Problem Set 5
Course **Security Engineering**
(Winter Term 2018)

Bauhaus-Universität Weimar, Chair of Media Security
Prof. Dr. Stefan Lucks, Nathalie Dittrich
URL: `http://www.uni-weimar.de/de/medien/professuren/mediensicherheit/teaching/`
Due Date: 21 Dec 2018, 13:30, via email to `nathalie.jolanthe.dittrich(at)uni-weimar.de`.

Goals: Getting familiar with **`gnatprove`**.

Task 1 – Introduction (No Credits)
Read the remaining Chapters of John English. Inform yourself about and install **`gnatprove`**:
`https://docs.adacore.com/spark2014-docs/html/ug/source/how_to_run_gnatprove.html`

Task 2 – Mini Project – State Machine with gnatprove (4 Credits)
Implement the **`thread`** package below, which modifies a global state:

```
1  package Thread is
2      type State  is (None, Ready, Running, Stopped, Sleeping, Waiting);
3      type Action is (Notify, Resume, Sleep, Start, Stop, Wait);
4
5      S: State := None;
6
7      procedure Initialize;
8      -- Sets S to Ready.
9
10     procedure Do_Action(A: Action) with
11     -- Updates the state S according to the given input state S, and the
12     -- given action A. Sets S to None if the transition is not defined.
13 end Thread;
```

1. Add the necessary **`Globals`** and **`Depends`** statements to successfully prove its correct data flow with **`gnatprove`** in **`flow`** mode.
2. Add the necessary Pre-/Post-conditions and whatever it needs to successfully prove its correctness with **`gnatprove`** in **`prove`** mode. You can add as many helper functions as you like.

Task 3 – Mini Project – Voting with gnatprove (5 Credits)
Implement the **`elections`** package below:

```
1  package Elections is
2      type Party is (None, A, B, C, D);
3      type Votes_Array is array(Party) of Natural;
4
5      Zero_Votes_Distribution: constant Votes_Array := (others => 0);
6      Votes_Distribution: Votes_Array := Zero_Votes_Distribution;
7      Num_Votes_Made: Natural := 0;
8      Num_Total_Voters: Natural := 0;
9
10     procedure Initialize(Num_Voters: Natural);
```

```
    -- Resets the number of made votes and votes for all parties to 0, and
    -- sets the number of total Voters to the given.
    procedure Vote_For(Vote: Party);
    function All_Voters_Voted return Boolean;
    function Find_Winner return Party;
    -- Returns Party with most votes assigned.
    -- Returns None if multiple parties share the highest votes.
end Elections;
```

1. Add the necessary **Globals** and **Depends** statements to successfully prove its correct data flow with **gnatprove** in **flow** mode.
2. Add the necessary Pre-/Post-conditions and loop invariants to successfully prove its correctness with **gnatprove** in **prove** mode.

Task 4 – Mini Project – Ticket Machine with gnatprove (4 Credits)

Implement the **ticket_machine** package below:

```
package Ticket_Machine is
  -- Simulation of a ticket machine that hands out tickets which
    -- cost 30 EUR each. The machine accepts only 5, 10, or 20 EUR
    -- notes and outputs the ticket immediately after the user has
    -- paid the required amount. Note that overspending is possible.
    -- The state is reset to 0 immediately after reset and after
    -- printing the ticket.
  subtype State is Natural range 0..30;
  type Action is (Insert_Five_Eur, Insert_Ten_Eur, Insert_Twenty_Eur, Reset);
  type Reaction is (Nothing, Reset, Print_Ticket);

  Ticket_Prize: constant Natural := 30;

  procedure Initialize(S: out State);
  procedure Do_Action(S: in out State; A: in Action; R: out Reaction);
end Ticket_Machine;
```

1. Add the necessary **Globals** and **Depends** statements to successfully prove its correct data flow with **gnatprove** in **flow** mode.
2. Add the necessary Pre-/Post-conditions and whatever it needs to successfully prove its correctness with **gnatprove** in **prove** mode. You can add as many helper functions as you like.

Task 5 – Mini-Project – Primitives in Ada (5 Credits)

Port the following x86 implementation of MurmurHash2 using the following interface.

```
uint64_t MurmurHash2(const void *key, int len, uint64_t seed) {
    const uint64_t m = (0xc6a4a793L << 32) | 0x5bd1e995L;
    const int r = 24;
    uint64_t h = seed ^ len;
    const unsigned char *data = (const unsigned char *)key;

    while (len >= 8) {
        uint64_t k = *(uint64_t*)data;

        k *= m;
        k ^= k >> r;
        k *= m;

        h *= m;
        h ^= k;
```

```

        data += 8;
        len -= 8;
    }

    switch(len) {
        case 7: h ^= (uint64_t)(data[6]) << 48;
        case 6: h ^= (uint64_t)(data[5]) << 40;
        case 5: h ^= (uint64_t)(data[4]) << 32;
        case 4: h ^= (uint64_t)(data[3]) << 24;
        case 3: h ^= (uint64_t)(data[2]) << 16;
        case 2: h ^= (uint64_t)(data[1]) << 8;
        case 1: h ^= (uint64_t)(data[0]);
        h *= m;
    };

    h ^= h >> 13;
    h *= m;
    h ^= h >> 15;
    return h;
}
```

```
with Interfaces;

package Murmur_Hash2 is

    type Int64  is new Interfaces.Integer_64;
    type Uint8  is new Interfaces.Unsigned_8;
    type Uint32 is new Interfaces.Unsigned_32;
    type Uint64 is new Interfaces.Unsigned_64;

    type Byte_Array is array (Uint64 range <>) of Uint8;

    subtype Key_Type   is Byte_Array (Uint64 range 0..7);
    subtype Hash_Type  is Byte_Array (Uint64 range 0..7);

    procedure Hash(Message: Byte_Array;
                   Seed: Key_Type;
                   Result: out Hash_Type);

end Murmur_Hash2;
```