Bauhaus-Universität Weimar
Fakultät Medien
Studiengang Medieninformatik (M.Sc)
16. Februar 2012

Safe and Secure Software

Bonus Assignment

Eik List, Matr. Nr. 51329
Martin Triebel, Matr. Nr. 50580

1 Mini-Project 1: SPARK-Proof The Gaussian sum formula (4 Points).

Write a program in SPARK that computes the Gaussian sum formula

$$1 + 2 + 3 + \ldots + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Make use of the data type `Positive`*. The goal of this task is to proof the correctness of your program using the SPARK proof checker. Your solution must include beside the SPARK package (ads and adb file) the following ones: the proof log (*`plg`* file), the proof reviews (*`prv`* file), and the proof summary (*`sum`* file). Below you will find a few helpful hints.*

- *Up to which value of n is the Gaussian sum computed correctly?*
- *Specify a new subtype of Positive, according to your observation, and use this subtype as the in-parameter of your subprogram.*

Wenn wir die Gauss-Summe mit einem Ergebnis bis maximal Positive'Last (2147483647) berechnen wollen, müssen wir den Eingabewert n begrenzen auf $1 \leq n \leq 46340$. Wir haben unseren Datentyp Constraint_Positive daher auf 1 .. 46340 beschränkt.

Für die Spezifikation gauss.ads brauchten wir eine derives-Annotation:

```
1 --# derives Result from N;
```

Dazu die eigentlich wichtige Post-Condition:

```
1 --# post Result = (N * (N + 1)) / 2;
```

Wir hatten bei der Ausführung von Spark eine Config-Datei mit den Grenzwerten für Positive'First, Positive'Last, etc. über den Switch

```
1 -config_file=spark.cfg
```

angegeben. Dennoch beschwerte sich beim Beweis der Proof Checker, dass er die genauen Grenzwerte nicht kannte. Um die Schreibarbeit bei der Arbeit mit dem Checker zu verringern, haben wir die Grenzwerte in der .ads als Pre-Conditions eingefügt:

```
1 --# pre Constraint_Positive'First = Positive'First
2   --#   and Positive'First = 1
3   --#   and Positive'Last = 2147483647
4   --#   and Integer'Base'First = -2147483648
```

```
5    --#    and Integer'Base'Last = 2147483647
6    --#    and Constraint_Positive'Last = 46340
7    --#    and Constraint_Positive'Last < Positive'Last;
```

Der Proof Checker erkannte dann die Werte bei den Beweisen.

Die Implementation summiert lediglich I in einer Schleife hoch. In der Theorie würde die Prozedur in der gauss.adb nur eine assert-Annotation zu Beginn der Schleife benötigen:

```
1 --# assert I < N and Result = (I * (I + 1)) / 2;
```

In der Praxis haben wir für einen einfacheren Beweis nach jedem Statement eine assert-Annotation eingefügt. Wir tasten uns also in kleinen Schritten voran, und müssen auch mehr Verification Conditions (VCs) beweisen. Dafür haben wir dann in jedem Teilbeweis ausreichend viele Hypothesen.

Auf Linux lohnt sich ein Shell-Skript zum Aufruf des Examiners, des Simplifiers und zum Start des Proof Checkers, wir haben uns ein Batchskript erstellt mit den Schritten:

```
1 spark -vcg -config_file=spark.cfg @spark
2 sparksimp -a
3 pogs -d=gauss -o=gauss.sum
4 cd gauss
5 checker -proof_log=gauss.plg
```

gauss.bat

Die Tools erzeugen uns die Dateien gauss_sum.fdl, gauss_sum.rls, gauss_sum.siv, gauss_sum.slg und gauss_sum.vcg. Der Proof Checker fragt uns nach dem Namen der Dateien und listet uns danach 13 VCs auf, die wir dann nacheinander beweisen müssen. Wir gehen dabei die VCs in beliebiger Reihenfolge einzeln durch, rufen eine noch unbewiesene VC auf und testen mit dem Befehl done, ob der Proof Checker die Conclusions der aktuellen VC selbständig beweisen kann.

```
1 1.
2 done.
```

Schafft er es nicht, so listet er nach done die bewiesenen Conclusions auf. Mit dem Befehl list erhalten wir die noch unbewiesenen Conclusions. In unserem Beweis scheitert der Proof Checker an der Deduktion allgemeiner mathematischer Regeln, etwa dass aus n <= i und i <= n folgt, dass i = n. Um den Workflow mit dem Proof Checker weiter zu vereinfachen, haben die allgemeinen mathematischen Regeln, an denen der Proof Checker scheiterte, in einer eigene Regel-Datei gauss.rul zusammengefasst. Diese konnten wir dann für jede VC über den Befehl consult einbinden und die verbleibenden Conclusions mit infer und dem Regelsatz unserer rls beweisen.

```
1 3.
2 done.
3 list.
4 consult 'gauss.rul'.
5 infer c#3 using gauss.
```

Wir haben mit unserem Ansatz alle VCs beweisen können. Zur Beweisführung gehören

- **Implementation:** gauss.ads und gauss.adb, dazu spark.cfg und unsere Regeln

in gauss.rul.

- **Sparkmake-Output:** spark.idx, spark.smf.
- **Examiner-Output:** gauss_sum.vcg (Verification Conditions), gauss_sum.fdl und gauss_sum.rls (Declaration und Regeln).
- **Simplifier-Output:** gauss_sum.siv (Verification Conditions) und gauss_sum.slg (Verification Log).
- **Proof-Checker-Output:** gauss_sum.plg (Proof Log), gauss_sum.cmd (Nutzer-Befehle während des Beweises) und die temporären Dateien gauss_sum.plg- und gauss_sum.cmd-.

2 Mini-Project 2: SPARK-Proof Faculty (4 Points).

Write a program in SPARK that computes the faculty of n. Make use of the data type `Positive`*. The goal of this task is to proof the correctness of your program using the SPARK proof checker. Your solution must include beside the SPARK package (*`ads` *and* `adb` *file) the following ones: the proof log (*`plg` *file), the proof reviews (* `prv` *file), and the proof summary (*`sum` *file). Below you will find a few helpful hints.*

- *Up to which value of n is n! computed correctly?*
- *Spepecify a new subtype of Positive, according to your observation, and use this subtype as the in-parameter of your subprogram.*
- *You can proof the correctness by using a proof function and the SPARK simplifier. Do not forget that you need a* `cfg`*-file for run-time checks.*

Diese Aufgabe haben wir gemeinsam mit der Gruppe von Felix und Paul gelöst.

Wenn wir die Fakultät mit einem Ergebnis bis maximal Positive'Last (2147483647) berechnen wollen, müssen wir den Eingabewert n begrenzen auf $1 \leq n \leq 12$. Wir haben unseren Datentyp Constraint_Positive daher auf 1 .. 12 beschränkt.

Für die Spezifikation factorial.ads brauchten wir wieder die Post-Condition und eine Beweisfunktion, die wir mit einer function-Annotation angeben:

```
1 --# function Test(N: Constraint_Positive) return Positive;
2 function Fac(N: Constraint_Positive) return Positive;
3 --# return Test(N);
```

Die Grenzwerte für Positive'First, Positive'Last, etc. haben wir wie in Aufgabe 1 wieder als Pre-Conditions eingefügt.

Die Implementation multipliziert nur I in einer Schleife auf. In der Theorie würde die Prozedur in der fac.adb nur eine assert-Annotation zu Beginn der Schleife benötigen:

```
1 --# assert I < N and Result = Test(I);
```

Für unseren Beweis haben wir wieder nach jedem Statement eine assert-Annotation eingefügt. Unsere Schritte zum Aufruf des Examiners, Simplifiers und zum Start des Proof Checkers sind:

```
1 spark -vcg -config_file=spark.cfg @spark
2 sparksimp -a
3 pogs -d=factorial -o=factorial.sum
4 cd factorial
5 checker -proof_log=factorial.plg
```

fac.bat

Die Tools erzeugen uns die Dateien fac.fdl, fac.rls, fac.siv, fac.slg und fac.vcg. Der Proof Checker fragt uns wieder nach dem Namen der Dateien und listet uns danach unsere VCs auf, die wir dann nacheinander beweisen müssen. Wir gehen wieder die VCs in beliebiger Reihenfolge einzeln durch wie in Aufgabe 1.

Wir haben mit unserem Ansatz alle VCs beweisen können. Zur Beweisführung gehören

- **Implementation:** fac.ads und fac.adb, dazu spark.cfg und unsere Regeln in fac.rul.
- **Sparkmake-Output:** spark.idx, spark.smf.
- **Examiner-Output:** fac.vcg (Verification Conditions), fac.fdl und fac.rls (Declaration und Regeln).
- **Simplifier-Output:** fac.siv (Verification Conditions) und fac.slg (Verification Log).
- **Proof-Checker-Output:** fac.plg (Proof Log), fac.cmd (Nutzer-Befehle während des Beweises) und die temporären Dateien fac.plg- und fac.cmd-.

3 Mini-Project 3: Whitebox testing: Statement coverage (4 Points).

Consider reasonable test cases for the vectors package, and then write a `testgen` *test driver for that package. Use* `gcov` *to ensure that all statements of the package are covered by your test. Achieve a perfect score by covering all lines of code (100 percent statement coverage.) Your solution must include beside your test driver the gcov counting (*`.gcov`* file).*

3.1 Test-Cases

Wir haben zum Testen der `Put`-Prozedur leeren, zweidimensionale und dreidimensionale Vectors berücksichtigt. Wir haben zum Testvergleich die Ausgaben in eine Textdatei umgeleitet und wieder eingelesen und zeichenweise verglichen. Für die Funktionen Kreuzprodukt, Skalarprodukt und Multiplikation jeweils First-, Last- und Mittelwerte für Vektoren getestet.

3.2 `gcov`

Zum Builden mussten wir die Flags `-fprofile-arcs, -ftest-coverage` und zum Linken das Flag `-fprofile-arcs` verwenden. `Gcov` generiert dabei `gcna` und `gcno` Dateien. Danach mussten wir das Test-Projekt einmal ausführen. Dabei für jede `.adb` und `.ads` eine gcov-Datei an. In dieser steht drin, wieviele Zeilen ausgeführt wurden. Danach muss man nur noch `gcov test_vectors` ausführen um die ausgeführten Zeilen zu sehen:

```
1 File 'D:\personal\organisation\dropbox\My␣Dropbox\Safe␣and␣
      Secure␣Software\Bonus\umsetzung\project03
2 \src\vectors.adb'
3 Lines executed:100.00% of 38
4 D:\personal\organisation\dropbox\My Dropbox\Safe and Secure
      Software\Bonus\umsetzung\project03\src\v
5 ectors.adb:creating 'vectors.adb.gcov'
```

4 Mini-Project 3: Parallel-Mini-RC4 key extraction (4 Points).

Write a program that computes the Key for a given Mini-RC4 keystream. The program takes one command-line arguments, the keystream represented as a hex string. Your implementation should use four tasks to compute the key in parallel. Furthermore, pressing the "q" key should immediately quit the program. This can be realised using the procedure `Ada.Text_IO.Get_Immediate`*. The program – if not interrupted – should output a candidate key represented as hex string.*

4.1 Ausgangssituation

RC4 besitzt eine variable Schlüssellänge bis 2048 Bit. Der Algorithmus besitzt eine 8×8-S-Box S Die Bytes werden sequentiell verschlüsselt. Vor der Verschlüsselung wird S mit Hilfe des Schlüssels K mit Pseudozufallswerten aufgefüllt.

```
for i in 0 .. 255 loop
  S[i] = i;
end loop;

j = 0;

for i in 0 .. 255 loop
  j = (j + S[i] + K[i mod k]) mod 256;
  swap(S[i], S[j]);
end loop;
```

Für die Verschlüsselung eines n Byte langen Plaintexts wird mit Hilfe der S-Box ein n Byte langer Schlüsselstrom generiert.

```
for k in 0 .. n loop
  i := i + 1;
  j := j + S[i];
  Swap(S[i], S[j]);
  t := S[i] + S[j];

  keystream[k] := S[t];
end loop;
```

Der Klartext wird byteweise mit dem Schlüsselstrom XOR-verknüpft um den Chiffretext zu erstellen. Für die Entschlüsselung wird der Chiffretext ebenfalls mit dem Schlüsselstrom XOR-verknüpft um den Plaintext zu erhalten:

$$C = K \oplus P, \qquad P = K \oplus C$$

Eine Angreiferin Eve kann daher aus einem bekannten Plaintext-Ciphertext-Paar (P, C) den Schlüsselstrom direkt herleiten: $K = C \oplus P$. Daraus kann sie allerdings nicht direkt den geheimen Schlüssel herleiten. Wir verwenden allerdings Mini-RC4 statt RC4. Hierbei ist der Schlüssel auf 4 Byte, also 32 Bit beschränkt:

```
subtype Key_Type is Byte_Array (0 .. 3);
```

Wir können demnach alle Schlüsselvarianten in praktikabler Zeit durchprobieren. und den ergebenden Schlüsselstrom mit unserem Eingabeparameter vergleichen. Dies ist unsere Ausgangssituation: Wir haben als Angreifer den Schlüsselstrom K irgendwie durch ein Plaintext-Ciphertext-Paar erhalten und wollen unserem Programm den

Schlüsselstrom eingeben um den geheimen Schlüssel zu ermitteln.

4.2 Umsetzung

Wir haben uns einen Testcase in der `byte_utils_test.ts` geschrieben, der für einen Schlüssel `(01, 02, 03, 04)` den Keystream ermittelt und ausgibt. Dieser Keystream diente uns dann zum Testen als Eingabeparameter für unsere Anwendung:

```
main
1cea9161eebc6f4d5cd6943247727895655 85bd1cfcd4e3448ba921fa17
3cfc40acc1a3ebc8fe4a3f71e09a31a9242f507fdd73ce7882654dae390
4bdb82cbc3099d06e5cbc7ab513638e2dffbcc62f5e4f21f86f074b3983
f950859bb38673147bf9e8552c845a2912b98422c640d12679cd8659f2a
83dfa1ee99f58f7b405c12ed257cb8f688530db0c67fcdcf21a14167dbc
2b932edd36f8f63b262f19070e5e62a1bdbdc4045668a112a3cac0c172c
14a2a56075b3f284b592fc90e10134cb6aa9b748fa20106a9145451d479
b350476458b13dad7cc99bef322823a8a42118dd8aabc3e0e289ec51e15
91ef9a226aa6577133b15d0996204959a097d71c 1200
```

Unsere Umsetzung besteht aus

- einem Hauptprogramm `main` mit
  - einem Main Task,
  - vier `Worker_Task`s sowie
  - einem `Observer_Task`;
- einem Package `Mini_RC4`,
- einem Package `Byte_Utils`,
- einem Protected Type `Mini_RC4_Properties`.

Das Hauptprogramm akzeptiert den Keystream und optional eine Maximallaufzeit als Kommandozeilenparameter. Danach startet es die Worker und den Observer. Der Observer bricht das Programm ab, wenn der Nutzer `"q"` drückt oder die Zeit abläuft. Das Package `Byte_Utils` vereint `Byte` und `Byte_Array`-Typen und Hilfsfunktionen zum Konvertieren, Klonen, Ausgeben oder Inkrementieren von `Bytes` und `Byte_Arrays`. Der Protected Type `Mini_RC4_Properties` merkt sich den aktuellen getesteten Schlüsselkandidaten, einen als korrekt identifizierten Schlüsselkandidaten und eine Flag, ob die Berechnung fertig (Done) ist.

Die Worker Tasks greifen auf eine Instanz des Protected Type zu, holen sich den aktuell nächsten Key Candidate, berechnen mit dem `Mini_RC4` den Key Schedule, holen sich den zugehörigen Keystream und vergleichen diesen mit dem anfangs vom Nutzer eingegebenen. Stimmen die Keystreams überein, so setzen die Worker Tasks die Eigenschaft `Done` im Protected Type auf `True`, speichern den verwendeten Schlüsselkandidaten und brechen ab. Das Hauptprogramm hält die Worker-Tasks und den Observer Task an und gibt den Schlüsselkandidaten aus.

4.3 Performance

Der Flaschenhals der Implementation ist das wiederholte Ausführen von Key Schedule und die Erzeugung des Keystreams. Die Implementation konnte auf einer i3-Vier-Kern-CPU mit 2.27 GHz mit 8 Tasks durchschnittlich 170.000 Schlüssel pro Sekunde testen. Zur Optimierung lassen wir zunächst einen kürzeren Keystream generieren und vergleichen diesen mit den ersten Bytes des Eingabeparameters. Erst im Fall wenn beide gleich sind, erzeugen und vergleichen wir den vollständigen Keystream. Somit erreichten wir durchschnittlich 340.000 Schlüssel pro Sekunde. Es wird deutlich, dass auch mit diesem Ansatz das Prüfen des halben Schlüsselraums von 2^{31} noch rund 6.300 Sekunden $\equiv$ 105 Minuten benötigt.

```
Short_Key_Stream: Byte_Array(0 .. 3);
Key_Stream: Byte_Array(0 .. Input_Stream'Length - 1);

-- ...

Key_Scheduler(Key_Candidate, Context);
Get_Keystream(Context, Short_Key_Stream);

if Is_Subset(Short_Key_Stream, Input_Stream) then
  Key_Scheduler(Key_Candidate, Context);
  Get_Keystream(Context, Key_Stream);

  if Equal(Key_Stream, Input_Stream) then
    Properties.Set_Correct_Candidate(Key_Candidate);
    Properties.Interrupt;
    exit;
  end if;
end if;
```

Optimierungsansatz in den Worker-Tasks.

```
 55/ 1200 seconds elapsed. 16909075 keys tested. 307437 keys/
    second. Current Key: 01020313
All tasks finished...TRUE

Success! Found key candidate:
01020304
For input
1cea9161eebc6f4d5cd6943247727895655 85bd1cfcd4e3448
ba921fa173cfc40acc1a3ebc8fe4a3f71e09a31a9242f507fd
d73ce7882654dae3904bdb82cbc3099d06e5cbc7ab513638e2
dffbcc62f5e4f21f86f074b3983f950859bb38673147bf9e85
52c845a2912b98422c640d12679cd8659f2a83dfa1ee99f58f
7b405c12ed257cb8f688530db0c67fcdcf21a14167dbc2b932
edd36f8f63b262f19070e5e62a1bdbdc4045668a112a3cac0c
172c14a2a56075b3f284b592fc90e10134cb6aa9b748fa2010
6a9145451d479b350476458b13dad7cc99bef322823a8a4211
8dd8aabc3e0e289ec51e1591ef9a226aa6577133b15d099620
4959a097d71c
Stopping tasks.
```

Output.