Kapitel 1

Testdokumentation

1.1 Unit-Tests

In Unit-Tests, auch als Modultest oder Komponententest bezeichnet, werden einzelne Teile der Software getestet. Unter Testen wir das Überprüfen, ob das Modell dem System entspricht, verstanden. Dies kann jedoch nur die Anwesenheit von Fehlern, nicht aber deren Abwesenheit nachweisen.

Im Projekt erhielt jedes Paket und die jeweils darin enthaltenen Klassen eine eigene Testdatei, in der die Unit-Tests ausgeführt werden konnten.

Da die Unit-Tests im vorliegenden Softwareprojekt bereits in der Implementierungsphase ausgeführt werden konnten, also noch vor der eigentlichen Validierungsphase, war es möglich, Fehler bereits frühzeitig zu erkennen. Zudem besteht ein weiterer Vorteil des Unit-Testens darin, dass beim Auftreten eines Fehlers dieser sehr genau eingegrenzt werden kann. Somit kann dieser Fehler schneller gefunden und dann auch behoben werden.

1.1.1 NicManagement

1.1.2 ConfigurationManagement

1.1.3 PacketDissection

1.1.4 Inspection

1.1.5 Treatment

Die LoC der Unit-Test-Datei der Klasse `Treatment` belaufen sich auf mehr als 1000. Diese hohe Zahl ist unter anderem darauf zurückzuführen, dass in `Treatment_test.cpp` auch die verwendete Hash-Funktion (XXH3) und die verwendete Map (Google-Dense-Hash-Map) auf Tauglichkeit für die spätere Verwendung in der Klasse getestet wurde. Zudem wurde ein Benchmark durchgeführt, der die Performance einer `unordered_map` und einer `dense_hash_map` vergleicht.

```
...
// TEST 0: Normal (No Testing)
// TEST 1: Unit Test without using DPDK
```

```
// TEST 2: Unit Test using DPDK
define TEST 1
...
```

Listing 1.1: Präprozessorstatment in `Treatment.h`

Durch Präprozessoranweisungen in `Treatment.h` und `Treatment.cpp` ist es möglich, Teile des Codes der Klasse `Treatment` das in eine Art „Unit-Test-Modus “ umzuschalten. Durch `TEST 1` werden beispielsweise Codesegmente umgangen, die DPDK verwenden und der Zugriff auf sonst private Membervariablen auf public gesetzt.

In den folgenden Sektionen werden beispielhaft einzelne Testfälle erläutert.

1.1.5.1 Beispiel: `check_syn_cookie()`

```
bool Treatment::check_syn:cookie(u_int32_t cookie_value, const Data\& d){
    // Extract the first 8 bit of the cookie (= timestamp)
    u_int8_t cookie_timestamp = cookie_value \& 0x000000FF;

    u_int8_t diff = _s_timestamp - cookie_timestamp;

    if(diff<1){
        // Calculate hash
        u_int32_t hash;

        // Case: same time interval
        if(diff == 0){
            // calculate expected cookie-hash
            hash = calc_cookie_hash(_s_timestamp, d._extip, d._intip, d.extport
            , d._intport);
            hash = hash \& 0xFFFFFF00;
            // stuff cookie-hash with 8 bit _s_timestamp
            hash |= (u_int8_t) _s_timestamp;
        }
        if(diff == 1){
            // calculate expected cookie-hash
            hash = calc_cookie_hash((_s_timestamp-1), d._extip, d._intip, d.
    extport, d._intport);
            hash = hash \& 0xFFFFFF00;
            // stuff cookie-hash with 8 bit _s_timestamp
            hash |= (u_int8_t) (_s_timestamp-1);
        }
         // test whether the cookie is as expected; if so, return true
         if(hash == cookie_value){
            return true;
        }
    }
    // if not unit test is running, RST will be sent
    #if TEST == 0
        //Send RST if (diff>1 XOR hash!=cookie_value)
```

```
34          PacketInfo* _rst = _packet_to_inside->get_empty_packet();
35          PacketInfoIpv4Tcp* _rst4= static_cast<PacketInfoIpv4Tcp*>(_rst);
36          _rst4->fill_payloadless_tcp_packet(d._extip, d._intip, d._export,
            d._intport, (cookie_value+1), 0, 0b00000100);
37          _rst4->recalculate_checksums();
38      }
39      #endif
40
41      //return false, so that treat_packets is able to continue
42      return false;
43  }
```

Listing 1.2: Methode: `check_syn_cookie()` in `Treatment.cpp`

Die Methode `check_syn_cookie()` (vgl. Abb. 1.2) überprüft, ob der empfangene SYN-Cookie in der richtigen Zeitspanne angekommen ist. Das ist zutreffend, wenn der Timestamp des empfangenen Cookie nicht mehr als 1 Zeiteinheit von dem aktuellen Timestamp-Wert `_s_timestamp` abweicht. Falls dem so ist, wird überprüft, ob der empfangene Cookie dem erwarteten Cookie entspricht. Die Methode gibt `true` zurück, falls dies der Fall ist so ist. Falls nicht, wird ein RST gesendet und der Rückgabewert ist `false`. Das RST wird aber nur dann gesendet, wenn sich das System im aktiven Modus (`TEST 0`) befindet. Wenn das System gerade im Unit-Test-Modus ist, in welchem kein DPDK verwendet wird, werden die Zeilen 33 bis 38 vom Compiler übersprungen.

```
1  TEST_CASE("check_syn_cookie", "[]"){
2      ....
3      SECTION("check_syn_cookie(): diff==1 with random numbers (without using the
        PacketDissection)", "[]"){
4          //Create a Treatment object
5          Treatment treat;
6
7          //Generate a random 8-bit-number
8          u_int8_t ran_num = (u_int8_t) rand();
9
10         //increment _s_timestamp up to ran_num
11         for(int i=0; i<ran_num; i++){
12             treat.s_increment_timestamp();
13         }
14
15        CHECK(treat.timestamp==ran_num);
16
17         u_int32_t extip = rand();
18         u_int32_t intip = rand();
19         u_int16_t extport = rand();
20         u_int16_t intport = rand();
21
22         //Create cookie_value with timestamp ran_num - 1
23         u_int32_t cookie_value = treat.calc_cookie_hash((ran_num - 1), extip,
           intip, extport, intport);
24         cookie_value = cookie_value  \& 0xFFFFFF00;
25         cookie_value |=(u_int8_t) (ran_num-1);
```

```
26
27          //Create a Data object
28           Data d;
29           d._extip = extip;
30           d._intip = intip;
31           d._extport = extport;
32           d._intport = intport;
33
34           CHECK(treat.check_syn_cookie(cookie_value, d));
35      }
36  ....
37  }
```

Listing 1.3: Unit-Test für die Methode `check_cookie_secret()` in `Treatment_test.cpp`

Im Unit-Test in Abb. 1.3 wird untersucht, ob die Methode `check_syn_cookie()` als Rückgabewert `true` hat (siehe Zeile 34). Außerdem wird in Zeile 15 überprüft, ob der Timestamp richtig imkrementiert wurde.

Zuerst wird in Zeile 5 das Objekt `treat` der Klasse `Treatment` erzeugt. In Zeile 8 wird anschließend eine 8-Bit lange Zufallszahl generiert, die in der Variable `ran_num` gespeichert wird. In einer for-Schleife wird dann der Timestamp des Objektes `treat` um genau diesen zufälligen Wert erhöht. Auch die vier 32-Bit langen Variablen `extip`, `intip`, `extport` und `intport` bekommen eine zufällige Zahl zugewiesen. Mit Hilfe der Methode `calc_cookie_hash()` wird in Zeile 23 der Cookie-Wert erzeugt. Hier ist zu beachten, dass der erste Parameter dieser Methode `ran_num-1` ist. Somit ist der Timestamp dieses Cookie-Wertes um genau 1 kleiner als derjenige, der in `treat` gespeichert ist. Das heißt, dass `diff` in Zeile 5 in Abbildung 1.2 genau 1 ist. Somit sollte die Methode `true` zurückgeben.In den Zeilen 28 bis 32 wird ein Datenobject `d` im Stack angelegt und mit Werten gefüllt. Der zuvor genierierte Cookie-Wert und das Datenobjekt werden anschließend in die zu überprüfende Methode `check_syn_cookie()` übergeben.

Das Verhalten der Methode `check_syn_cookie()` wird noch in acht weiteren Sektionen getestet. Hier werden unter anderem die Fälle durchlaufen, dass die Differenz des aktuellen Zeitstempels und des in `cookie_value` übergebenen Zeitstempels null, größer eins oder kleiner null ist. Außerdem gibt es einen Test, indem die IP-Adressen und Port-Nummern des Cookies und des Datenobjektes nicht übereinstimmen.

1.1.5.2 Beispiel: `s_increment_timestamp()`

```
1  void Treatment::s_increment_timestamp(){
2      // increment _s_timestamp by one
3      ++_s_timestamp;
4  }
```

Listing 1.4: Methode: `s_increment_timestamp()`

Die Methode `s_increment_timestamp()`, die in Abb. 1.4 dargestellt ist, macht vergleichsweise wenig: Sie erhöht den Wert der Membervariable `_s_timestamp` um 1.

```
1  TEST_CASE("s_increment_timestamp()", "[]"){
2      ...
```

```
    SECTION("Increment _s_timestamp up to 1000 (>255>size if u_int8_t)", "[]"){
        Treatment treat;

        u_int8_t count = 0;

        for(int i=0; i<1000; i++){
            CHECK(treat._s_timestamp == count);
            treat.s_increment_timestamp();
            count++;
        }
    }
    ...
}
```

Trotz des vergleichbar kleinen Funktionsumfangs muss getestet werden, wie sich die Variable `_s_timestamp` verhält, wenn sie mehr als 255 mal inkrementiert wurde. Denn der Datentyp `u_int8_t`, von dem diese Membervariable ist, umfasst lediglich 8 Bit und hat somit einen Wertebereich von 0 bis 255. Deshalb sollte es beim 256. Inkrementieren zum arithmetischen Überlauf kommen. Ab hier beginnt `_s_timestamp` wieder bei 0.

1.1.5.3 Beispiel: Benchmark

Um herauszufinden, welche Map sich am besten für das Softwareprojekt einigt, wurde ein Benchmark erstellt. Dieser vergleicht die Performance einer Unordered-Map mit der einer Dense-Map.

```
TEST_CASE("Benchmark", "[]"){
    typedef std::unordered_map<Data, Info, MyHashFunction> unordered;
    unordered unord;
    google::dense_hash_map<Data, Info, HashMyFunction> densemap;
    clock_t tu;
    clock_t tr;
    clock_t td;
    Data empty;
    empty._extip = 0;
    empty._intip = 0;
    empty._extport = 0;
    empty._intport = 0;
    densemap.set_empty_key(empty);
    Info flix;
    flix._offset = 123;
    flix._finseen = 0;

    //------------------------------------------------------------

    //------------------------------------------------------------

    long runs = 1;
    clock_t uclock [runs] = {};
```

```
24      clock_t dclock [runs] = {};
25      long runner = 600000;
26      Data arr [runner] = {};
27
28      for(long r = 0; r < runs; ++r) {
29
30
31          for(long i = 0; i < runner; ++i){
32              arr[i]._extip = rand();
33              arr[i]._intip = rand();
34              arr[i]._extport = rand();
35              arr[i]._intport = rand();
36          }
37          // std::cout << arr[124]._extport << std::endl;
38
39          auto startu = std::chrono::high_resolution_clock::now();
40          tu = clock();
41          for(long i = 0; i < runner; ++i){
42              unord.emplace(arr[i], flix);
43              //std::cout << "This is the extIp from connection felix: "<< iu->
    first.extip << std::endl;
44          }
45          for(long i = 0; i < runner; ++i){
46              unord.find(arr[i-1 % runner]);
47              unord.find(arr[i]);
48              unord.find(arr[i+1 % runner]);
49              unord.find(arr[i+50 % runner]);
50              //std::cout << "This is the extIp from connection felix: "<< iu->
    first.extip << std::endl;
51          }
52          tu = clock() - tu;
53          auto finishu = std::chrono::high_resolution_clock::now();
54
55
56          auto startd = std::chrono::high_resolution_clock::now();
57          td = clock() ;
58          for(long i = 0; i < runner; ++i) {
59              densemap.insert(std::pair<Data, Info>(arr[i], flix));  // insert
    rather than densemap[arr[i]]
60          }
61          for(long i = 0; i < runner; ++i){
62              densemap.find(arr[i-1 % runner]);
63              densemap.find(arr[i]);
64              densemap.find(arr[i+1 % runner]);
65              densemap.find(arr[i+50 % runner]);
66              // std::cout << "This is the extIp from connection felix: "<< id->
    first.extip << std::endl;
67          }
68          td = clock() - td;
```

```
69          auto finishd = std::chrono::high_resolution_clock::now();
70
71          std::chrono::duration<double> elapsedu = finishu - startu;
72          std::chrono::duration<double> elapsedd = finishd - startd;
73          dclock[r] = td;
74          uclock[r] = tu;
75          BOOST_LOG_TRIVIAL(info) << "Elapsed time of unordered: " << elapsedu.
            count();
76          BOOST_LOG_TRIVIAL(info) << "Clocks of patch:" << tu ;
77          BOOST_LOG_TRIVIAL(info) << "Elapsed time of dense: " << elapsedd.
            count();
78          BOOST_LOG_TRIVIAL(info) << "Clocks of dense:" << td;
79
80      }
81      int sumd = 0;
82      int sumu = 0;
83      for (long x = 0; x < runs; ++x) {
84          sumd =  sumd + dclock[x];
85          sumu =  sumu + uclock[x];
86      }
87      BOOST_LOG_TRIVIAL(info) << "This is the average clock count of densemap of
        " << runs << " rounds, of each " << runner << " elements inserted, and " <<
         4*runner  << " elements searched : " << sumd/runs;
88      BOOST_LOG_TRIVIAL(info) << "This is the average clock count of
        unordered_map of " << runs << " rounds, of each " << runner << " elements
        inserted, and " << 4*runner  << " elements searched : " << sumu/runs;
89  }
```

Listing 1.5: Benchmark zum Vergleich der Performance einer Unordered-Map und einer Dense-Map

Das Ergebnis des Benchmarks in Abb. 1.5 zeigt, dass ...

1.1.5.4 Beispiel: Densemap

Um die Verhaltensweise der Densemap vor deren Nutzung im Code besser kennenzulernen, wurden mehrere Tests geschrieben, welche die Grundfunktionalitäten der Map wie zum Beispiel das Löschen oder Hinzufügen von Werten.

```
1  TEST_CASE("Map", "[]"){
2      ...
3      SECTION("Densemap: Erase one element whose key is known", "[]"){
4          google::dense_hash_map<Data, Info, MyHashFunction> densemap;
5
6          Data empty;
7          empty._extip = 0;
8          empty._intip = 0;
9          empty._extport = 0;
10         empty._intport = 0;
11         densemap.set_empty_key(empty);
```

```
12
13          Data deleted;
14          deleted._extip = 0;
15          deleted._intip = 0;
16          deleted._extport = 1;
17          deleted._intport = 1;
18          densemap.set_deleted_key(deleted);
19
20          Data d1;
21          d1._extip = 12345;
22          d1._intip = 12334;
23          d1._extport = 123;
24          d1._intport = 1234;
25
26          Info i1;
27          i1._offset = 3;
28          i1._finseen = false;
29          densemap[d1] = i1; //calculates index over d1, store d1 as first an i1
        as second
30
31          Data d2;
32          d2._extip = 12345;
33          d2._intip = 12334;
34          d2._extport = 123;
35          d2._intport = 1234;
36
37          Info i2;
38          i2._offset = 3;
39          i2._finseen = false;
40          densemap[d2] = i2;
41
42          CHECK(densemap.size() == 2);
43          densemap.erase(d1);
44          CHECK(densemap.size() == 1);
45          densemap.erase(d2);
46          CHECK(densemap.size() == 0);
47      }
48  ...
49  }
```

Listing 1.6: Unit-Tests zum Löschen von Elementen in der Densemap

Der obige Unit-Test (vgl. Abb. 1.6) verdeutlicht, dass direkt nach Anlegen der Densemap die Methode `set_empty_key()` aufgerufen werden muss. Ohne diese Methodenaufruf ist auch nicht der Aufruf weiterer `dense_hash_map`-Methoden möglich. Deshalb wird in den Zeilen 7 bis 12 ein solcher empty-Key angelegt und als Arugment der Methode `set_empty_key()` übergeben. Dieses Argument darf kein Schlüsselwert sein und wird niemals für legitime Einträge in der Map genutzt.

Zudem wird zum Löschen von Einträgen in der Densemap mit der Methode `erase()` das Auf-

rufen der Methode `set_deleted_key()` benötigt. Dieser deleted-Key muss sich vom empty-Key unterscheiden. Da in diesem Unit-Test das Löschen von Elementen getestet werden soll, wird ein Datenobjekt mit dem Namen `deleted` angelegt, befüllt und der Methode `set_deleted_key()` übergeben.

Danach wird die Map mit zwei weiteren Einträgen befüllt. Somit muss die Densemap nun die Größe von 2 haben. Nach dem Löschen von `d1` wird überprüft, ob die Größe der Densemap sich nun auf 1 verringert hat. Nachdem der zweite Eintrag gelöscht wurde, wird in Zeile 46 nochmals auf das Übereinstimmen der Densemap-Größe mit dem Wert 0 getestet.

1.1.6 RandomNumberGenerator

1.1.6.1 Grundlegende Erläuterungen

Der `RandomNumberGenerator` (RNG) ist ein Pseudozufallszahlengenerator, der auf dem Xorshift-Algorithmus basiert. Dieser hat das Ziel, auf effiziente Weise möglichst zufällig verteilte Ganzzahlen zu generieren, welche vom Angreifer als Portnummern und IP-Adressen für von ihm ausgehende Pakete verwendet werden können. Der entwickelte RNG enthält jeweils eine Methode zur Berechnung von 16-bit und 32-bit-Zahlen, weshalb die Typen uint16_t udn uint32_t verwendet werden.

Wie sich im Code der Klasse in Abbildung 1.7 erkennen lässt, wird im Konstuktor ein **Seed**, also ein Startwert, mit der aus verschiedenen Gründen ungeeigneten Funktion `rand()` generiert. Dieser Wert wird daraufhin in den beiden Methoden durch Xor- und Shift-Operationen so verändert, dass die Ergebnisse pseudozufällig sind, also scheinbar zufällig, aber berechenbar. Für weiterführende Erläuterungen und Informationen empfiehlt sich eine Ausarbeitung von George Marsaglia [**?**].

Abschließend muss noch angemerkt werden, dass der Rückgabeparameter der Methode `gen_rdm_16_bit()` in Zeile 12 so verändert wird, dass gleich eine valide User-Portnummer zurück gegeben wird.

```
1  RandomNumberGenerator::RandomNumberGenerator() {
2      uint32_t _seed_x32 = rand();
3      uint16_t _seed_x16 = (uint16_t)_seed_x32;
4  }
5
6  uint16_t RandomNumberGenerator::gen_rdm_16_bit() {
7      _seed_x16 ^= _seed_x16 << 7;
8      _seed_x16 ^= _seed_x16 >> 9;
9      _seed_x16 ^= _seed_x16 << 8;
10     // this method returns a valid port number
11     // range should be: 1024 to 49152
12     return _seed_x16 % 48128 + 1024;
13 }
14
15 uint32_t RandomNumberGenerator::gen_rdm_32_bit() {
16     _seed_x32 ^= _seed_x32 << 14;
17     _seed_x32 ^= _seed_x32 >> 13;
18      _seed_x32 ^= _seed_x32 << 15;
19     return _seed_x32;
20 }
```

Listing 1.7: Klasse des RandomNumberGenerator

1.1.6.2 Einfache Tests

Die Tests in Abb. 1.8 sollen sicherstellen, dass auch tatsächlich ein Wert in der gewünschten Größe zurückgegeben wird.

```
1  TEST_CASE("random_number_generator_basic", "[]") {
2      ...
3      SECTION("check size of return value for 16 bit", "[]") {
4      RandomNumberGenerator xor_shift;
5          u_int16_t test_value = xor_shift.gen_rdm_16_bit();
6          // checks wheter the size of the return value is or 2 byte (16 bit)
7          CHECK(sizeof(test_value) == 2);
8      }
9
10     SECTION("check size of return vaule for 32 bit", "[]") {
11         RandomNumberGenerator xor_shift;
12         u_int32_t test_value = xor_shift.gen_rdm_32_bit();
13         // checks wheter the size of the return value is 4 byte (32 bit)
14         CHECK(sizeof(test_value) == 4);
15     }
16     ...
17 }
```

Listing 1.8: Test der Größe des Rückgabewerts

Dazu wird in Zeile 2 und in Zeile 9 jeweils ein Objekt `xor_shift` vom Typ `RandomNumberGenerator` erstellt, wodurch der Konstruktor aufgerufen und ein Seed bereitgestellt wird. In den darauffolgenden Zeilen wird dann zur beispielhaften Generierung einer Zahl einmal die Methode `gen_rdm_16_bit()` und das Pendant für 32 bit aufgerufen. Schließlich wird in den Zeilen 5 und 12 überprüft, ob die Variable `test_value` auch tatsächlich die für Portnummern und IP-Adressen benötigte Größe vorweisen kann.

Durch die in Abb. 1.9 dargestellten Tests soll geprüft werden, ob der Algorithmus bei gleichem Seed auch die gleiche Zahl generiert. Dies ist eine typische Eigenschaft von Pseudozufallszahlengeneratoren.

```
1  TEST_CASE("random_number_generator_basic", "[]") {
2      ...
3      SECTION("Check whether the same numbers are generated with the same
4      seed for 16 bit","[]") {
5          RandomNumberGenerator xor_shift_1;
6          RandomNumberGenerator xor_shift_2;
7          // set the seed to the same value in both RNGs
8          xor_shift_1._seed_x16 = 30000;
9          xor_shift_2._seed_x16 = 30000;
10         u_int16_t test_1_16_bit = xor_shift_1.gen_rdm_16_bit();
11         u_int16_t test_2_16_bit = xor_shift_2.gen_rdm_16_bit();
```

```
12          // check whether the results are the same too
13          CHECK(test_1_16_bit == test_2_16_bit);
14      }
15
16      SECTION("Check whether the same numbers are generated with the same
17      seed for 32 bit","[]") {
18          RandomNumberGenerator xor_shift_1;
19          RandomNumberGenerator xor_shift_2;
20          // set the seed to the same value in both RNGs
21          xor_shift_1._seed_x32 = 30000;
22          xor_shift_2._seed_x32 = 30000;
23          u_int32_t test_1_32_bit = xor_shift_1.gen_rdm_32_bit();
24          u_int32_t test_2_32_bit = xor_shift_2.gen_rdm_32_bit();
25          // check whether the results are the same too
26          CHECK(test_1_32_bit == test_2_32_bit);
27      }
28  }
```

Listing 1.9: Test der Größe des Rückgabewerts

Wie oben wurde dieser Test sowohl für die 16-bit-Methode als auch für die 32-bit-Methode geschrieben. Zunächst werden, wie in Z. 5 f. zu sehen, zwei Objekte der Klasse `RandomNumberGenerator` erzeugt. Anschließend wird der mit `rand()` erzeugte Seed verändert und bei beiden RNGs auf den gleichen Wert gesetzt. In Zeile 10 und 11 wird dann für jedes der beiden RNG-Objekte die Methode zum Generieren einer 16-bit-Zahl aufgerufen. Nun kann in Z. 13 sichergestellt werden, dass die Zahlen `test_1_16_bit` und `test_2_16_bit` auch wirklich gleich sind.

1.1.6.3 Test der Verteilung der Zufallszahlen

1.1.6.4 Zeitlicher Vergleich mit rand()

1.1.7 Angreifer

1.2 Ergebnisse der Tests entsprechend des Testdrehbuchs

1.3 Sonstige Tests am Testbed