man benützen, um die Daten der gefundenen Schlüsselwörter dem Parser zu übergeben.
In Kapitel 2.14.3 ist die EBNF des neuen TP-File-Formates gegeben.

3.1.1. Vorraussetzungen – Tokens

Im Lexer wird die Header-Datei `tpparser.tab.h` eingebunden[1]. In dieser Datei sind die Tokens definiert. Jedes Token hat einen symbolischen Namen, der mit `SYM_` beginnt. Diesen symbolischen Namen ist je eine eindeutige ganzzahlige Nummer zugewiesen.

3.1.2. Schlüsselwörter

Der Lexer muss ausser den Schlüsselwörtern vor allem ganze Zahlen erkennen können, d.h. Zahlen vom Typ `int` (integer). Die Schlüsselwörter und Datenfelder sind in Tabelle 1 zu finden.

Ausserdem gibt es einige Spezialzeichen, die auch vom Lexer behandelt werden müssen:

- `'\ n'` (Zeilenende) inkrementiert die globale Variable `linenr`. Falls ein Fehler im eingelesenen TP-File ist, kann somit die Zeilennummer, in der der Fehler ist, ausgegeben werden.
- `#`: Diese ganze Zeile des TP-Files ist auskommentiert. Der Lexer lässt alle Zeichen bis zum nächsten Zeilenende aus. Die Variable `linenr` wird inkrementiert, da auch Kommentarzeilen aufgezählt werden müssen. Danach beginnt wieder der normale, nichtauskommentierte Teil des TP-Files.
- `[ #\ t\ n]`: jetzt noch übriggeblienebe Leerzeichen, Tabs, neue Zeilen und Kommentarzeichen werden herausgefiltert.
- . frisst alle bis jetzt noch nicht behandelten Zeichen auf und gibt dem Parser ein SYM_ILLEGAL zurück, da eigentlich keine Zeichen mehr unbehandelt sein sollten.

[1] Sie wird automatisch vom Parser-Generator `bison` erzeugt, indem an der Kommandozeile der Schalter `-d` angegeben wird. Es ist deshalb nötig, dass zuerst der Parser übersetzt wird, damit diese Header-Datei generiert wird und vom Lexer resp. von `flex` benützt werden kann.

Schlüsselwort / Datenfeld	Typ	Rückgabe in
IP-Nummern (SYM_IP)	integer.integer.integer.integer	yylval.text
Port (SYM_PORT)	[a-z0-9-*+/.]*	yylval.text
Variablen (SYM_IDENTIFIER)	[A-Z][a-zA-Z0-9]*	yylval.text
ORIGID (SYM_ORIGID)	integer.integer	yylval.origid .testcase yyl-val.origid.packet
INTEGER (SYM_INT)	integer	yylval.val
Flags (SYM_FLAG)	[SAFRUPsafrup]+	yylval.text
”packet” (SYM_PACKET)		
”testcase” (SYM_TESTCASE)		
’{’ (SYM_BEGIN)		
’}’ (SYM_END)		
”TCP” (SYM_TCP)		
”UDP” (SYM_UDP)		
”ICMPecho” (SYM_ICMPECHO)		
”ICMPunreach” (SYM_ICMPUNREACH)		
”ICMPredir” (SYM_ICMPREDIR)		
”ICMPtexc” (SYM_ICMPTEXC)		
”ICMPtstamp” (SYM_ICMPTSTAMP)		
”send” (SYM_SEND)		
”receive” (SYM_RECEIVE)		
’-’ (SYM_HYPHEN)		
’?’ (SYM_UNKNOWN)		
’.’ (SYM_DOT)		
’;’ (SYM_SEMICOLON)		

Tabelle 1: Erkannte Tokens durch den Lexer

3.1.3. Rückgabetypen

Da die Daten der Schlüsselwörter verschiedenen Typs sind, muss eine einfache Rückgabe-Datenstruktur im Lexer und im Parser definiert werden. Der Lexer gibt für Zahlenwerte einen `int`-Typ zurück. Für die Felder `IP` und `FLAGS` muss der Lexer den Typ String zurückgeben können. Da der längste String durch die Länge der längsten IP-Nummer auf 16 Zeichen begrenzt ist, muss die Datenstruktur für die Rückgabe den Typ `char[16]` definieren. Die `ORIGID`, die für `ICMP` benützt wird, besteht aus zwei `u_long` Variablen. Deshalb bietet sich eine Struct für diese zwei Variablen an. Insgesamt kommt man somit auf folgende Datenstruktur, dargestellt in Abbildung 17.

```
typedef union{
    int val;
    char text[16];
        struct {
                u_long testcase;
                u_long packet;
        } origid;
}ATTRIBUTE;
#define YYSTYPE ATTRIBUTE
```

Abbildung 17: Rückgabe-Datenstruktur

Flex ermittelt den Rückgabetyp anhand des Makros `YYSTYPE`. Dieses Makro muss umdefiniert werden. Der neue Typ soll die Datenstruktur `ATTRIBUTE` sein. Diese Umdefinierung erreicht man, indem man die neue Datenstruktur `ATTRIBUTE` benennt und dann `YYSTYPE` mit `#define YYSTYPE ATTRIBUTE` neu definiert.

Diese Datenstruktur ist global. Der Parser kann also die globale Variable auslesen und erhält somit die vom Lexer übergebenen Werte. Damit der Parser das richtige Feld der Datenstruktur ausliest, muss man für jedes gefundene Token den Rückgabetyp definieren, indem man den Variablennamen der Rückgabedatenstruktur vor das Token schreibt.

3.2. Parser

Aufbauend auf diesem Lexer habe ich ein Parsergerüst und ein Datenstrukturgerüst gemacht. Beat Strasser hat dieses Parsergerüst und die Datenstruktur in seiner Semesterarbeit [Str06] vervollständigt und somit den definitiven neuen Parser für `fwtest` programmiert.

3.3. tpparser.y

3.3.1. involved

Ein Paket gehört neu zum Testfall, falls die Sende-IP-Nummer zu einem der beiden Schnittstelle und die Ziel-IP-Nummer zur anderen Schnittstelle gehört. Die funktion wurde so abgeändert, dass diese Kriterien eingehalten werden.

3.4. Änderungen im bestehenden fwtest-Code

3.4.1. Übergabe von Parametern statt Benützung von globalen Variabeln

fwtest hatte Funktionen, denen globale Variabeln als Parameter übergeben wurden und andere Variabeln wirklich als globale Variabeln in der Funktion benutzt wurden. Eine Funktion, die auf globalen Variabeln arbeitet, ist unflexibel. Es ist nicht möglich, eine zweite Netzwerkschnittstelle in Betrieb zu nehmen, wenn die Funktionen auf globalen Variabeln arbeiten. Deshalb mussten einige Funktionen so abgeändert werden, dass sie nur noch auf übergebenen Parametern arbeiten. Somit können die Funktionen auf den pro Schnittstelle übergebenen Werten arbeiten und es ist kein Problem, eine neue Schnittstelle hinzuzufügen.

3.5. main.h

Die Struktur `_progvars` wurde um die Variabeln

```
char * iface2;
pcap_t * p2;
uint32_t ip2;
struct libnet_ether_addr mac2;
uint32_t localnet2;
uint32_t netmask2;
uint32_t gw2;
struct libnet_ether_addr *hwaddr2;
```

für die zweite Schnittstelle und um

```
pcap_t * p_any;
```

für das "any"-Device erweitert. Ausserdem wird eine neue Variable

```
tnode * timeslots;
```

für die neue Datenstruktur benützt. Diese Variable zeigt auf den ersten Zeitpunkt und wird während der Programmausführung nicht verändert In `main.c` gibt es fünf neue Funktionen. Die Prototypen dieser neuen Funktionen sind in `main.h` dazugekommen. Es sind:

```
void schedule_cleanup();
void erase_events();
void erase_packetinfo();
int cmp_tcp_udp_pkts(event*e,packet*p);
int cmp_icmp_pkts(event*e,packet*p);
```

Die ersten zwei Funktionen waren im alten Parser. Jetzt sind sie neu geschrieben in `main.c`. Die Funktion `erase_packetinfo` ist neu, weil die Datenstruktur ein neu eine Struktur `packetinfo` definiert, die auch gelöscht werden muss. In `packetinfo` können die IP-Nummern oder die Variablen davon und alle Felder **eines** Protokolles gespeichert werden. Die Struktur sieht so aus:

```
typedef struct _packetinfo {
  uint32_t srcip, dstip;
  variable *srcipvar, *dstipvar;
  union {
    tcpflds             *tcp;
    udpflds             *udp;
    icmpechoflds        *icmpecho;
    icmpunreachflds     *icmpunreach;
    icmpredirflds       *icmpredir;
    icmptexcflds        *icmptexc;
    icmptstampflds      *icmptstamp;
  } p;
} packetinfo;
```

Neu werden die Pakete über ihre ID verglichen. Darum sind zwei neue Funktionen, eine für TCP- und UDP-Pakete und eine für alle ICMP-Pakete, dazugekommen. Die Prototypen stehen in `main.h`.
Die neuen Variablen

```
char use_pids;
char use_tcp;
```

werden verwendet, um anzugeben, wo die ID's mitgesendet werden sollen und ob für TCPUDP das TCP- oder das UDP-Protokoll verwendet werden soll.

3.6. main.c

Im der Datei `main.c` mussten einige Funktionen angepasst oder neu geschrieben werden. Da die neue Datenstruktur andere Felder definiert, mussten fast alle Funktionen auf diese Datenstruktur angepasst werden. Die Hauptsächliche Anpassung ist das ändern der Variablennamen von structs.

3.6.1. main

In der Funktion `main` müssen drei neue Variabeln der progrvars initialisiert werden. Es sind die Variablen `pv.use_pids`, `pv.use_tcp` und `pv.timeslots`. Die ersten beiden können durch eine Kommandozeilenoption gesetzt werden. Die letzte wird auf `NULL` initialisiert, um eine leere Liste anzuzeigen.

Neu wird eine zweite Netzwerkschnittstelle für das Senden und das Empfangen benützt. Diese Schnittstelle wird in der Funktion `main` mit dem Aufruf von `if_init` initialisiert. Es müssen neu alle relevanten Parameter der Funktion übergeben werden, da globale Variabeln wie schon erwähnt für zwei Schnittstelle nicht geeignet sind, weil die Funktion dann nicht entscheiden kann, für welche Schnittstelle sie aufgerufen wurde und welche Variabeln sie jetzt benützen soll. Mit der Übergabe der Parameter sind die Funktionen sehr flexibel.

Um auch die MAC-Adressen der Hosts im zweiten Netzwerk herauszufinden, ruft `main` die Funktion `lnet_init` auch für die zweite Schnittstelle auf. Auch dieser Funktion werden alle relevanten Variabeln als Parameter übergeben.

Die Funktion `main` öffnet das TP-File und ruft den Parser mit dem geöffnete Input-Stream auf. Das TP-File wird geparst und kann dann wieder geschlossen werden. Die Datenstruktur

steht jetzt ganz. Es fehlt nur noch das schrittweise Generieren der zu sendenden Pakete, jeweils für den aktuellen Zeitpunkt (siehe 3.6.3).

`lnet_build_pkts` wird in der Funktion `main` nicht mehr aufgerufen, da nicht mehr alle Pakete, wie bis anhin, bereits am Anfang generiert werden.

Die zweite Schnittstelle soll auch zum Sniffen benützt werden. `main` initialisiert also auch die zweite Schnittstelle mit `lpcap_init`. Alle relevanten Variabeln werden der Funktion neu übergeben.

Das Sniffen war bisher so, dass von der einzigen Schnittstelle im blockierenden Modus ständig Daten eingelesen wurden. Mit zwei Schnittstellen ist das nicht mehr möglich. Es muss von beiden Schnittstellen abwechslungsweise gelesen werden können, resp. von der Schnittstelle, die gerade verfügbare Daten hat. Somit geht ein aufruf von `pcap_loop` im blockierenden Modus nicht mehr. `lpcap_init` setzt neu den `nonblock`-Modus auf die Schnittstelle. Im main wird `pcap_loop` durch eine nie terminierende while-Schleife ersetzt. Innerhalb diese while-Schleife kann zweimal `pcap_dispatch` aufgerufen werden, einmal für jede Schnittstelle. So kann von beiden gelesen werden, keine blockiert. Da so die while-Schleife permanent ausgeführt würde und fwtest den Prozessor zu quasi 100 % belasten, würde, wird mit `lpcap_init` das Pseudodevice "any" im **blockierenden** Modus geöffnet. Dieses Device snifft auf allen Schnittstellen gleichzeitig. So wird erreicht, dass die while-Schleife blockiert bleibt, solange keine Daten auf keiner Schnittstelle vorhanden sind. Sobald aber auf einer Schnittstelle Daten vorhanden sind, ist die Blockierung aufgehoben und die while-Schleife setzt fort. Die entsprechende Schnittstelle, die die Daten zur Verfügung hat, gibt sie dann durch den Aufruf von `pcap_dispatch` weiter.

3.6.2. schedule_event

Die Funktion `schedule_event` muss den Timer neu nur noch auf den Timeout-Wert, der in `main.h` definiert ist, setzen. Es müssen keine Zeiten mehr ausgerechnet werden, um den Timer auf den richtigen Wert setzen zu können. Diese Funktion ruft für den ersten Zeitpunkt die Funktion `lnet_build_pkts` auf, damit die Pakete für diesen Zeitpunkt generiert werden. Diese Pakete werden mit `process_send_events` gesendet. Danach wird der Timer gestartet.
Jeder weitere Zeipunkt wird in `process_timestep` abgearbeitet.

3.6.3. process_timestep

Die Funktion `process_timestep` ruft als erstes die Funktion `process_recv_events` auf, um die angekommenen Pakete zu prüfen. Die Funktion `process_timestep` bearbeitet somit die Pakete, die im letzten Zeitpunkt gesendet wurden. Somit ist gewährleistet, dass das Timeout für die zuvor gesendeten Pakete abgelaufen ist. `schedule_event` wird nicht mehr aufgerufen, da der Timer mit einem festen Intervall weiterläuft und keine Zeiten mehr berechnet werden müssen.

Nach dem Abarbeiten der Receive-Events kann zum nächsten Zeitpunkt übergegangen werden. Falls es keinen nächsten Zeitpunkt mehr gibt, sind alle Pakete gesendet worden und fwtest kann beenden.

Die Pakete, die im nächsten Zeitpunkt gesendet werden sollen, werden mit dem aufruf von `lnet_build_pkts` für diesen Zeitpunkt generiert. Jetzt kann `process_send_events` aufgerufen werden, um die Pakete für den jetzigen neuen Zeitpunkt zu senden. Danach werden die von `lnet_build_pkts` generierten Pakete mit `lnet_erase_pkts` wieder gelöscht.

`recv` und `send` müssen neu auf das gleiche Element zeigen, weil neu die Send- und Receive-Events in der gleichen Datenstruktur sind. Beide zeigen somit auf `pv.timestep->testcases`.

3.6.4. cmp_<proto>_pkts

Alle diese funktionen mussten der neuen Datenstruktur angepasst werden. Die gemachten Anpassungen sind hauptsächlich Änderungen der Variablennamen der structs.

3.6.5. erase_packetinfo

Dies ist eine neue Funktion, die die Struktur `packetinfo` löscht und von `erase_events` aufgerufen wird.

3.6.6. erase_events

Diese Funktion ist neu programmiert und jetzt in `main.c` (vorher war sie im alten Parser). Sie löscht alle Events und ruft `erase_packetinfo` auf. Aufgerufen wird sie von `schedule_cleanup`.

3.6.7. schedule_cleanup

Neu ist diese funktion in `main.c`, da sie vorher im alten Parser war. Sie ist neu programmiert, angepasst auf die neue Datenstruktur. Um die Events für jeden Zeitpunkt zu löschen, wird `erase_events` aufgerufen.

3.6.8. cmp_tcp_udp_pkts

Neu werden die Pakete über die ID verglichen, statt über die IP- und Port-Nummern. Diese Funktion vergleicht ein angekommenes Paket mit dem entsprechenden Receive-Event des Paketes.
Es gibt zwei Möglichkeiten, wie die ID im Paket mitgeschickt werden:

- im IP-ID-Feld
- als Daten

Anhand der gewählten Methode vergleicht diese Funktion die erwartete ID entweder mit dem IP-ID-Feld des empfangenen Paketes oder mit dem Datenfeld des Paketes. Falls die ID's gleich sind, wird 1 zurückgegeben, sonst 0.

3.6.9. cmp_icmp_pkts

Diese Funktion vergleicht ein ankommendes ICMP-Paket mit dem dazugehörigen Receive-Event. Für den Vergleich wird die Paket-ID verwendet. Für ICMP ist nur eine IP-ID möglich, da nicht einfach beliebige Daten ans Paket angehängt werden können. Falls vom Benützer Payload-ID gewählt wurde, wird eine Warnung ausgegeben, dass das ICMP-Paket trotzdem über die IP-ID verglichen wird. Falls die ID's gleich sind, wird 1 zurückgegeben, sonts 0.

3.7. lnet.h

Einige Prototypen mussten angepasst werden, damit den Funktionen die nötigen Variablen übergeben werden können, statt dass sie auf globale Variablen zugreifen. Die Funktionen, die eine neue Signatur haben, sind die Folgenden:

- `lnet_init`
- `lnet_scan_net`
- `lnet_print_table`
- `lnet_get_hwaddr`
- `lnet_build_pkts`
- `lnet_build_ether`

Neu dazugekommen ist der Prototyp der Funktion `find_event`. Diese Funktion war vorher im alten Parser und ist jetzt in `lnet.c`.

Eine ganz neue Funktion ist `lnet_erase_pkts`. Der Prototyp dieser funktion ist auch neu in `lnet.h`.

3.8. lnet.c

In `lnet.c` mussten einige Funktionen der neuen Datenstruktur angepasst werden, weil es gewisse Felder nicht mehr gibt und andere Felder in einer neuen Struktur gespeichert sind.

3.8.1. lnet_init

Die Signatur der Funktion `lnet_init` musste geändert werden. Neu werden alle Variablen, die für die Schnittstelle initialisiert werden sollen, als Parameter übergeben.

Die Funktion ist so geändert, dass sie alle Werte weiter an `lnet_scan_net` und an `lnet_print_table` übergibt.

3.8.2. lnet_scan_net

Diese Funktion musste abgeändert werden, damit sie nicht mehr auf die globalen Variablen der bis jetzt einzigen Schnittstelle zugfreift, sondern, dass die Variablen als Parameter übergeben werden können. Somit ist die Funktion flexibel und für mehrere Schnittstellen brauchbar.

3.8.3. lnet_print_table

Diese Funktion erhält jetzt neu auch alle relevanten Variabeln als Parameter übergeben. Sie greift nicht mehr auf globale Variablen zum, damit sie die Tabelle von verschiedenen Schnittstellen anzeigen kann.

Es gibt auch hier eine neue lokale Variable wie in 3.8.2. Die Signatur ist angepasst worden.

3.8.4. lnet_build_pkts

Die Funktion `lnet_build_pkts` ist neu. Sie erstellt die Pakete nur noch für den übergebenen Zeitpunkt, statt wie früher alle Pakete. Somit wird weniger Speicher auf einmal benötigt. Da jetzt zwei Schnittstellen benützt werden, muss die Funktion entscheiden, auf welcher der beiden Schnittstelle das Paket gesendet werden muss.

Der Entscheid, auf welcher Schnittstelle das Paket gesendet werden muss, basiert auf dem Vergleich der und-Verknüpfung der Sende-IP-Nummer mit der Netzwerkmaske und dem Netzwerk. Falls die und-Verknüpfung mit der ersten Netzwerkmaske gleich dem ersten Netzwerk ist, muss das Paket über die erste Schnittstelle gesendet werden. Falls nicht, muss das Paket über die zweite Schnittstelle gesendet werden. Diese Entscheidung muss für jedes Paket einzeln gemacht werden. In der Datenstruktur gibt es für jedes Sende-Event einen Zeiger `libnet_t*l`. Dieser Zeiger benützt, um den Rückgabewert des Aufrufes von `libnet_init` zu speichern. Damit wird dann auch das Paket generiert. Die Funktion ruft je nach Protokoll, das das Paket verwenden soll, die entsprechende `libnet_build_<proto>`-Funktion auf.

Für die Paketgenerierung werden auch die Variablenwerte benützt. Sie müssen bei der Generierung bereits zugewiesen werden. Falls der Wert noch nicht zugewiesen ist, wird das Paket nicht generiert und es wird eine Warnung ausgegeben. Sobald die Werte eingesetzt sind, kann mit diesen Werten nochmals entschieden werden, ob das Paket überhaupt involviert ist. Schliesslich kann das Paket generiert werden. Sobald das Paket generiert ist, geht die Funktion zum nächsten Send-Event des aktuellen Zeitpunktes.

3.8.5. lnet_build_ether

Diese Funktion ist so umprogrammiert, dass sie keine globalen Variablen mehr benützt, sondern dass man ihr die relevanten Variablen übergeben kann. So ist sie Schnittstellunabhängig und kann für beliebige Schnittstellen benützt werden.

3.8.6. lnet_get_hwaddr

Damit die Funktion flexibel benutzbar wird, sind die Zugriffe auf die lokalen Variablen entfernt worden. Neu werden alle nötigen Variablen als Parameter der Funktion übergeben. Das Problem des doppelten Zeigers ist wie in 3.8.2 gelöst worden.

3.8.7. lnet_cleanup

Für die neue Datenstruktur ist auch eine neue cleanup-Funktion nötig. Deshalb ist diese Funktion ganz neu programmiert. Sie iteriert durch die Liste aller Zeitpunkte in der äusseren Schleife. In der inneren Schleife iteriert sie über die Events. Sie löscht von jedem Event die von `libnet` allozierten Datenstrukturen, indem sie `libnet_destroy(e->l);` aufruft.

3.8.8. find_event

`find_event` war vorher im alten Parser. Jetzt ist sie hier neu programmiert, angepasst auf die neue Datenstruktur. Sie iteriert in der äusseren Schleife über alle Zeitpunkt und in der inneren über alle Events des aktuellen Zeitpunktes der äusseren Schleife. Da es keine Unterscheidung zwischen Sende- und empfangs-Events mehr gibt, da das Senden und das Empfangen im gleichen Events ist, ist die Variable `int sendevent` unbenützt.

3.8.9. lnet_erase_pkts

Nach jedem Senden der Pakete des aktuellen Zeitpunktes müssen die von `lnet_build_pkts` generierten Pakete wieder gelöscht werden. So ist der Speicherbedarf von fwtest während der ganzen Ausführung relativ klein. Somit können sehr grosse Testläufe mit sehr vielen Paketen ausgeführt werden.

Die Funktion iteriert durch alle Sende-Events des übergebenen Zeitpunktes und löscht die von `libnet` erzeugten Datenstrukturen.

3.9. util.h

Die Prototypen der Funktionen `parse_netmask` und `if_init` sind neu. Es sind einige Parameter dazugekommen, damit die Funktionen nicht mehr auf die globalen Variablen direkt zugreifen müssen.

3.10. util.c

Einige Funktionen in dieser Datein mussten für die zweite Schnittstelle angepasst werden.

3.10.1. parse_args

`parse_args` hat zwei neue Optionen:

- `-j` für die Angabe der zweiten Schnittstelle
- `-m` für die Angabe der zweiten Netzwerkmaske

Für die Option `-j` wird der Name der zweiten Schnittstelle eingelesen.

Eine zweite Schnittstelle braucht auch ein zweites lokales Netzwerk und eine zweite Netzwerkmaske. Mit der Option `-m` können die zweite Netzwerknummer und die zweite Netzwerkmaske angegeben werden.

`parse_netmask` wird mit den für `-m` angegebenen Werten aufgerufen, um diese Werte für die zweite Schnittstelle zu initialisieren. Falls keine zweite Schnittstelle angegeben wird, versucht `parse_args` das Netzwerk und die Netzwerkmaske für die Schnittstelle herauszufinden. Die Informationen über die zweite Schnittstelle und ihr lokales Netzwerk und ihre Netzwerkmaske werden schliesslich auch ausgegeben.

3.10.2. parse_netmask

Die Signatur musste geändert werden. Die Netzwerkmakse und das lokale Netzwerk werden neu auch als Parameter übergeben, statt als globale Variablen benützt. So ist es möglich, dass mehrere Netzwerkmasken geparst werden können und somit mehrere Schnittstellen benützt werden können.

3.10.3. if_init

Damit beliebig viele Schnittstellen initialisiert werden können, wurde die Funktion so abgeändert, dass die Variablen als Parameter übergeben werden können. Die Funktion hat also eine neue Signatur. Die Variable für das Gateway wird jetzt auch übergeben. Somit kann es für beliebige Schnittstellen herausgefunden werden.

3.10.4. usage

`usage` zeigt jetzt auch, wie man mit den Kommandozeilenparametern `-j` und `-m` die zweite Schnittstelle benützen kann.

3.11. lpcap.h

Von den folgenden Funktionen ist die Signatur neu:

- `lpcap_init`

Es gibt eine neue Funktion in `lpcap.c`. Deshalb ist der Prototyp der Funktion auch in `lpcap.h` dazugekommen. Die neue Funktion ist

```
void lpcap_capture_dummy(u_char*args,const struct pcap_pkthdr*header,
 const u_char*pt);
```

3.12. lpcap.c

3.12.1. lpcap_init

Da neu auch von zwei Schnittstellen gesnifft werden soll, muss die Funktion `lpcap_init` fähig sein, beliebige Schnittstellen initialisieren zu können. Deshalb darf sie nicht auf globale Variablen, die die Schnittstelle betreffen, zugreifen. Besser ist eine Übergabe der relevanten Variablen als Parameter der Funktion.

Die beiden effektiven Schnittstellen haben den Datalink-Typ `DLT_EN10MB`. Die "any"-Schnittstelle hat allerdings den Datalink-Typ `DLT_LINUX_SLL`. Da die Benützung der "any"-Schnittstelle unbedingt nötig ist, muss auch dieser Datalink-Typ erlaubt sein.

Sobald alles initialisiert ist, wird für die zu öffnende Schnittstelle ausser für die "any"-Schnittstelle der nicht-blockierende Modus eingestellt.

3.12.2. lpcap_capture_dummy

Dies ist eine neue Funktion in `lpcap.c`. Jeder Aufruf von `pcap_dispatch` erwartet eine Callback-Funktion als Parameter. Diese Funktion wird aufgerufen, falls ein Paket vorhanden ist. Für die beiden effektiven Schnittstellen wird die Funktion `lpcap_capture` mitgegeben. Auch für die "any"-Schnittstelle muss eine solche Funktion an `pcap_dispatch` übergeben werden. Da die effektive Schnittstelle und die "any"-Schnittstelle jeweils das gleiche Paket empfangen, darf die "any"-Schnittstelle nicht die gleiche Callback-Funktion wie die effektive Schnittstelle benützen, sonst würde jedes empfangene Paket zweimal verarbeitet werden. Deshalb ist diese neue Funktion `lpcap_capture_dummy` definiert. Sie wird für die "any"-Schnittstelle benützt. Sie macht nichts, da das von der "any"-Schnittstelle empfangene Paket nicht weiter verarbeitet werden muss. Es wird dann von der effektiven Schnittstelle verarbeitet.

Eine weitere Möglichkeit wäre, dass die "any"-Schnittstelle alle Pakete verarbeitet, aber die effektiven Schnittstellen nicht. Das ist aber keine gute Idee, denn so würden überhaupt alle Pakete, auch von dritten Schnittstellen, die nicht von fwtest benützt werden, verarbeitet werden.

3.12.3. lpcap_capture

Diese Funktion musste so abgeändert werden, dass sie auf beiden Schnittstellen testet, ob das gerade von einer Schnittstelle gesniffte Paket auf dieser gesendet wurde. Falls das der Fall ist, wird das Paket ignoriert.

3.12.4. lpcap_handle_arp

Damit diese Funktion für zwei Schnittstellen benützt werden kann, musste der Aufruf von `lnet_get_hwaddr` für die zweite Schnittstelle hinzugefügt werden. Anhand der empfangenen IP-Nummer ist klar, auf welche Schnittstelle sich das Paket bezieht. Somit kann `lnet_get_hwaddr` mit den Werten für diese Schnittstelle aufgerufen werden.

3.12.5. lpcap_arp_reply

Diese Funktion muss auf der richtigen Schnittstelle des ARP-Reply-Paket senden. Anhand des erhaltenen Paketes kann die Schnittstelle herausgefunden werden. Es gibt eine neue lokale Variable für den Namen der Schnittstelle, die benützt werden soll. Sie wird auf die entsprechende Schnittstelle initialisiert.

3.13. log.c

3.13.1. log_false_neg

Eine kleine Anpassung an die neue Datenstruktur war für diese Funktion nötig.

3.13.2. log_false_pos

Eine kleine Anpassung an die neue Datenstruktur war für diese Funktion nötig.

3.14. symboltable.h

In `symboltable.h` sind zwei Datenstrukturen definiert. Die Hauptdatenstruktur ist eine Baumstruktur, die die verschiedenen Scopes beinhaltet. Der Schlüssel ist die der Scope, also die Testfallnummer.

Die zweite Datenstruktur ist eine lineare, einfach verkettete Liste, die die Variablen enthaltet. Für jede Variable wird der Name, der zugwiesene Wert und eine Variablennummer gespeichert. Somit kann auf Variablen sowohl über ihren Namen wie auch über ihre Nummer zugegriffen werden. In jedem Knoten des Baumes gibt es eine solche lineare Liste, die alle Variablen in diesem Scope speichern.

Die Symboltabelle kann mit einer Funktion vollständig gelöscht werden.

Definiert sind auch die Prototypen der Funktionen der Symboltabelle. Es gibt zwei Kategorien von Funktionen:

- Funktionen, mit denen auf die Symboltabelle zugegriffen werden kann
- Funktionen, die intern benützt werden.

3.15. symboltable.c

Mit den ersten sechs Funktionen kann auf die Symboltabelle zugegriffen werden. Die weiteren sechs Funktionen werden intern benützt.

3.15.1. sym_init

Diese Funktion initialisiert eine neue Symboltabelle.

3.15.2. sym_search

Diese Funktion sucht eine bestimmte Variable anhand ihrer Variablennummer und dem angegebenen Testcase. Falls die Variable existiert, wird ein Zeiger auf das Listenelement, das diese Variable beschreibt, zurückgegeben. Mit diesem Zeiger kann auf die Felder der Variablendefinition zugegriffen werden. Falls die Variable nicht exisitert, wird der Null-Zeiger zurückgegeben.

3.15.3. sym_search_byname

Mit dieser Funktion kann nach einer Variable durch ihren Namen und der Testfallnummer gesucht werden. Falls sie existiert, wird ein Zeiger auf das Listenelement, das sie beschreibt, zurückgegeben, ansonsten wird der Null-Zeiger zurückgegeben.

3.15.4. sym_insert

Diese Funktion fügt eine neue Variable in die Symboltabelle ein. Falls im angegebenen Testfalls die Variable mit der angegebenen Nummer schon existiert, wird keine neue Variable kreiert, sondern es wird der Zeiger auf das Listenelement zurückgegeben, das die Variable beschreibt.

Falls die Variable mit der angegebenen Nummer im angegebenen Testfall noch nicht existiert, wird ein neues Listenelement kreiert und in den Baum im Knoten des angegebenen Testfalles eingefügt. Falls der Knoten der angegebenen Testfallnummer noch nicht exisitert, wird ein neuer Knoten in den Baum eingefügt. Zurückgegeben wird der Zeiger auf das neu kreierte Listenelement.

3.15.5. sym_insert_byname

Diese Funktion fügt eine neue Variable anhand ihres Namens und der Testfallnummer, in der sie definiert ist, in die Symboltabelle ein. Zurückgegeben wird der Zeiger auf das neu kreierte element. Falls die Variable schon existiert, wird der Zeiger auf das entsprechende Listenelement zurückgegeben.

3.15.6. delete_sym_table

Mit dieser Funktion kann die ganze Symboltabelle gelöscht werden. Sie traversiert den Baum in postorder-Reihenfolge und löscht jeweils das linke und rechte Kind, dann die Variablen-Liste des aktuellen Knotens und am Schluss den Knoten selber. Der Zeiger, der übergeben wurde, wird auf 0 gesetzt. Nachdem die ganze Symboltabelle gelöscht ist, hat der Zeiger auf die Symboltabelle also den Wert 0, das heist er ist der Null-Zeiger.

3.15.7. insert_var

Diese Funktion erzeugt ein neues Variablen-Element und initialisiert es. Die Variablennummer wird auf den übergebenen Wert gesetzt. Anschliessend wird das neue Element in die übergebene Variablen-Liste am Schluss der Liste eingefügt. Die Liste muss nicht sortiert sein, da die Variablen unabhängig voneinander sind, das heisst, es gibt unter ihnen keine Reihenfolge. Sie gehören alle zum gleichen Testfall. Zurückgegeben wird der Zeiger auf das neu kreierte Variablen-Element.

Da die Funktionen `sym_insert` und `sym_insert_byname` bereits testen, ob das Variablen-Element existiert, muss das hier nicht nochmals geschehen und es kann davon ausgegangen werden, dass die Variable noch nicht in der Liste ist. Diese Funktion darf deshalb nur intern benützt werden

3.15.8. insert_var_byname

Diese Funktion erzeugt eine neues Variablen-Element und initialisiert das Namen-Feld mit dem übergebenen Namen. Anschliessend wird es in die übergebene Variablen-Liste am Schluss der Liste eingefügt. Zurückgegeben wird der Zeiger auf das neu kreierte Variablen-Element.

Da die Funktionen `sym_insert` und `sym_insert_byname` bereits testen, ob das Variablen-Element existiert, muss das hier nicht nochmals geschehen und es kann davon ausgegangen werden, dass die Variable noch nicht in der Liste ist. Diese Funktion darf deshalb nur intern benützt werden

3.15.9. node_search

Mit `node_search` wird der durch die Testfallnummer angegebene Knoten im übergebenen Baum gesucht. Der übergebene Baum entspricht dem Zeiger auf die Symboltabelle.

Falls der gesuchte Knoten existiert, gibt die Funktion den Zeiger auf den Knoten zurück, anderenfalls gibt sie den Null-Zeiger zurück.
Diese Funktion wird intern benützt.

3.15.10. var_search

Diese Funktion sucht das Variablen-Element anhand der Variablennummer in der übergebenen Variablen-Liste. Falls das element existiert, wird der Zeiger darauf zurückgegeben, anderenfalls wird der Null-Zeiger zurückgegeben.
Diese Funktion wird intern benützt.

3.15.11. var_search_byname

Mit dieser Funktion kann man nach dem Variablen-Element anhand des Variablennamens in der übergebenen Liste suchen. Falls das Element existiert, wird der Zeiger darauf zurückgegeben, anderenfalls der Null-Zeiger.
Diese Funktion wird intern benützt.

3.15.12. insert_node

Die Funktion `insert_node` fügt einen neuen Knoten in den Baum ein. Der Schlüssel ist die übergebene Testfallnummer. Da die Funktionen `sym_insert` und `sym_insert_byname` bereits

testen, ob der Knoten existiert, ist das hier nicht mehr nötig und es kann davon ausgegangen werden, dass der Knoten sicher nicht im Baum ist. Diese Funktion darf deshalb nur intern benützt werden.

3.15.13. delete_variables

Diese Funktion löscht eine ganze Liste von Variablen-Elementen. Sie wird intern benützt, wenn die ganze Symboltabelle gelöscht wird.

3.16. semanticchecker.h

Um nach dem Parsen der TP-Datei sicherzustellen, dass keine Variable als erstes in einer Send-Definition vorkommt, da es dann unmöglich wäre, dass sie einen Wert zugewiesen hat, und um sicherzustellen, dass eine Port-Variable in jedem Paket nur in einem Port-Feld und eine IP-Variable in jedem Paket nur in einem IP-Feld vorkommt, also der Typ der Variable immer gleich ist, braucht es einen Semantic-Checker. Dieser geht durch die ganze Datenstruktur und Testet den Typ jeder Variable und ob es vor dem ersten Gebrauch jeder Variable eine Zuweisung gibt. In `semanticchecker.h` sind die beiden Prototypen der beiden Funktionen des Semantic-Checkers definiert.

3.17. semanticchecker.c

3.17.1. semanticchecker_check

Mit dieser Funktion wird der Semantic-Checker gestartet. Sie ist so aufgebaut, dass sie weitere Funktionen aufrufen kann und jeweils die Rückgabewerte der Unterfunktionen oderverknüpft und zurückgibt.
Sie ruft die einzige Funktion `semanticchecker_check_var_types` auf.

3.17.2. semanticchecker_check_var_types

Diese Funktion iteriert über die ganze Datenstruktur und überprüft für jede Variable, ob sie, vor dem ersten Benützen in einem Send-Event, in einem Receive-Event steht, damit sie vorher zugewiesen werden kann. Ausserdem prüft sie für jede Variable, ob sie in jedem Vorkommnis den gleichen Typ hat. Falls eine der beiden Überprüfungen fehlschlägt, wird eine Fehlernummer zurückgegeben, sonst wird 0 zurückgegeben.

4. Zusammenfassung

4.1. Übersicht

fwtest ist ein Programm, mit dem man Firewalls auf ihre korrekte Funktionsweise testen kann. Urspünglich programmiert wurde es von Gerhard Zaugg [Zau05]. Zur gleichen Zeit, wie ich Änderungen an fwtest gemacht habe, hat Beat Strasser in seiner Semesterarbeit [Str06] fwtest um UDP- und ICMP-Funktionalität erweitert.

4.2. Ziel

Das Ziel war es, die Zeit des alten TP-Files zu eliminieren und fwtest als eine Instanz auf einem Rechner ausführen zu können. Ein zweites Ziel war es, fwtest so zu erweiternt, dass auch Firewalls mit NAT getestet werden können. Diese Änderungen und Ergänzungen bedingten ein neues Design für das TP-File und einen neuen Lexer und einen neuen Parser dazu.

Um fwtest als eine Instanz benützen zu können, musste eine zweite Schnittstelle hinzugefügt werden. Somit mussten viele Funktionen abgeändert werden, weil sie auf globale Variablen der einzigen Schnittstelle, die es gab, zugreifen.

4.3. Design

4.3.1. Sniffen von zwei Schnittstellen

Da **fwtest** jetzt als eine Instanz gestartet wird, muss es eine zweite Schnittstelle unterstützen. Es muss möglich sein, von zwei Schnittstellen gleichzeitig zu sniffen. Das Sniffen von einer Schnittstelle darf also nicht blockierend sein. Die Prozessorauslastung soll aber klein bleiben, wenn keine Daten an den Schnittstellen anstehen.

Mit der Verwendung von der Pseudo-Schnittstelle "any" können beide Schnittstellen im nicht-blockierenden Modus sniffen und die "any"-Schnittstelle blockiert die Ausführung der while-Schleife, solange keine Daten ankommen. "any" muss im blockierenden Modus sniffen.

4.3.2. Timeout

Nicht jedes gesendete Paket kommt an. Deshalb muss ein Timer gestartet werden, der nach einem bestimmten Timeout abläuft. Danach kann fwtest zum nächsten Zeitpunkt über gehen.

4.3.3. NAT

fwtest unterstützt neu auch das Testen von Firewalls mit NAT. Da gewisse Felder von der Firewall geändert werden, wenn NAT benützt wird, müssen Variablen für diese Felder unterstützt werden.

Ein Paket kann neu auch spezifiziert werden, wie man es erwartet, da es nicht gleich aussehen muss, wenn es empfangen wird.

4.3.4. Paket-ID

Um ein empfangenes Paket identifizieren zu können, wird eine eindeutige Nummer mitgesendet. Das ist eine gute Lösung, das Receive-Event des entsprechenden Paketes zu finden und

dann die Variablen zu überprüfen und die Werte mit den erwarteten zu vergleichen.
Für TCP und UDP gibt es zwei Varianten, die ID mitzusenden:

- im IP-ID-Feld
- als Daten

Für ICMP wird die ID immer im IP-ID-Feld mitgesendet. Das ist auch die Standardeinstellung. Der Benützer kann an der Kommandozeile angeben, dass er die ID in den Daten mitsenden will.

4.3.5. Globale Zeit

Die absolute Zeit kann aus dem TP-File eliminiert werden, da fwtest nur noch als eine Instanz gestartet wird und somit keine Synchronisation über die Zeit notwendig ist.

Es gibt eine relative Zeit zwischen den Pakete, die durch die Paketnummer definiert ist. Grössere Paketnummern gehören zu einem späteren Zeitpunkt. Pakete mit gleicher Paketnummer, die zwingend in verschiedenen Testfällen sein müssen, gehören zum gleichen Zeitpunkt.

4.3.6. TP-File

Im TP-File können jetzt Testfälle spezifiziert werden. Zu jedem Testfall gehört mindestens ein Paket. Jeder Testfall hat eine eindeutige Nummer. Das Protokoll wird direkt bei der Paketspezifikation ausgewähtl, es gilt nicht mehr das gleiche Protokoll für jedes Paket in der gleichen Datei. Ein Paket wird spezifiziert, wie es gesendet werden soll und wie es ankommen soll. Es können Variablen für einzelne Felder benützt werden, da das wichtig und nötig für Tests von Firewalls mit NAT ist. Es kann auch angegeben werden, ob ein Paket nicht ankommen darf oder ob es egal ist, dass ein Paket ankommt oder auch nicht ankommt.

Der Aufbau des TP-Files ist übersichtlich und man sieht sofort, in welchem Kontext ein Paket steht, weil man die Zugehörigkeit zu einem Testfall sofort sieht.

4.3.7. Datenstruktur

Die Datenstruktur besteht aus einer Liste mit Zeitpunkten. Für jeden Zeitpunkt gibt es eine Liste mit Events. Jedes zu sendende Paket wird in einem Sendevent und jedes Erwartete Paket in einem Receiveevent beschrieben. Generiert werden die Pakete erst kurz vor dem Senden, damit die zugewiesenen Variablenwerte miteinbezogen werden können. Nach dem Senden wird das generierte Paket wieder gelöscht. Somit braucht fwtest zur Laufzeit wenig Speicher, da nicht alle Pakete von Anfang an generiert werden.

4.4. Implementation

4.4.1. Lexer

Für das neue TP-File war ein neuer Lexer und ein neuer Parser erforderlich. Der Lexer kennt eine Reihe von Schlüsselwörtern und gibt diese als Tokens dem Parser zurück. Einige Schl2usselwörter erwarten einen Wert, der auch dem Parser über eine Rückgabe-Datenstruktur übergeben wird.

4.4.2. main

Die Struktur der Funktion `main` musste angepasst werden. Es arbeitet jetzt auf der neuen Datenstruktur. Der Timer musste so angepasst werden, dass er nur noch mit dem Timeout arbeitet und keine Zeiten mehr ausrechnet.
Für die Analyse der empfangenen Pakete soll zuerst das entsprechende Receive-Event gefunden werden. Dann müssen die Variablen behandelt werden und danach können die Felder auf die Gleichheit mit den erwarteten Werten geprüft werden.

4.4.3. lnet

lnet musste hauptsächlich an die neue Datenstruktur angepasst werden und so umgebaut werden, dass zwei Schnittstellen benützt werden können.

4.4.4. util

Da es neue Kommandozeilenargumente gibt, musste **parse_args** und **usage** erweitert werden, um diese Benützen zu können. Ausserdem mussten ein paar Funktionen abgeändert werden, damit zwei Schnittstellen benützt werden können.

4.4.5. lpcap

Für die neue "any"-Schnittstelle musste eine neue Capture-Funktion hinzugefügt werden. Diese wird zwar nicht benützt und macht deshalb auch nichts. Aber **libpcap** benötigt die Übergabe einer solchen Funktion.

4.4.6. log

Wenige Änderungen mussten gemacht werden, damit das Logging nach dem Umbau von fwtest funktioniert. Ausserdem sind zwei Funktionen dazu gekommen, die die Wertzuweisung von IP- und Port-Variablen loggen, damit man eine Übersicht über die zugwiesenen Variablen hat.

4.4.7. Symboltabelle

Für die Unterstützung von Variablen, die innerhalb eines Testfalles gültig sein sollen, ist eine Symboltabelle erforderlich. Diese Symboltabelle muss eine Art von Scopes zur Verfügung stellen, in denen die Variablen gespeichert werden. Variablen können leicht erzeugt werden, indem man die nötigen Datenstrukturen direkt von der Symboltabelle erzeugen lässt. Man muss nur den Scope, in dem sie erzeugt werden soll, und den Variablennamen angeben und einen Eintrag kreieren lassen. Der Scope ist der Testfall, das heisst die Testfallnummer, in dem die Variable vorkommt.

Mit 5 einfachen Funktionen, die zur Verfügung gestellt werden, ist es einfach, mit Variablen umzugehen.

5. Schlussfolgerung

fwtest ist ein interessantes und starkes Tool, um Firewalls zu testen. Es ist interessant, an einem Programm, das Pakete generiert und ins Netzwerk einschleust und Pakete snifft, zu programmieren. Auch die Zusammenarbeit mit Beat Strasser (siehe seine Semesterarbeit [Str06]) war eine interessante Erfahrung.

Die NAT-Erweiterung von `fwtest` konnte ich gut implementieren. Das neue TP-File-Design, in dem das Paket, auch wie es erwartet wird, spezifiziert werden kann und in dem auch Variablen für die Port- und IP-Nummer verwendet werden können und die Einführung der Paket-ID erlauben eine gute, einfache und saubere Implementation der NAT-Funktionalität. Dieses Ziel wurde somit erreicht. Der Design-Entscheid für die Übertragung der Paket-ID und für das TP-File sind aus jetziger Sicht gut und warfen keine zusätzlichen Probleme auf. Das neue TP-File ist einfach und logisch aufgebaut und somit gut leserlich. Deshalb würde ich das wieder auf diese Weise erweitern.

Da `fwtest` jetzt als eine Instanz gestartet wird, sind die Synchronisationsprobleme behoben und die absolute Zeit konnte eliminiert werden. Dieses Ziel konnte durch die Unterstützung einer zweiten Netzwerkschnittstelle ohne grosse Probleme erreicht werden. Es war jedoch zeitaufwendig, da einige Funktionen für die neue Datenstruktur und für die neue Schnittstelle angepasst werden mussten. Die einzige Schwierigkeit bei der Unterstützung der zweiten Schnittstelle ist das gleichzeitige Sniffen von beiden Schnittstellen, da keine von beiden in einem blockierenden Modus sein darf.

Die Ziele konnte ich somit alle ohne grössere Probleme erreichen. Allerdings waren gewisse Änderungen sehr zeitaufwendig. Gelernt habe ich ganz allgemein, wie man eine Testumgebung programmiert, in welcher man Pakete in ein Netzwerk einschleust und in welcher man das Netzwerk snifft. Ausserdem habe ich gelernt, dass die Einarbeitung in ein bestehendes Programm nicht sehr trivial ist.

6. Ausblick

6.1. Variabler Timeout

Eine sinnvolle mögliche Erweiterung von fwtest ist ein dynamisches Wählen des Timeout-Wertes. Falls fwtest erkennt, dass der Timeout so kurz ist, dass sehr viele Pakete erst nach dem Ablaufen des Timeouts ankommen, kann es den Timeout-Wert selbstständig erhöhen und die Pakete nochmals senden.

Sinnvoll ist auch eine Kommandozeilen-Option, mit der der Benützer einen Timeout-Wert vorgeben kann. Somit bleibt der Wert nicht unveränderbar und fwtest muss für eine Änderung des Wertes nicht neu kompiliert werden.

6.2. Sendewiederholung von verlorenen/zu spät angekommenen Paketen

Falls Pakete nach dem Ablaufen des Timeout-Wertes ankommen, kann fwtest sie nochmals senden. Somit ist die Wahrscheinlichkeit grösser, dass fwtest wirklich feststellen kann, ob ein Paket von der Firewall geblockt wurde oder ob es einfach verloren gegangen ist oder zu spät angekommen ist.

Die maximale Anzahl von Sendewiederholungen muss vom Benützer festgelegt werden können. Falls ein Paket nach allen Sendewiederholungen nicht angekommen ist, kann fwtest davon ausgehen, dass es von der Firewall geblockt wird. Ein gutes Verhältnis zwischen dem Timeout-Wert und der Anzahl Sendewiederholungen muss gewählt werden, damit ein Testlauf nicht zu lange dauert.

Mit der Einführung der Sendewiederholung kommen andere Probleme dazu. Es müssen zum Beispiel ankommenden Duplikate richtig behandelt werden können, denn wenn ein Paket nur verspätet ist und deshalb das Paket nochmals gesendet wird, kann es sein, dass beide Pakete ankommen. Diese beiden Pakete sind aber genau das Gleiche.

Die Datenstrukturen werden komplizierter. fwtest muss in jedem Zeitpunkt die nicht-angekommenen Pakete nochmals senden und für diese Pakete nochmals einen Timeout starten, da das Paket bei der wiederholten Sendung auch nicht ankommen muss. Ausserdem ist ein Zähler für die Anzahl Sendungen jedes Paketes nötig. Bei jeder Sendung wird er inkrementiert. Sobald der Wert gleich der maximalen Anzahl Sendewiederholungen ist, wird das Paket nicht mehr gesendet, auch wenn es noch nicht angekommen ist.

Falls das Paket einige Zeitpunkte später ankommt, muss immernoch das originale event gefunden werden. Dann kann das Paket mit den erwarteten Werten verglichen werden, wie wenn es im erwarteten Zeitpunkt angekommen wäre.

A. Aufgabenstellung

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Information Security
www.infsec.ethz.ch

Semester thesis for Adrian Schüpbach

Firewall Testing with NAT

an extension to fwtest v0.6

Supervisor:	Diana Senn
Professor:	Prof. D. Basin
Issue Date:	October 2005
Submission Date:	February 2006

1 Introduction

We live in a world where all the company networks are connected to the Internet. Nobody can control the Internet, therefore a company has to protect their data from unauthorised access through the Internet. This is done by firewalls whose analogon in the physical world are locks. Everybody understands that doors need to be locked to prevent unauthorised access. It is the same in the digital world: unauthorised access to a companies network should be prevented, and this can be done by one or several firewalls.
Using the analogon of the door lock again, everybody understands that it is not enough to have a door lock. Only if the lock is locked properly and only authorised people have got a key to unlock it, we have what we want. It is the same in the digital world. It is not enough to have a firewall. We can only be satisfied if the firewall is doing what we expect from it. And to find out if a firewall satisfies our expectations (stated by a policy) we need to test it.

2 Motivation

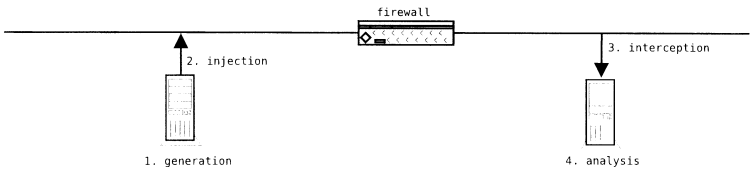

Figure 1: Flow of the Test Packets

Firewall Testing consists of two parts: the theoretical part of finding adequate test cases, and the practical part of running these test cases (each consisting of one or more test packets) on the real system. The aim of this thesis is to cover the second part.

Running test cases on a real system consists of four steps as shown in figure 1. The first step is the generation of the network packets. According to a given test case the corresponding network packets have to be built. The second step consists of injecting the packets generated in step one into the network. The third and the fourth step are then to intercept the packets which were injected in step two behind the firewall, comparing them to the expectations, and logging the result.

3 Assignement

3.1 Objectives

During his diploma thesis [1], Gerhard Zaugg has written a simple tool – named fwtest – which can do the above for TCP packets in a bidirectional way. This tool is a good starting point, but there are some things that need further work. Two of them should be looked at in this thesis: NAT and Timing.
As figure 1 shows there are two parties involved in testing which need some kind of synchronisation for being able to conduct bidirectional tests. For the first version of fwtest we decided to use time (and the NTP protocol) for this purpose. Unfortunately we had to find out in our tests at the end that this does not work very good. So one goal of this semester thesis is to combine these two parties into one, as shown in figure 2, and to find another way (than timing) of specifying test cases, i.e. which test packet belongs to which test case and in which order.
The second goal is to be able to handle NAT, i.e. to give the possibility to the tester (the person using fwtest) to specify every packet twice: how it looks before the firewall, and how he expects it behind the firewall. Also the tester should be able to use variables, which are then instantiated during testing. For example if we know that the firewall will translate every source IP and port to its own IP and some port. Then we just want to set a variable for this port in the specification of the test case to denote that there will be any number but that it has to be the same for the whole test case.

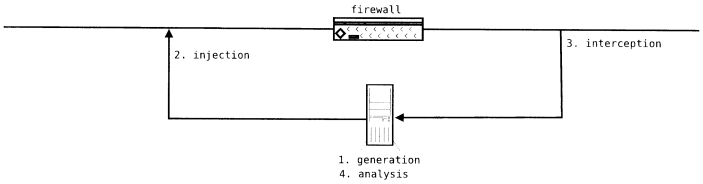

Figure 2: A single instance of fwtest

3.2 Tasks

- Understanding fwtest
- Designing a new generation of tp-file which knows test cases (instead of single test packets) and can cope with changing packets

- Adapting the parser to the new tp-file (no time, 2 versions of every packet, ...)
- Change the packet generation from preset (all packets are generated before testing) to adaptive (packets are generated when needed) to be able to take the newest information into account
- Changing fwtest to run as a single instance (instead of on two machines)

As there will be another student (Beat Strasser) working on fwtest in an overlapping time frame, some synchronisation between the two students is expected. This will mainly consist in determining a new format for the tp-files together and working on the same subversion repository.

3.3 Deliverables

- At the beginning of the semester thesis an agreement must be signed which allows the supervisor of this thesis, his project partners and ETH Zurich to use and distribute the software written during the thesis.
- At the end of the second week, a detailed time schedule of the semester thesis must be given and discussed with the supervisor.
- At the end of the diploma thesis a presentation of 20 minutes must be given during an Infsec group seminar. It should give an overview as well as the most important details of the work.
- The final report may be written in English or German. It must contain an abstract written in both English and German, this assignement and the schedule. It should include an introduction, an analysis of related work, and a complete documentation of all used software tools. Three copies of the final report must be delivered to the supervisor.
- Software and configuration scripts developed during the thesis must be delivered to the supervisor on a CD-ROM.

References

[1] Gerry Zaugg. Firewall testing. http://www.infsec.ethz.ch/people/dsenn/DA_GerryZaugg_05.pdf.

16th August 2005

Prof. D. Basin

B. Zeitplan

1. Woche (Fr, 28.10.2005 - Fr, 04.11.2005):

- Zeitplan erstellen
- Überlegungen, wie TP-File geändert werden kann, damit man definieren kann, wie das gesendete und das empfangene Paket aussehen soll
- Überlegung, wie man die Zeit eliminieren kann, wie man Paketzugehörigkeit speichern kann, wie man erlaubte Gleichzeitigkeit von unabhängigen Paketen beschreibt und wie man Reihenfolge/zeitliche Abhängigkeit im TP-File definieren kann
- Lösungsideen zu diesen Aufgaben finden, Vor- & Nachteile aufschreiben

2. Woche (Fr, 04.11.2005 - Fr, 11.11.2005):

- Spezifikation der TP-Datei gemäss ausgewählter Methode aus Woche 1
- Wie erkennt man, dass ein gesendetes Paket das Gleiche, wie ein empfangenes Paket ist? → zum Beispiel ID als Payload
- Lexer-Datei schreiben
- Parser-Datei schreiben
- Testprogramm, das Datei einliest und parst → testen
- Dokumentation des TP-Datei-Formats schreiben

3. Woche (Fr, 11.11.2005 - Fr, 18.11.2005):

- Parser in fwtest integrieren

4. Woche (Fr, 18.11.2005 - Fr, 25.11.2005):

- Parser fertig integrieren
- Parser in fwtest testen
- Dokumentation schreiben, wie Parser in fwtest integriert ist

5. Woche (Fr, 25.11.2005 - Fr, 02.12.2005):

- Überlegungen, wie man am besten von zwei Schnittstellen Pakete einlesen kann und wie man sie am besten über zwei Schnittstellen senden kann
- Überlegen, wie Event-Struktur angepasst werden muss, um zwei Schnittstellen bedienen zu können
- Überlegen, wie struct's aussehen müssen, um zwei Schnittstellen benützen zu können
- Prozessorzeit beim Sniffen nicht zu sehr beanspruchen → nonblocking & blocking
- Lösungsideen finden und Vor- & Nachteile aufschreiben

6. Woche (Fr, 02.12.2005 - Fr, 09.12.2005):

- neue struct's und neue Event-Struktur spezifizieren
- Dokumentation über neue Event-Struktur und neue struct's schreiben → Spezifikation aufschreiben
- Funktionsweise des Programms/Programmfluss und Funktionsweise der neuen Event-Strukt dokumentieren

7. Woche (Fr, 09.12.2005 - Fr, 16.12.2005):

- struct's anpassen resp. erweitern, möglichst so, dass das Programm während der Anpassung weiter funktioniert
- Zweite Schnittstelle einbeziehen bei der Initialisierung
- Sniffen mit beiden Schnittstellen (wegen libpcap & blocking/nonblocking), auch wenn die neue Schnittstelle noch keine Funktion hat
- cleanup anpassen wegen neuer Schnittstelle und angepassten struct's

8. Woche (Fr, 16.12.2005 - Fr, 23.12.2005):

- Neue Event-Struktur implementieren
- Überlegen, wie und was präsentieren

9. woche (Mo, 09.01.2006 - Fr,13.01.2006):

- Neue Event-Struktur fertig implementieren
- Dokumentation darüber updaten (Probleme bei der Implementation)
- Präsentationsskizze erstellen

10. Woche (Fr, 13.01.2006 - Fr, 20.01.2006):

- Test des Programms, Fehler verbessern
- Präsentaion beginnen

11. Woche (Fr, 20.01.2006 - Fr, 27.01.2006):

- Präsentation fertig schreiben
- Präsentation korrigieren
- Reservezeit für Fehlerkorrektur im Programm

12. Woche (Fr, 27.01.2006 - Fr, 03.02.2006):

- Reservezeit für Fehlerkorrektur im Programm und in der Präsentation

13. woche (Fr, 03.02.2006 - Di, 07.02.2006):

- Präsentation üben

C. README-File

```
================================================================================

    FWTEST v1.0 (February 2006)
    Author:       Gerry Zaugg <zauggge@gmail.com>
    Contributors: Adrian Schuepbach <scadrian@student.ethz.ch>
                  Beat Strasser <b8@student.ethz.ch>
    Maintainer:   Diana Senn <diana.senn@inf.ethz.ch>

================================================================================

0.  Content

    1. Description
    2. Software Requirements
    3. How to Perform a Test Run
       3.1 Test Environment
       3.2 Mandatory Files
       3.3 run_fwtest.sh
       3.4 Test Packets File
           3.4.1 General layout
           3.4.2 Protocol specific fields (PROT_FIELDS)
           3.4.3 Preprocessor
           3.4.4 Variables
           3.4.5 Parse errors
       3.5 Running Fwtest
       3.6 Example
    4. Timeout
    5. Source Files



1.  Description

Fwtest v1.0 provides a simple, portable interface to perform firewall testing.
It is executed on a so-called testing host that sends test packets through a
firewall. Fwtest crafts and injects the test packets and captures them if they
pass the firewall. When receiving packets, fwtest performs an exhaustive
analysis revealing irregularities (i.e. packets that are accepted by the
firewall although they were expected to be dropped or packets that are
discarded/changed although they were expected to be passed). The irregularities
are logged and serve as source of information to identify failures.

Fwtest v1.0 was a further development of fwtest v0.5 which evolved in the
context of a Diploma Thesis by Gerry Zaugg. Besides an extension to UDP and
ICMP, the new version has experienced some underlying restructuring in order to
```

fit for NAT. For more information, read the corresponding documentation(s):

[1] Gerry Zaugg. Firewall Testing. January 2005.
http://www.infsec.ethz.ch/people/dsenn/DA_GerryZaugg_05.pdf
[2] Adrian Schuepbach. Testen von Firewalls mit NAT. February 2006.
http://www.infsec.ethz.ch/people/dsenn/SA_AdrianSchuepbach_06.pdf
[3] Beat Strasser. Eine UDP-/ICMP-Erweiterung fuer fwtest. February 2006.
http://www.infsec.ethz.ch/people/dsenn/SA_BeatStrasser_06.pdf

2. Software Requirements

Fwtest v1.0 is known to compile and run under Linux Debian 3.0 with the 2.4.24 kernel, Linux Red Hat 7.3 with the 2.4.18-3 kernel and Gentoo Linux 2005.1 with the 2.6.14 kernel. Probably, it also runs under OpenBSD, FreeBSD and NetBSD.

You will need flex, bison, gcc, make, libc and libc-dev to build fwtest. We make use of m4 and the administration tools iptables or ipchains.

We need the following libraries:
* libpcap-0.7.2
* libnet-1.1.2.1
* libdnet-1.8
* libc

Libpcap, libnet and libdnet have to be installed from source since we use the header files (pcap.h, libnet.h, dnet.h).

NOTE: Run /sbin/ldconfig before using libdnet. ldconfig creates the necessary links and cache (for use by the run-time linker, ld.so) to the most recent shared libraries (for more information: man ldconfig).

3. How To Perform a Test Run

3.1 Test Environment

This section gives an overview on how a firewall test is performed with fwtest.

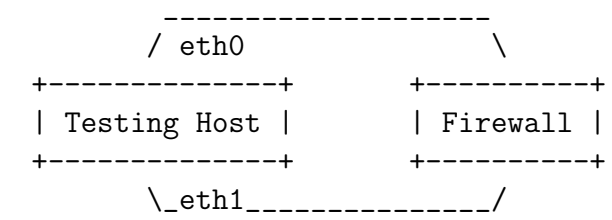

Figure 1: Sample test environment

We assume that you have set up a test environment similar to the one illustrated in figure 1: A testing host is connected to a firewall via two network devices. Fwtest will run on the testing host and craft, inject, capture and analyze packets as well as log irregularities.
You have to provide a test packet file holding the specifications of the test cases (consisting of several test packets) that fwtest will generate, inject and capture. The layout of the this file is explained in section 3.4.

3.2 Mandatory Files

We now discuss how to run fwtest on a testing host.

Copy the following files into the test directory:

(1) Source files: Listed in the last chapter of this file.
(2) run_fwtest.sh: Fundamental shell script performing the test run.

3.3 run_fwtest.sh

run_fwtest.sh leads you through the process of firewall testing.
You have to be root when executing the script. Furthermore, you must provide some arguments when starting the script. The test run will then be performed automatically.

run_fwtest.sh requires at least six arguments:
(1) Test packets file
    File containing the test packets
    Read more about the file format below.
(2) Log file
    File where the irregularities are reported
(3) Network 1 and mask in CIDR notation
    Specifies the first network that the testing host represents.
    (e.g. 192.168.1.0/29)
(4) Network 1 interface
    The network interface to capture the packets from network 1 (e.g. eth0)
(5) Network 2 and mask in CIDR notation
    Specifies the second network that the testing host represents.
    (e.g. 172.16.70.0/29)
(6) Network 2 interface
    The network interface to capture the packets from network 2 (e.g. eth1)
(7) Option -p
    Transmit packet id in payload (TCP/UDP only*)

```
(8) Option -u
    Interpret TCPUDP packets as UDP instead of TCP

* ICMP packets usually don't allow to send a payload. The option -p covers only
TCP and UDP packets; the ICMP packet id is always sent in the IP header. Note
that the IP identification field is only 16 bit - if you deal with more than
2^16 packets including ICMP packets and together with the payload option, you
will certainly run into problems: statistics may be misinterpretable, so be
warned.

Before fwtest can be called, you must make sure that the gateway's MAC address
is contained in the local system's ARP cache. To achieve this, you may ping the
gateway on each device while the firewall/gateway is (still) accepting ICMP
packets:

ping -c 1 192.168.72.1
ping -c 1 172.16.70.1

Example how run_fwtest.sh may be called:

./run_fwtest test1 test1.log 192.168.72.0/29 eth0 172.16.70.0/29 eth1 -u

run_fwtest.sh executes the following steps:
(1) Check the arguments for validity.
    Exit if one of them is bad or not specified.
(2) Seek for the required programs, libraries and header files.
    Exit if one of them is missing.
(3) Compile fwtest if necessary.
(4) Set up local firewall rules to drop all incoming packets so that the testing
    host does not response to the packets it captures. This can only be done
    if either iptables or ipchains is installed on your system.
(5) Run fwtest.
    - Run a preprocessor on the packets definition file.
    - Testing is performed (i.e. packets are crafted, injected,
      captured, analyzed) and the irregularities are logged.
    - Fwtest will print some useful information, especially warnings and errors.

    To make this point clear: fwtest is the firewall testing tool that
    performs testing whereas run_fwtest.sh is only a shell script that
    prepares the testing host for the test run and compiles and runs fwtest
    for you. Read more about invoking fwtest in section 3.5.
(6) Remove the local firewall rules.
(7) Exit successfully.

The irregularities are stored in the log file. Evaluate them to identify
problems and abnormalities.
```

3.4 Test Packets File

3.4.1 General layout

The general layout of a file containing test packet specifications looks like this:

```
testcase I {
  packet K { PROTOCOL send { PROT_FIELDS } receive { PROT_FIELDS } }
  packet L { PROTOCOL send { PROT_FIELDS } receive ok }
  packet M { PROTOCOL send { PROT_FIELDS } receive {} }
  packet N { PROTOCOL send { PROT_FIELDS } receive ? }
}
testcase J { ... }
```

I,J,K,L,M and N are integers which define the testcase/packet id. Every testcase is run in parallel. The packet id stands for a global timeslot, so for example the packet 1.2 (testcase 1, packet 2) is sent at the same time as packet 3.2.

For each packet you have to declare the protocol type (see below) as well as a send and a receive clause where protocol specific fields are specified. In the receive part you specify which values you expect the packet to have when the packet arrives at the destination host. "receive ok" is a shortcut for a receive block with exactly the same values as in the send clause. You may also have an empty receive clause {} if you expect the packet to be dropped by the firewall. A question mark means you're not interested whether the firewall drops or forwards the packet.

3.4.2 Protocol specific fields (PROT_FIELDS)

Please read documentation [1] for further details on TCP and [3] for more information on UDP and ICMP.

```
TCP (Transmission Control Protocol):
{ srcip dstip srcprt dstprt flags seqnr acknr }
    source ip, destination ip, source port, destination port, control flags (a
    combination of S/A/F/R/U/P), sequence number, acknowledgment number

UDP (User Datagram Protocol):
{ srcip dstip srcprt dstprt }
    source ip, destination ip, source port, destination port

ICMPecho (ICMP Echo request/reply):
{ srcip dstip type idnr seqnr }
    source ip, destination ip, type (0/8), identification number, sequence
```

```
    number

ICMPunreach (ICMP Destination unreachable):
{ srcip dstip code origid }
    source ip, destination ip, code (0-15), original packet id

ICMPredir (ICMP Redirect):
{ srcip dstip code gwip origid }
    source ip, destination ip, code (0-4), gateway ip, original packet id

ICMPtexc (ICMP Time exceeded):
{ srcip dstip code origid }
    source ip, destination ip, code (0-1), original packet id

ICMPtstamp (ICMP Timestamp request/reply):
{ srcip dstip idnr seqnr otime rtime ttime }
    source ip, destination ip, identification number, sequence number,
    originate timestamp, receive timestamp, transmit timestamp
```

3.4.3 Preprocessor

Fwtest v1.0 filters the given test packet file with a preprocessor (m4). So you may define constants for often used IPs or ports. Fwtest includes by default the file defined by the environment variable FWTEST_MACROS. run_fwtest.sh sets this to macros.m4 which contains constants for possible ICMP types and codes.
It's possible to disable the preprocessor by setting the environment variable FWTEST_USEM4 to 'no'.

3.4.4 Variables

You may use variables for IP and port numbers in the packet specifications. As a firewall using NAT masks the source IP/port, you have to use such variables in the receive clause of a packet specification because the real values are not known before starting a test run. During the test, fwtest will automatically assign the variables with the gathered values of the received packets.

Variable names have to start with an uppercase letter and may not be longer than 15 characters. Numbers can be used in variable names except for the first character. Variables are only allowed for IP numbers and for port numbers. A variable is valid in the scope of a testcase. Within a testcase, a variable is assigned only once and all packets that use the same variable, have to have the same value for the field the variable is used. In different testcases the same variable name is NOT the same variable, because it is another scope.

3.4.5 Parse errors

Whenever you get parse errors on a test packet file, the line number will be displayed. Whereas the number is correct for syntax errors, Fwtest v1.0 errs mostly on a semantical error e.g. the discovery of a duplicate packet id. Because packet specifications are internalized only at the end of a testcase, such errors are not detected until the end of the actual testcase.

Since the order of the packets is not relevant (only the packet id specifies the time slot), Fwtest reverses the packets to simplify matters because of the internal data structure; so, errors may be detected in theo opposite order than they appear in the specification.

3.5 Running Fwtest

We briefly explain how to invoke fwtest without making use of run_fwtest.sh. fwtest expects the test packets file name as argument, and takes the following options:

```
        -l <log file>  Log file wherein the irregularities are stored
        -n <network>   Network 1 and mask in CIDR notation (e.g. 192.168.1.0/29)
        -i <interface> Network 1 interface to capture the packets.
        -m <network>   Network 2 and mask in CIDR notation (e.g. 172.16.70.0/29)
        -j <interface> Network 2 interface to capture the packets.
        -p             Transmit packet id in payload (neglecting ICMP)
        -u             Interpret TCPUDP packets as UDP instead of TCP
```

You have to specify the networks and the interfaces (-n, -i, -m and -j).

Fwtest may be called like this (make sure you are root):

```
./main -l test1.log -n 192.168.72.0/29 -i eth0 -m 172.16.70.0/29 -j eth1 test.tp
```

The test packets file (e.g. test.tp) has to be defined. You have no chance to perform a test without a test packets file. If the syntax of the file is invalid, fwtest will display an error message and quit instantly.

3.6 Example

Let's just show at a simple test run in a possible environment:

Test host setup:

```
    ifconfig eth0 down
    ifconfig eth1 down
    ifconfig eth0 172.16.70.2 netmask 255.255.255.0 up
```

```
    ifconfig eth1 192.168.72.2 netmask 255.255.255.0 up
    route add -net 172.16.70.0/24 gw 172.16.70.1
    route add -net 192.168.72.0/24 gw 192.168.72.1

Firewall setup:

    ifconfig eth0 down
    ifconfig eth1 down
    ifconfig eth0 172.16.70.1 netmask 255.255.255.0 up
    ifconfig eth1 192.168.72.1 netmask 255.255.255.0 up
    echo 1>  /proc/sys/net/ipv4/ip_forward
    echo 1>  /proc/sys/net/ipv4/conf/default/proxy_arp

Ping the firewall from the test host (the firewall is still accepting all
incoming packets):

    ping -c 1 172.16.70.1
    ping -c 1 192.168.72.1

Install the firewall rules on the firewall (e.g. drop all packets, but forward
tcp packets to 192.168.72.3:25):

    iptables -F
    iptables -X
    iptables -P INPUT DROP
    iptables -P OUTPUT DROP
    iptables -P FORWARD DROP
    iptables -A FORWARD -p tcp -d 192.168.72.3 --dport 25 -j ACCEPT

Create a test file named 'test.tp' like this:

    define(ipa, 172.16.70.3)
    define(ipb, 192.168.72.3)
    testcase 1 {
         packet 1 { TCP send { ipa ipb 4000 25 S 60 - } receive ok }
         packet 2 { ICMPunreach send { ipb ipa PORT_UNREACHABLE 1.1 }
                               receive {} }
    }
    testcase 2 {
         packet 1 { UDP send { ipa ipb 5005 domain } receive {} }
    }

Now fire off fwtest by calling run_fwtest.sh:

    ./run_fwtest.sh test.tp test.log 172.16.70.0/24 eth0 192.168.72.0/24 eth1

After the test run is complete, the statistics will be displayed which reports
```

one packet as successfully forwarded (true_negative) and packet 1.2 and 2.1 as successfully rejected (true_positive).

4. Timeout

A timer is startet by fwtest after the packets for a given timeslot are sent out. When the timer expires after a specified timeout value, fwtest steps to the next timeout. The timeout value is stored in the header file main.h. After changeing the timeout value, fwtest has to be recompiled.

5. Source Files

```
Makefile          - Makefile
main.c            - main program
tpparser.l        - lexer grammar
tpparser.y        - parser grammar
lpcap.c           - packet capturing routines
lnet.c            - packet crafting and injection routines
log.c             - log routines
util.c            - helper routines
symboltable.c     - a symbol table for the test packets file variables
semanticchecker.c - a semantic-checker for the test packets file
main.h            - main definitions & TIMEOUT definition
tpparser.h        - parse definitions
lpcap.h           - packet capture definitions
lnet.h            - packet crafting definitions
log.h             - log definitions
symboltable.h     - symbol table definitions
semanticchecker.h - semantic-checker definitions
```

Literatur

[Str06] Beat Strasser. Testen von Firewalls – Eine UDP-/ ICMP- Erweiterung für fwtest. http://www.infsec.ethz.ch/people/dsenn/SA_BeatStrasser_06.pdf, Februar 2006.

[Zau05] Gerhard Zaugg. Firewall Testing. http://www.infsec.ethz.ch/people/dsenn/DA_GerryZaugg_05.pdf, Januar 2005.