D5.4
CEA's use case final report

Project number:	731453
Project acronym:	VESSEDIA
Project title:	Verification engineering of safety and security critical dynamic industrial applications
Start date of the project:	1st January, 2017
Duration:	36 months
Programme:	H2020-DS-2016-2017

Deliverable type:	Report
Deliverable reference number:	DS-01-731453 / D5.4/ 1.0
Work package contributing to the deliverable:	WP 5
Due date:	December 2019 – M36
Actual submission date:	20th December 2019

Responsible organisation:	CEA
Editor:	Mounir KELLIL
Dissemination level:	PU
Revision:	1.0

Abstract:	The objective of this document is to discuss the work progress of the analysis of source code associated to a number of critical functionalities of the CEA use case. This document also highlights the lessons learnt from the use of different software verification tools (Frama-C WP, Frama-C EVA, and VeriFast).
Keywords:	Firmware update, 6LoWPAN, data communication

The project VESSEDIA has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731453.

Editor

Mounir KELLIL (CEA)

Contributors (ordered according to beneficiary numbers)

Pierre, ROUX (CEA)

Allan BLANCHARD (CEA)

Bart JACOBS (KUL)

Executive Summary

The 6LoWPAN management platform ensures over-the-air software update of low power devices in a 6LoWPAN network.

This deliverable discusses the work progress about the use of a number of software verification tools to analyse different fragments of the source code of the 6LoWPAN management platform. This document also presents a number of remarks regarding the lessons learnt from the analysis work.

In particular, the document exposes the usage experience of Frama-C, its WP plugin, and EVA plugin for C code analysis and VeriFast plugin for Java code analysis.

WP plugin has been used for a particular part of the C code: MPL routing code, to verify a number of code-specific properties, whereas EVA plugin has been used to identify potential runtime errors over the whole C code of the 6LoWPAN management platform.

The training phase on the software verification tools has taken an important part of the work effort in the analysis process of the 6LoWPAN management platform.

In addition, the work on the ACSL/WP annotations for the MPL code has followed several rounds in order to improve the accuracy of the annotations and get effective analysis results. Based on the annotations, 50 goals have been proved as valid from 152 goals. 102 goals reached a timeout. No goal has been proved to be not valid.

Also, EVA analysis tool was very useful for detecting potential runtime errors, some of which have been confirmed and corrected after further verification, others have been considered as not effective, whereas others are under review.

Besides, Java source code verification with VeriFast allowed to identify risks that practical tests have not identified. The analysis result raised a race condition issue that has been corrected.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Goal of the Document

This document aims at elaborating on the work progress relating to CEA use case’s analysis results, since the release of the intermediate report (D5.3). This document also discusses a number of lessons learnt from the use of different verification tool for the 6LoWPAN management platform.

1.2 Structure of the Document

This document is organized as follows. Section 2 discusses the evaluation target and explains the choice of the evaluation tools. Section 3 elaborates on the different steps of the verification procedures that were applied to the different parts of the evaluation target. Section 4 details the lessons learnt from the experience with the different verification tools applied to the 6LoWPAN management platform. Finally, section 5 concludes this document.

1.3 Related deliverables

This deliverable complements D5.3 (CEA use case intermediate report in two ways. First, this deliverable presents the work progress from the D5.3 by exposing the different analysis activities of the MPL routing code, the firmware management code as well as the gateway. Second, the present deliverable summarizes the analysis activities done during the VESSEDIA project and highlights the experience and lessons learnt from the use of Frama-C (especially WP and EVA) and VeriFast verification tools.

This deliverable is closely related to the D1.2, which describes the requirements of the different use cases of WP5 and points out the assets to protect in each use case.

This deliverable is also closely related to D3.1, which discusses the automation of the inference of properties on safety-critical scenarios, including the firmware update scenario.

Chapter 2 Target of Evaluation

2.1 Description

The 6LoWPAN platform aims at providing firmware updates for 6LoWPAN mesh networks. It comprises three functional components: 1) LLN Network Manager, 2) LLN node, and 3) LLN gateway (cf. Figure 1).

- **Management server:** this component runs on Android OS. It is in charge of transmitting firmware updates and reboot requests to the Low power & Lossy Network (LLN) node.
- **Gateway (GW):** it runs on Embedded Linux OS. It is in charge of interconnecting the LLN network with a WAN to enable communication exchange between the management server and the LLN node.
- **LLN node:** it is an embedded hardware platform with a microcontroller in addition to one or more sensors and/or actuators. It runs on Contiki OS. The managed node runs some specific application layer program (e.g., transmitting environmental/physical information like temperature, position, etc.) to the management server. In addition, the LLN node is in charge of forwarding/routing data packets in the LLN network. Also, the LLN node dynamically loads the new (piece of) firmware after it completes the reception of firmware update data from the management server.

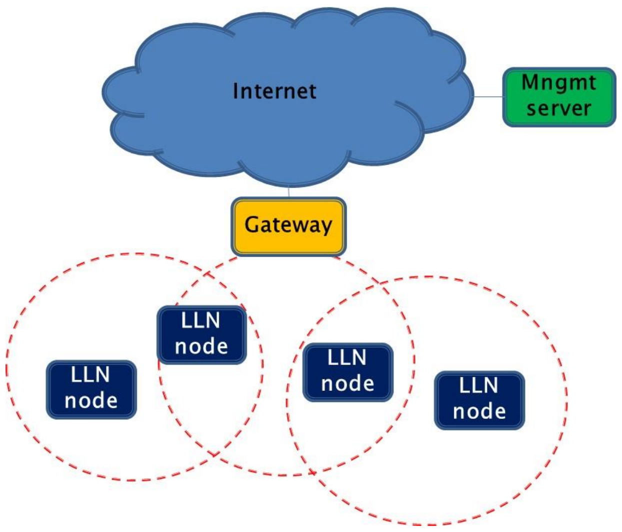


Figure 1: 6LoWPAN Platform Overview

The gateway and LLN node functional components of the 6LoWPAN management platform have been implemented using C in a Contiki OS environment (although the gateway runs on a Linux/embedded Linux environment). The network manager has been implemented with Java/Android. Figure 2 illustrates this and highlights the main functionalities per functional component.

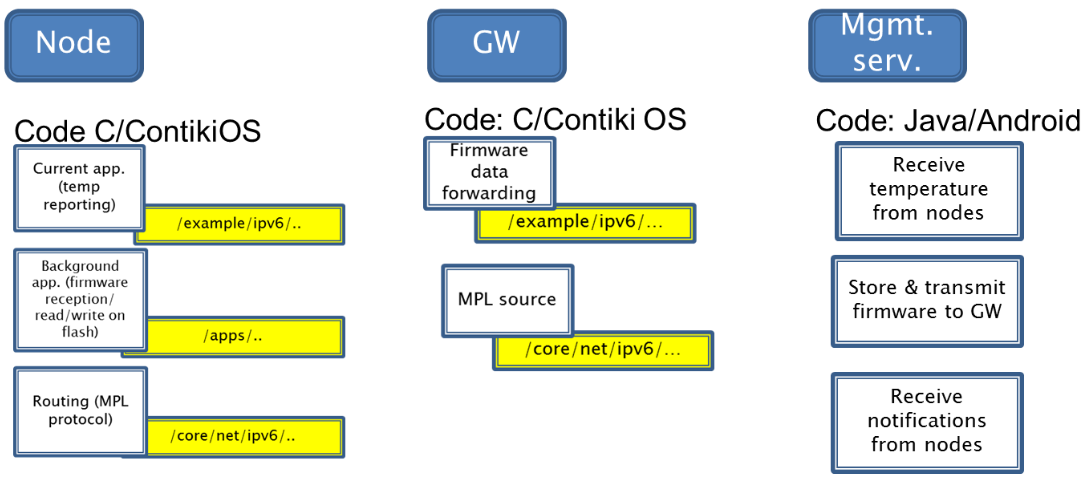


Figure 2: 6LoWPAN Platform – key functionalities per functional component

2.1.1 Choice of the C code to be analysed

Different critical assets related to the 6LoWPAN management platform have been identified in D1.2. The C source part represents a subset of the critical assets identified for the 6LoWPAN management platform. This subset can be summarized in two critical functionalities:

- *Operations on the Flash memory of the LLN node:* it includes the set of operations related to Flash memory initialization before the LLN node starts writing the first data of the new firmware on the Flash memory, as well as read/write operations of firmware data on a pre-allocated Flash memory area. These operations are particularly critical because any Flash memory read/write failure will impact the LLN node (e.g., node failure, permanent reboot, etc.) and, consequently, impact its neighbouring nodes as well.
- *Firmware data transmission in the 6LoWPAN mesh network:* it includes the operations related to the transmission of the firmware data hop-by-hop, starting from the gateway until the last-hop nodes. The transmission mechanism is based on MPL protocol [2], which operates, in a distributed fashion, following three parallel phases: broadcast packets to next-hop neighbours, announce missing packets, and re-broadcast announced missing packets. This three-phased protocol should run safely by ensuring that the full firmware has been delivered to all the network nodes with a minimum communication overhead.

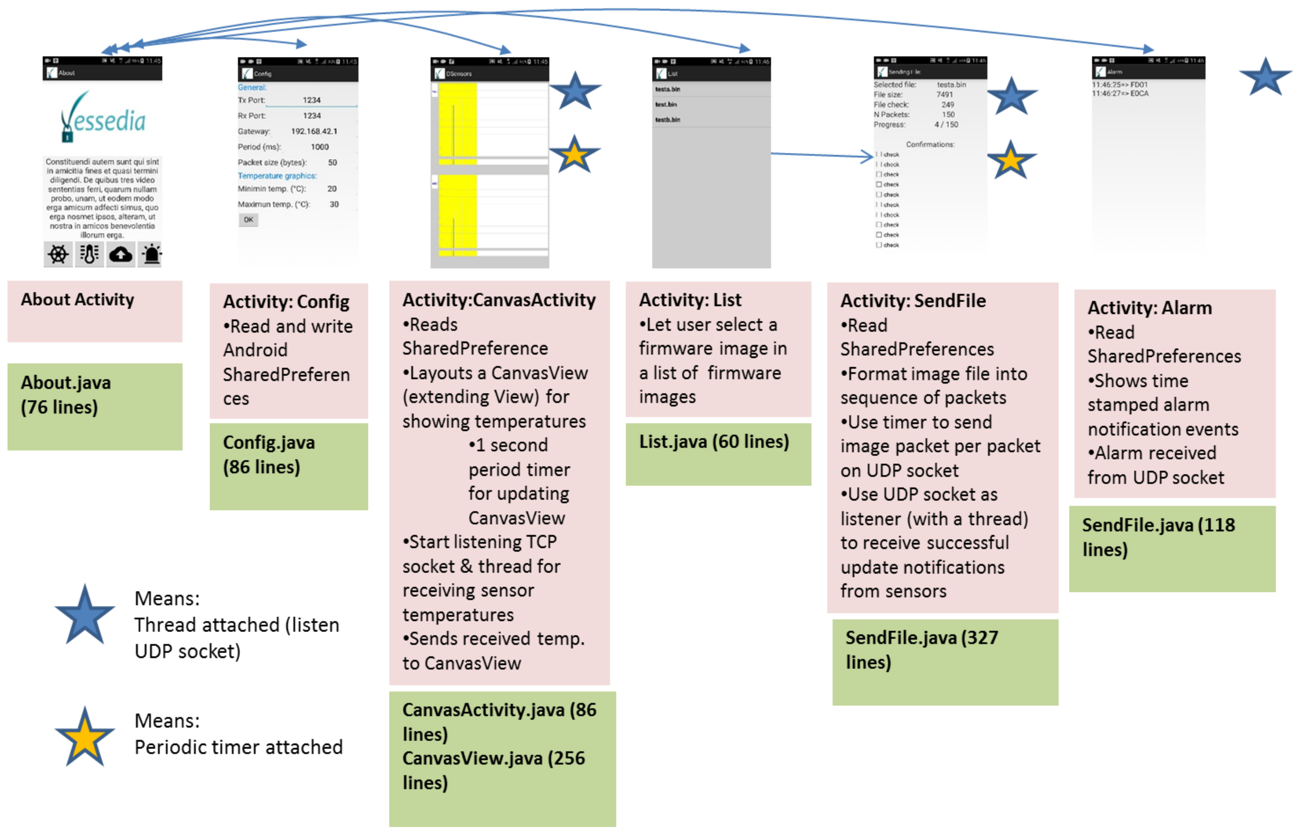
Critical function	**Critical asset (D2.1)**				
	Firmware image transmission	Flash memory partition initialization	Notification of end of firmware transmission	Loading of the new firmware	Reboot command
Operations on the Flash memory of the LLN node.		X	X	X	X
Firmware data transmission in the 6LoWPAN mesh network.	X		X		X

Table 1: Critical functionalities and associated critical assets from D1.2

2.1.2 Choice of the Java code to be analysed

The Java source of the network manager comprises various parts (or Activity templates) illustrated in the following figure.

To perform code analysis on the network manager's Java source code, all the Activity templates will be analysed with the focus on detecting race conditions (e.g., unsynchronized accesses to shared variable in a multi-threading scenario).



About Activity

About.java (76 lines)

Activity: Config
•Read and write Android SharedPreferences

Config.java (86 lines)

Activity:CanvasActivity
•Reads SharedPreference
•Layouts a CanvasView (extending View) for showing temperatures
•1 second period timer for updating CanvasView
•Start listening TCP socket & thread for receiving sensor temperatures
•Sends received temp. to CanvasView

CanvasActivity.java (86 lines)
CanvasView.java (256 lines)

Activity: List
•Let user select a firmware image in a list of firmware images

List.java (60 lines)

Activity: SendFile
•Read SharedPreferences
•Format image file into sequence of packets
•Use timer to send image packet per packet on UDP socket
•Use UDP socket as listener (with a thread) to receive successful update notifications from sensors

SendFile.java (327 lines)

Activity: Alarm
•Read SharedPreferences
•Shows time stamped alarm notification events
•Alarm received from UDP socket

SendFile.java (118 lines)

★ Means:
Thread attached (listen UDP socket)

★ Means:
Periodic timer attached

Figure 3: Activity templates of the network manager's java source code

2.2 Security objectives

The management of low power networks like 6LoWPAN networks is particularly important for network operators and network service providers because it aims at ensuring a good network performance, while maximizing network lifetime by correcting software bugs, enforcing security services, and so forth. However, LLN networks are usually deployed in hard-to-reach environments (e.g., inside pipelines, and hazardous zones) or could be massively deployed over large areas like industrial plants, smart cities, etc. This makes manual maintenance particularly difficult. In such cases, remotely managing nodes is an obvious alternative to the management of nodes via physical access. The CEA use case is represented by a remote management platform for 6LoWPAN mesh networks (6LoWPAN management platform). The platform integrates over-the-air firmware update operations on the 6LoWPAN nodes, where the firmware update may be partial/modular or full.

The set of firmware update functions modify the behaviour of the 6LoWPAN network and associated services and applications from one setting to another, without interrupting the current setting. This critical procedure should be well protected against various security threats (cf. D1.2 for further details on this objective) and safe by avoiding service interruption or abnormal behaviour of the 6LoWPAN network (e.g., unexpected node reboot, connection interruption, buffer overflow, etc.). Code analysis of the 6LoWPAN platform enables the **verification of the correct behaviour** of the said platform and **identifies potential runtime errors** during the firmware update phase.

2.2.1 Choice of the verification tools for the C code

To perform static analyse of the C source code associated to the LLN node and gateway functional components of the 6LoWPAN platform, the Frama-C WP [3][4][5] and Frama-C EVA [6] plugins have been used. Frama-C WP enables to verify the correct behaviour of the program whereas Frama-C EVA enables to detect potential runtime errors in the program.

2.2.1.1 Use of WP plugin

The WP plugin [3] is based on the Weakest Precondition calculus, which is a technique used to prove program properties, expressed through ACSL annotations of C functions, based on the {P}S{Q} Hoare triple (P: precondition, Q: post condition associated to a code fragment S).

From the aforementioned list of assets, it was decided to use WP plugin to verify the properties of the firmware image transmission asset, because this particular asset includes many critical C functions that aim at ensuring the reliable delivery of the firmware data from the management server to the 6LoWPAN nodes. A "reliable delivery" means that, when a 6LoWPAN node receives a firmware packet, it not only passes this packet to the application layer, but it also both stores a copy of this packet, broadcasts another copy to the neighbouring nodes, and re-broadcasts any stored copy of packet if some neighbouring node did not receive it. This distributed store-and-forward operations require packet buffering structures as well as variables and associated functions in the 6LoWPAN node. These C program elements need to be verified/analysed by the code developer in order to make sure that the said operations behave as expected.

2.2.1.2 Use of EVA plugin

The Eva plugin [4] is in charge of automatically computing sets of possible values for variables of an analyzed program and warns about possible runtime errors. Eva plugin has been used to detect potential runtime errors in the whole C code of the 6LoWPAN management platform (6LoWPAN node and 6LoWPAN gateway). This C code encompasses all the critical assets mentioned in section 2.1.1.

It is worth mentioning, though, that a preliminary work has been conducted with regard to the use of WP plugin for the verification of the following assets: flash memory partition initialization, notification of end of firmware transmission, loading of the new firmware, reboot command (cf. section 3.1.1 of D5.3). However, because of the significant amount of work needed to annotate the whole C code of the 6LoWPAN management platform, the priority has been given to the application of EVA tool to check those assets against potential runtime errors while focusing the WP analysis work mainly on firmware image transmission.

2.2.2 Choice of the verification tool for the Java code

The software update management server is a key component of the OTA software update solution. Therefore, its reliability is essential.

Though extensive tests can be used to make the management server somehow trustworthy, full trust in the software management server demands software verification.

Because the 6LoWPAN use case was initially meant as a proof of concept demonstrator, both server and terminal functionalities have been merged in the same equipment for simplicity.

The Android Operating System was selected for this equipment, so that server functionalities are implemented as an Android application.

Software verification of an Android application requires a software verification tool that is able to perform verification on java programs.

In the context of VESSEDIA, we have selected VeriFast [7] for performing Java software verification on this Android application. However, VeriFast is not limited to software verification of Java programs: other programming languages such as the C language can be analyzed with VeriFast as well.

The focus for program verification was not put on the Android OS itself, but rather on the Android application source code. The purpose was to detect malfunctioning of the code such as race conditions, resource leakage, and so on. Race conditions, in particular, may translate into a seldom and random malfunctioning, which may not be observed during elementary tests of the Java application.

In a way similar to other software verification tools, VeriFast requires source code annotations that allow the programmer to express in a formal way the properties that he/she expects from the implementation. In particular, pre-conditions and post-conditions can be described for each method in a Java class, under the form of annotations which are ignored by compilers but exploited by verification programs such as VeriFast.

The programmer describes which assumptions (pre-conditions) are to be considered from method arguments at method calling time, and he/she should describe as well which properties are expected on the returned value of the method.

Chapter 3 Use case realization

The analysis of the 6LoWPAN management platform encompasses the analysis of C code’s critical functions (operations on the Flash memory of the LLN node and firmware data transmission in the 6LoWPAN mesh network) and Java code’s critical functions (all the activity templates, with the focus on detecting race conditions) exposed in section 2.1.

3.1 Analysis of the C source part – Firmware image transmission code

Analysing firmware transmission of the CEA use case leads to the analysis of the MPL source code of Contiki OS[1], encompassing the 6LoWPAN node (MPL router) and 6LoWPAN gateway (MPL source) . This code is particularly critical in that if a bug occurs in the MPL routing, this may result in the interruption of the firmware transfer procedure. Therefore, it is particularly important to analyse the MPL source code with a particular focus on loops and buffers.

3.1.1 Preparation

Firmware image transmission is ensured using MPL routing protocol, which runs on the 6LoWPAN gateway (MPL source) and the 6LoWPAN node (MPL router). The operations related to MPL routing at the 6LoWPAN node have been verified using both the WP plugin and the EVA plugin of Frama-C. For the MPL operations at the gateway (MPL source), only the EVA plugin has been used.

The table below summarizes the different sections of the MPL source code on the 6LoWPAN node.

Annotated segments of the source code		Description
6LoWPAN node	`static void init()`	Initialization of MPL routing timers and buffers
	`static struct sliding_window * window_allocate()`	Allocation of the sliding window on the MPL router, where the sliding window is associated to a given MPL source
	`static void window_update_bounds(void)`	Update upper and lower bounds of the sliding window
	`static struct mcast_packet * buffer_reclaim(void)`	Release a data buffer entry
	`static struct mcast_packet * buffer_allocate(void)`	Allocate additional memory space for a data buffer
	`static void icmp_output(void)`	Transmit ICMP control messages
	`static void icmp_input(void)`	Receive ICMP control messages
	`static uint8_t in (void)`	Filter incoming multicast packet based on already internally registered multicast addresses.
	`static void double_interval (void *ptr)`	Double the trickle interval when the current one expires
	`static clock_time random_interval (clock_time_t, unint8_t)`	Generating clock ticks randomized over a given time interval
	`static void reset_trickle_timer (uint8_t)`	Reset the trickle timer
	`static uint8_t accept(uint8_t in)`	Parse the hop-by-hop MPL header
	`static void handle_timer (void *)`	Manage trickle timer structure (e.g., update its values) based on received multicast packets
	`static void  out (void)`	Add the MPL hop-y-hop option header to the outgoing IPv6 packet
6LoWPAN gateway	static void multicast_send (void)	Transmit multicast packets in the 6LoWPAN network
	static void ipaddr_add (const uip_ipaddr_t *)	Adding an IP address to the routing table

Table 2: Annotated segments for firmware data transmission in the 6LoWPAN mesh network

[1] MPL implemenation of Conitki OS has been a bit modified by CEA to enable interfacing with the firmware-specific operation on the Flash memory of the LLN node.

The table below summarizes the different sections of the MPL source code on the gateway.

3.1.1.1 Preparation for code verification with WP

D5.3 described some preliminary work on the preparation of MPL code analysis with the WP plugin (installation of the WP plugin, first ACSL annotations written for the MPL code, and modification of a Contiki OS-specific script written by FOKUS to use the WP plugin for MPL code analysis).

```
frama-c-gui            roll-tm_wp.c    \
        -cpp-extra-args=" -DCONTIKI=1"\
" -DCONTIKI_TARGET_NATIVE=1"\
" -DNETSTACK_CONF_WITH_IPV6=1"\
" -DUIP_CONF_IPV6_RPL=1"\
" -I/usr/local/include"\
.
.
" -I../../../../platform/native/"\
" -I../../../.."\
"         -DCONTIKI_VERSION_STRING=\"Contiki-3.x-3214-
gedb3046\""    \
        -wp                                    \
        "$@"
exit
```

Figure 4: Fragment of the WP script for MPL code analysis

Since then, the work on the ACSL annotations for the MPL code has followed several rounds in order to improve the accuracy of the annotations and get effective analysis results. The main difficulty lied in adapting conventional ACSL annotations, exemplified in WP tutorials [4][5], to the case of MPL code, where the code structure includes different types of data structures; many pointers, nested conditional segments, and loop iterations based on pointers.

A typical example of this case is the annotation of function `window_allocate( );` a function that allocates a sliding window data structure per MPL source (like in TCP) for message storage and retransmission (cf. *Figure 5*). In this function, it was necessary to formulate the inequality of the loop invariant associated to the pointer *`iterswptr`* of the list of sliding windows, with integer parameters (in the form: `A<=k<=i`, where *`A`* is an integer value. *`k`* and *`i`* are integer variables) and not (intuitively) with `iterswptr` pointer type parameters. This kind of formulation exercise is very common throughout all the MPL code. A similar example is shown in Figure 6, with the `buffer_reclaim()` function. *Figure 7* shows another example of annotated code for the function `window_update_bounds( ),` where there is a quite complex code fragment structure including a nested ‘if’ statement within a ‘for’ loop.

```
/*@
requires \valid (iterswptr);
ensures \valid (iterswptr);
assigns windows[0..(ROLL_TM_WINS -1)].count;
assigns windows[0..(ROLL_TM_WINS -1)].lower_bound;
assigns windows[0..(ROLL_TM_WINS -1)].upper_bound;
assigns windows[0..(ROLL_TM_WINS -1)].min_listed;
*/

static struct sliding_window * window_allocate()
{
//ACSL comment: when the loop finishes, 'iterswptr' will be at (windows - 1) value
because of the large inequality. It's a decrementing loop
       /*@
      loop invariant iterswptr < &windows[ROLL_TM_WINS - 1] &&
!SLIDING_WINDOW_IS_USED(iterswptr) ==> (iterswptr+1)->count == 0 &&
      (iterswptr+1)->lower_bound == -1 && (iterswptr+1)->upper_bound == -1 &&
(iterswptr+1)->min_listed == -1;
      loop invariant &windows[0]<iterswptr<=&windows[ROLL_TM_WINS - 1];
      loop invariant !SLIDING_WINDOW_IS_USED(iterswptr) ==> \forall integer k ;
0<=k<=((&windows[ROLL_TM_WINS - 1]-iterswptr)/sizeof(struct sliding_window)) ==>
windows[(ROLL_TM_WINS -1)-k].count==0 && windows[(ROLL_TM_WINS -1)-k].lower_bound==-
1 && windows[(ROLL_TM_WINS -1)-k].upper_bound==-1 && windows[(ROLL_TM_WINS -1)-
k].min_listed==-1;
      loop assigns iterswptr,  iterswptr->count, iterswptr->lower_bound, iterswptr-
>upper_bound, iterswptr->min_listed;
      loop variant iterswptr-windows;
     */

  for(iterswptr = &windows[ROLL_TM_WINS - 1]; iterswptr >= windows;
      iterswptr--) {
    if(!SLIDING_WINDOW_IS_USED(iterswptr)) {
      iterswptr->count = 0;
      iterswptr->lower_bound = -1;
      iterswptr->upper_bound = -1;
      iterswptr->min_listed = -1;
      return iterswptr;
    }
  }
  return NULL;
}
```

Figure 5: MPL's window_allocate( ) function with ACSL annotations

```
static struct mcast_packet *
buffer_reclaim()
{
  struct sliding_window *largest = windows;
  struct mcast_packet *rv;
 /*@
requires \valid (largest);
requires \valid (rv);
ensures \valid (largest);
ensures \valid (rv);
 */

/*@
        loop invariant iterswptr < &windows[ROLL_TM_WINS - 1] && (iterswptr->count >
largest->count) ==>  largest == (iterswptr+1) ;
        loop invariant &windows[0] <iterswptr<=&windows[ROLL_TM_WINS - 1];
        loop invariant (iterswptr->count > largest->count) ==> \forall integer k ;
(int)&windows[ROLL_TM_WINS         -         1]>k>=         (int)iterswptr         &&
largest==&windows[(ROLL_TM_WINS -1)-k];
        loop assigns iterswptr, largest, iterswptr->count;
        loop variant iterswptr- windows;
      */
  for(iterswptr = &windows[ROLL_TM_WINS - 1]; iterswptr >= windows;
      iterswptr--) {
    if(iterswptr->count > largest->count) {
      largest = iterswptr;
    }
  }
...
       /*@
       loop  invariant  locmpptr  <  &buffered_msgs[ROLL_TM_BUFF_NUM  -  1]  &&
MCAST_PACKET_IS_USED(locmpptr) &&(locmpptr->sw == largest) &&
       (SEQ_VAL_IS_EQ(locmpptr->seq_val,         largest->lower_bound))         ==>
rv==(locmpptr+1);
       loop                    invariant                    (int)(buffered_msgs-
1)<=(int)locmpptr<=(int)&buffered_msgs[ROLL_TM_WINS - 1];
      loop invariant MCAST_PACKET_IS_USED(locmpptr) &&(locmpptr->sw == largest) &&
(SEQ_VAL_IS_EQ(locmpptr->seq_val, largest->lower_bound)) ==> \forall integer m ;
0<m<=(ROLL_TM_BUFF_NUM - 1) && rv == &buffered_msgs[(ROLL_TM_BUFF_NUM-1)-m];
       loop assigns locmpptr, rv, largest->count;
   loop variant locmpptr-buffered_msgs;
       */
```

```
        for(locmpptr = &buffered_msgs[ROLL_TM_BUFF_NUM - 1];
      locmpptr >= buffered_msgs; locmpptr--) {
    if(MCAST_PACKET_IS_USED(locmpptr) && (locmpptr->sw == largest) &&
       SEQ_VAL_IS_EQ(locmpptr->seq_val, largest->lower_bound)) {
      rv = locmpptr;
      PRINTF("ROLL TM: Reclaim seq. val %u\n", locmpptr->seq_val);
      MCAST_PACKET_FREE(rv);
      largest->count--;
      window_update_bounds();
      VERBOSE_PRINTF("ROLL TM: Reclaim - new bounds [%u , %u]\n",
                    largest->lower_bound, largest->upper_bound);
      return rv;
    }
  }
```

Figure 6: MPL's buffer_reclaim ( ) function with ACSL annotations

```
*@
requires \valid (iterswptr);
requires \valid (locmpptr) && \valid (locmpptr->sw);
ensures \valid (iterswptr);
ensures \valid (locmpptr);
ensures \valid (locmpptr->sw);
assigns windows[0..(ROLL_TM_WINS -1)].lower_bound;
assigns windows[0..(ROLL_TM_WINS -1)].upper_bound;
 */
static void
window_update_bounds()
{
       /*@
       loop invariant &windows[0]<iterswptr<=&windows[ROLL_TM_WINS - 1];
       loop   invariant   \forall   integer   k   ;     0<k<=(ROLL_TM_WINS   -   1)==>
windows[(ROLL_TM_WINS -1)-k].lower_bound==-1 ;
       loop assigns iterswptr, iterswptr->lower_bound;
         loop variant iterswptr-windows;
        */
  for(iterswptr = &windows[ROLL_TM_WINS - 1]; iterswptr >= windows;
      iterswptr--) {
    iterswptr->lower_bound = -1;
  }
```

```
       /*@
        loop invariant locmpptr < &buffered_msgs[ROLL_TM_BUFF_NUM - 1] &&
MCAST_PACKET_IS_USED(locmpptr) ==> iterswptr ==(locmpptr+1)->sw;
        loop invariant locmpptr < &buffered_msgs[ROLL_TM_BUFF_NUM - 1] &&
MCAST_PACKET_IS_USED(locmpptr) && (iterswptr->lower_bound < 0 ||
SEQ_VAL_IS_LT(locmpptr->seq_val, iterswptr->lower_bound)) ==> iterswptr-
>lower_bound == (locmpptr+1)->seq_val;
        loop invariant locmpptr < &buffered_msgs[ROLL_TM_BUFF_NUM - 1] &&
MCAST_PACKET_IS_USED(locmpptr) && (iterswptr->upper_bound < 0 ||
SEQ_VAL_IS_GT(locmpptr->seq_val, iterswptr->upper_bound)) ==> iterswptr-
>upper_bound == (locmpptr+1)->seq_val;
        loop invariant &buffered_msgs[0]<=locmpptr<=&buffered_msgs[ROLL_TM_WINS -
1];
       loop invariant MCAST_PACKET_IS_USED(locmpptr) ==>\forall integer l ;
0<l<=(ROLL_TM_BUFF_NUM - 1) && iterswptr ==buffered_msgs[(ROLL_TM_BUFF_NUM-1)-
l].sw;
        loop invariant MCAST_PACKET_IS_USED(locmpptr) && (iterswptr->lower_bound <
0 || SEQ_VAL_IS_LT(locmpptr->seq_val, iterswptr->lower_bound))==> \forall integer
m ; 0<m<=(ROLL_TM_BUFF_NUM - 1) && iterswptr->lower_bound==iterswptr->lower_bound
== buffered_msgs[(ROLL_TM_BUFF_NUM-1)-m].seq_val;
loop invariant MCAST_PACKET_IS_USED(locmpptr) && (iterswptr->upper_bound < 0 ||
SEQ_VAL_IS_GT(locmpptr->seq_val, iterswptr->upper_bound)) ==> \forall integer n ;
0<n<=(ROLL_TM_BUFF_NUM - 1) && iterswptr->upper_bound==iterswptr->upper_bound ==
buffered_msgs[(ROLL_TM_BUFF_NUM-1)-n].seq_val;
        loop assigns locmpptr, iterswptr, iterswptr->lower_bound, iterswptr-
>upper_bound ;
         loop variant locmpptr-&buffered_msgs[0];
         */
   for(locmpptr = &buffered_msgs[ROLL_TM_BUFF_NUM - 1];
       locmpptr >= buffered_msgs; locmpptr--) {
     if(MCAST_PACKET_IS_USED(locmpptr)) {
       iterswptr = locmpptr->sw;
       VERBOSE_PRINTF ("ROLL TM: Update Bounds: [%d - %d] vs %u\n",
                      iterswptr->lower_bound, iterswptr->upper_bound,
                      locmpptr->seq_val);
       if(iterswptr->lower_bound < 0
          || SEQ_VAL_IS_LT(locmpptr->seq_val, iterswptr->lower_bound)) {
         iterswptr->lower_bound = locmpptr->seq_val;
       }
       if(iterswptr->upper_bound < 0 ||
          SEQ_VAL_IS_GT(locmpptr->seq_val, iterswptr->upper_bound)) {
         iterswptr-> upper_bound = locmpptr->seq_val;
       }
     }
   }
}
```

Figure 7: MPL’s window_update_bounds() ( ) function with ACSL annotations

All the functions of the MPL code on the 6LoWPAN node (MPL router) have been annotated.

3.1.1.2 Preparation for code verification with EVA

In order to use Frama-C EVA plugin for MPL code analysis, the WP script has been slightly modified (just replacing the WP plugin command line by the EVA plugin command line) and then applied to both the 6LoWPAN node (MPL router) and the gateway (MPL source).

```
frama-c-gui   -eva-slevel   100   -eva   -main
double_interval roll-tm.c                    \

       -cpp-extra-args=" -DCONTIKI=1"\

" -DCONTIKI_TARGET_NATIVE=1"\

" -DNETSTACK_CONF_WITH_IPV6=1"\

.

.

" -I../../../../core/ctk"\

" -I../../../../core/net/llsec"\

" -I../../../../platform/native/"\

" -I../../../.."\

              "$@"


exit
```

Figure 8: Fragment of the EVA script for MPL code analysis (example with double_interval( ) function

All the functions of the MPL code on the 6LoWPAN node (MPL router) have been verified with EVA.

3.1.2 Verification process

3.1.2.1 Verification with WP

The verification of the annotated MPL code, which covers all the functions of the MPL code, lasts about 5 minutes. The verification has been performed on an Intel Core i3 CPU m370 2.4 GHz processor.

3.1.2.2 Verification with EVA

All the MPL code functions have been verified with EVA plugin. This encompasses both the 6LoWPAN node part (MPL router) and the gateway part (MPL source). The verification of MPL code functions lasts about 2 to 3 seconds per function. The verification has been performed on an Intel Core i3 CPU m370 2.4 GHz processor..

3.1.3 Results

3.1.3.1 Results of WP verification

The WP analysis generated the following results:

```
.
.
[wp] Proved goals:   50 / 152
  Qed:             37  (4ms-50ms-988ms)
  Alt-Ergo:        13  (32ms-86ms-448ms)
(1495) (interrupted: 102)
```

In sum, based on the annotations, 50 goals have been proved as valid from 152 goals. 102 goals reached a timeout. No goal has been proved to be not valid.

3.1.3.2 Results of EVA verification for 6LoWPAN node

The results of EVA analysis for the MPL code on the 6LoWPAN node are summarized in the following table.

Function	Nb. of alarms	Descritpion
icmp_output()	1	Pointer comparision
icmp_input ()	14	Pointer comparision (6), out of bounds read (4) , division by zero (1), precondition 'valid_xxx' got status unknown (1), precondition 'initialization, s1' got status unknown (1), precondition 'dangligness, s1' got status unnown (1).
handle_timer()	2	Pointer comparision
windwo_update_bound()	2	Pointer comparision
buffer_reclaim()	2	Pointer comparision
buffer_allocate ()	1	Pointer comparision
accept()	4	Pointer comparision (2), precondition 'valid_xxx' got status unknown (2)
init( )	2	Pointer comparision
in ()	4	Pointer comparision (2), precondition 'valid_xxx' got status unknown (2)
out ()	3	Pointer comparision (1), precondition 'valid_xxx' got status unknown (2)
double_interval()	18	Out of bounds read/write (12), Signed overflow(2), invalid RHS operand of shift (3), division by zero (1)
random_interval()	2	Invalid RHS operand for shift, division by zero
reset_trickle_timer()	1	Acessing out of bounds index
window_lookup ()	4	Pointer comparision, precondition 'valid_xxx' got status unknown, precondition 'initialization, s1' got status unknown, precondition 'dangligness, s1' got status unnown,
window_allocate ()	0	N/A

Table 3: lists of alarms for MPL code based on EVA analysis

EVA alarms enabled to point out a number of potential runtime errors in the 6LoWPAN node's MPL routing code. Some are effective, other are not effective, whereas others are under review. Further details are provided in the following sub-sections, with a focus on the functions that issued critical alarms: `random_interval( )`, `double_interval( )`, and `icmp_input( )`.

3.1.3.2.1 random_interval ( ) function

A very low risk of division by zero has been pinpointed in the `random_interval( )` function and corrected as follows:

Old code fragment:

```
static clock_time_t
random_interval(clock_time_t i_min, uint8_t d)
{
  clock_time_t min = TRICKLE_TIME(i_min >> 1, d);
  VERBOSE_PRINTF("ROLL TM: Random [%lu, %lu)\n", (unsigned long)min,
                 (unsigned long)(TRICKLE_TIME(i_min, d)));
  return min + (random_rand() % (TRICKLE_TIME(i_min, d) - 1 - min));
}
```

In this old version of the code fragment, the expression `TRICKLE_TIME(i_min, d) - 1 - min)` is equal to zero when:

```
TRICKLE_TIME(i_min, d)= 1 + min
⇒ TRICKLE_TIME(i_min, d)= 1 + TRICKLE_TIME(i_min >> 1, d)
⇒ i_min << d= 1+ ((i_min >>1) << d)
⇒ i_min * 2^d=1+(i_min/2)^d
```

The above equation is verified when `i_min==1 && d==0, i_min==2 && d==0, imin==3 && d==log₂(2/3), i_min==4 && d== log₂(1/3)`,etc.

Given that `d` is of type `uint8_t`, so only the first two cases (`i_min==1 && d==0 and i_min==2 && d==0`) will generate a division by zero.

From the source code, it turns out that the value of `d` is set to the value of `i_current` parameter of the structure trickle_param. This `i_current` parameter is set to zero each time the function `resest_trickle_timer ( )` is called.

In addition, from the source code, it can be noticed that the value of `i_min` is configurable via the macro `ROLL_TM_IMIN` and may take three possible recommended values 16, 32, or 64 (depending on both the MPL forwarding strategy (aggressive vs conservative) and the choice RDC driver (Contikimac vs. nullrdc)). For other/future RDC drivers other values of i_imin may be proposed.

From the above discussion, it turns out that it is not very likely to have one of the first two cases (`i_min==1 && d==0 and i_min==2 && d==0`) in a real scenario. But, given that this probability is not null, it is safer to avoid the risk of the division by zero by modifying the code as follows:

New code fragment:

```
static clock_time_t
random_interval(clock_time_t i_min, uint8_t d)
{
  clock_time_t min = TRICKLE_TIME(i_min >> 1, d);
  VERBOSE_PRINTF("ROLL TM: Random [%lu, %lu)\n", (unsigned long)min,
                 (unsigned long)(TRICKLE_TIME(i_min, d)));
 clock_time_t e_;
e_= (TRICKLE_TIME(i_min, d) - 1 - min);
if (e_!=0) return min + (random_rand() % e_);
// return min + (random_rand() % (TRICKLE_TIME(i_min, d) - 1 - min));
}
```

3.1.3.2.2 double_interval ( ) function

20 alarms have been generated by the analysis:

- 1 division by zero
- 14 invalid memory accesses (out of bounds read)
- 2 integer overflows
- 3 invalid shifts

1) Division by zero

This alarm concerns the same line of code as the `random_interval( )` function, because this function is called by `double_interval( )` function.

2) Invalid memory accesses (out of bounds read)

This is illustrated by the following warning

```
[eva:alarm] roll-tm.c:512: Warning:  out of bounds read.
assert \valid_read(&param->xxx);
```

Where 'xxx' is a field of a structured called `trickle_param`, defined as follows:

```
struct trickle_param {
  clock_time_t i_min;           /* Clock ticks */
  clock_time_t t_start;         /* Start of the interval (absolute
clock_time) */
  clock_time_t t_end;           /* End of the interval (absolute
clock_time) */
  clock_time_t t_next;          /* Clock ticks, randomised in [I/2, I) */
  clock_time_t t_last_trigger;
  struct ctimer ct;
  uint8_t i_current;            /* Current doublings from i_min */
  uint8_t i_max;                /* Max number of doublings */
  uint8_t k;                    /* Redundancy Constant */
  uint8_t t_active;             /* Units of Imax */
  uint8_t t_dwell;              /* Units of Imax */
  uint8_t c;                    /* Consistency Counter */
  uint8_t inconsistency;
};
```

This alarm is not effective because `double_interval( )` is a callback function of Contiki OS's function `ctimer_set( )`, which sets a callback timer for a time sometime in the future. `param` is used by `ctimer_set( )` function by supplying an opaque pointer `param` as an argument to the callback function `double_interval`( ). Therefore, the pointer `param` will be pointing an effective structure when it is used by `double_interval( )`.

3) Integer overflow

This alarm is described as follows:

```
[eva:alarm] roll-tm.c:516: Warning: signed overflow. assert -2147483648 ≤ (int)param-
>i_current + 1;
 [eva:alarm] roll-tm.c:516: Warning:  signed overflow. assert (int)param->i_current +
1 ≤ 2147483647;
```

This integer overflow is not possible because of the following ''if' condition preceding the 'assertion.

```
if(param->i_current < param->i_max) {
    param->i_current++;
  }
```

Indeed, *`i_max`* parameter of the structure `trickle_param` may take two possible hard-coded values (of type `ROLL_TM_IMAX_##m`, where `m` is a binary variable):
'1' (corresponding to `Imax`=250 seconds) and,
'9' (corresponding to `Imax`=500 seconds).

4) Invalid shifts
The alarms are associated to the 3 shifting operations of `i_min` parameter of the structure `trickle_param`, and are illustrated hereafter.

```
 [eva:alarm] roll-tm.c:520: Warning:
  invalid RHS operand for shift. assert 0 ≤ (int)param->i_current < 32;
...
 [eva:alarm] roll-tm.c:492: Warning:
  invalid RHS operand for shift. assert 0 ≤ (int)d < 32;
...
 [eva:alarm] roll-tm.c:497: Warning:
  invalid RHS operand for shift. assert 0 ≤ (int)d < 32;
```

The 3 shifting operations are:
`i_min >> 1`, `i_min << d` , and `i_min << param->i_current`.

The alarm of the first shift is not effective because shifting is done by 1 position, which is lower that the width of the type of `i_min` (i.e., `uint8_t`).
For the two other shifting operations, the `i_current` value is bounded by `i_max` as follows:

```
if (param->i_current < param->i_max)
param->i_current ++;
```

Where `i_max` may take two possible values: 1 or 9.

Also, the value of `d` in the code is set to the value of `i_current` parameter. In view of that, if `i_max` is set to 9, this may result in a shifting by a number larger than the width of the type of `i_min`, whenever i_current value reaches i_max via the incrementation operation: `param->i_current ++`.

Further tests are needed to check this case.

3.1.3.2.3 icmp_input ( ) function

11 alarms have been generated by the analysis:
1 division by zero
3 invalid memory accesses
7 others

1) Division by zero
This alarm concerns the same line of code as the `random_interval( )` function, because this function is called by `double_interval( )` function. Therefore, this alarm is not effective.

2) invalid memory accesses (out of bounds read)
This type of alarms is illustrated hereafter:

```
[eva:alarm] roll-tm.c:1219: Warning:   out of bounds read. assert
\valid_read(seq_ptr);
[eva:alarm] roll-tm.c:1172: Warning: out of bounds read. assert
\valid_read(&locslhptr->flags); [eva:alarm] roll-tm.c:1197: Warning:  out of
bounds read. assert \valid_read(&locslhptr->seq_len);
```

All the mentioned alarms are associated to the following code segment:

```
#define UIP_ICMP_PAYLOAD  ((unsigned char *)&uip_buf[uip_l2_l3_icmp_hdr_len])
..
locslhptr = (struct sequence_list_header *)UIP_ICMP_PAYLOAD; ...

seq_ptr = (uint16_t *)((uint8_t *)locslhptr
                           + sizeof(struct sequence_list_header)); ...
  for(; seq_ptr < end_ptr; seq_ptr++) {
        /* Check for "They have new" */
        /* If an advertised seq. val is GT our upper bound */
        val = uip_htons(*seq_ptr);
```

These alarms are being reviewed by checking ICMP format parsing.

3) Others
These alarms concern pointer castings, and are not effective.

3.1.3.3 Results of EVA verification for the 6LoWPAN gateway

3.1.3.3.1 Multicast_send( ) function

No alarm generated.

3.1.3.3.2 ipaddr_add ( ) function

The analysis of this function generated 12 alarms:

2 invalid memory accesses (out of bound read)
6 accesses out of bounds index
4 integer overflows (signed overflow)

1) invalid memory accesses (out of bound read)
The associated alarms are shown hereafter:

```
[eva:alarm] border-router.c:189: Warning: out of bounds read. assert
\valid_read(&addr->u8[i]);
[eva:alarm] border-router.c:189: Warning: out of bounds read. assert
\valid_read(&addr->u8[(int)(i + 1)]);
```

These alarms cannot be effective because the associated code is within a loop, which exits when `i` is equal to `sizeof(uip_ipaddr_t)` where `uip_ipaddr_t` is a union defined in core/net/ip/uip.h as follows:

```
typedef union uip_ip6addr_t {
  uint8_t  u8[16];
  uint16_t u16[8];
} uip_ip6addr_t;
```

2) Accesses out of bounds index

These alarms are shown in what follows:

```
 [eva:alarm] border-router.c:192: Warning:   accessing out of bounds index.
assert 0 ≤ tmp;   (tmp from blen++)
[eva:alarm] border-router.c:192: Warning: accessing out of bounds index. assert
tmp < 128; (tmp from blen++)
…
 [eva:alarm] border-router.c:199: Warning:
  accessing out of bounds index. assert tmp_2 < 128;  (tmp_2 from blen++)
```

All these alarms are not effective because *blen* variable is used by the protothread (`PT_THREAD( )`) of the gateway (the protothread is not analyzed by EVA) so that each time the `ipaddr_add( )` function is called within the protothread, the `blen` value is set/reset to zero, as shown hereafter.

```
PT_THREAD(generate_routes(structhttpd_state *s))


{
   blen = 0;
  ADD("Neighbors<pre>");
  for(nbr = nbr_table_head(ds6_neighbors); nb r != NULL;  nbr =
nbr_table_next(ds6_neighbors, nbr))
{
    ipaddr_add(&nbr->ipaddr);
    …
    if(blen > sizeof(buf) - 45) {
      ..
      blen = 0;
    }
  }
```

3) Integer overflows (signed overflow)

The associated alarms relate to `blen` variable. These alarms are not effective, because `blen` variable is set/reset in the protothread (`PT_THREAD( )`) of the gateway. The protothread is not accessed by EVA.

3.2 Analysis of the C source part – Firmware Management on the 6LoWPAN node

3.2.1 Preparation

In order to use the Frama-C EVA plugin for the firmware management code analysis, the MPL code's EVA script has been modified to take into account the openmote-CC2538 HW platform's Flash. The new EVA script has then been applied to the different functions of the firmware management code.

```
frama-c -eva-slevel 100 -eva -main firmware_handling_process
wiseprom_node.c                                  \
       -cpp-extra-args=" -DCONTIKI=1"\
" -DCONTIKI_TARGET_OPENMOTE_CC2538=1"\
" -DNETSTACK_CONF_WITH_IPV6=1"\


.
." -I../../core/net/llsec/noncoresec"\
" -I../../platform/openmote-cc2538/"\
" -I../.."
#"   -DCONTIKI_VERSION_STRING=\"Contiki-3.x-3214-gedb3046\""
       \


       "$@"
exit
```

Figure 9: Fragment of EVA script for firmware management code analysis (case of FlashGet( ) function)

3.2.2 Verification process

For the firmware management code, the following table shows the set of functions that have been verified with EVA plugin. The verification of each MPL code function lasts about 2 to 3 seconds.

Function	Description
`void tcpip_handler(void)`	Write/read firmware data in/from the Flash memory, enable/disable interruptions, check firmware file integrity (CRC),
`static uip_ds6_maddr_t join_mcast_group(void)`	Configure the network interface for firmware data reception
`void firmware_handling_process(void)`	Manage the Flash vector table (erase/set/update), enable/disable interruptions, and call the reboot the system
`FlashGet(uint32_t)`	Return a 4-Byte value located in a given Flash memory address

Table 4: list of firmware management code functions verified with EVA plugin

3.2.3 Results

The results of EVA analysis for the firmware management code are summarized in the following table.

Function	Results
`void tcpip_handler(void )`	out of bounds read (1)
`static uip_ds6_maddr_t join_mcast_group(void)`	No warning
`void firmware_handling_process(void)`	out of bounds read (1)
`FlashGet(uint32_t)`	out of bounds read (1)

Table 5: list of alarms for the firmware management code based on EVA analysis

3.2.3.1 FlashGet ( ) function

The associated alarm (out of bound read) is not effective because the associate line of code (`return (HWREG(ui32Addr)`) relates to Texas Instrument's macro HWREG ( ) pointing to the address value indicated by `ui32Addr`.This address value is hard-coded in the 6LoWPAN node's firmware management code.

3.2.3.2 Firmware_handling_process ( ) function

The associated alarm relates to the call of function `FlashGet ( )`. This alarm is not effective, for the same reason as that mentioned in section 3.2.3.1.

3.2.3.3 join_mcast_group ( ) function

No alarm raised by EVA.

3.2.3.4 tcpip_handler ( ) function

The associated alarm relates to the call of function `FlashGet ( )`. This alarm is not effective, for the same reason as that mentioned in section 3.2.3.1.

3.3 Analysis of the Java source part – 6LoWPAN management server

3.3.1 Preparation

As mentioned in section 2.1.2, code analysis on the network manger's Java source code has targeted all the Activity templates, with the focus on detecting race conditions (e.g., variable sharing in a multi-threading scenario). A preliminary verification phase of the whole Java source code, in collaboration with KUL, led to identify the "Alarm.java" source file (cf. Figure 3 of section 2.1.2), as a potential source of possible race conditions.

This java source file implements a particular activity (to be understood in terms of "android concept") that is in charge of notifying alarm events issued from the 6LoWPAN node.

From the server perspective, alarms are notified under the form of incoming UDP packets. When the "Alarm" activity is shown, a listening thread is started to monitor incoming UDP packet on a specific UDP port.

This thread is created in the "`public void onCreate(Bundle savedInstanceState)`" function, and terminated in the "`protected void onDestroy( )`" function. Both functions are defined in the "Activity" class of the Android API, and may be overridden in the inheriting classes.

The best way to terminate the thread is to let the thread terminate itself when "`protected void onDestroy( )`" function is called. For this purpose, a global boolean variable named "goon" has been defined. This variable is permanently checked in the thread. `goon` variable is set to `false` in the "`protected void onDestroy( )`" so that once `onDestroy` is called, the previously created thread should gracefully terminate. The listening thread cannot block when receiving a UDP datagram, because a timeout value has been set on the UDP socket. This timeout guarantees that the change in the "goon" global variable will be captured in the thread shortly after the "`protected void onDestroy( )`" function has been called.

The Alarm.java source file has been fully annotated so that VeriFast analysis can be performed to detect any malfunctioning such as resource leakage or race conditions. The following extracts from the annotated Alarm.java file gives some hints about the Alarm activity structure and annotations.

```
public final class Alarm extends Activity implements Runnable
{
        Other globalvariable definitions removed
        boolean goon;
        Other global variable definitions removed
```

```
        public Alarm()
        //@ requires true;
        //@ ensures raw_state();
        {
        }


        /*@
        predicate raw_state() = this.raw_state(Activity.class)() &*& rxport |-> _ &*& goon
|-> _ &*&     myHandler |-> _ &*& mp |-> _ &*& alarmView |-> _;

        predicate state() =
        this.state(Activity.class)() &*& [_]goon |-> ?goon;

        @*/

        public void onCreate(Bundle savedInstanceState)
        //@ requires LooperThread(currentThread, nil) &*& raw_state();
        //@ ensures LooperThread(currentThread,
        {this}) &*& state() &*&    [_]LooperObject(currentThread, this,
activity_state(this));
        {
                implementation removed
                goon=true;
                //@ leak goon |-> _;
                Thread thread= new Thread(this);
                thread.start();
                //@ close state();
        }
```

```
       //@ predicate pre() = rxport |-> _ &*& [_]goon |-> ?goon &*& [_]myHandler |->
       ?myHandler_ &*& myHandler_ != null;
       //@ predicate post() = true;

       public void run()
        //@ requires pre();
        //@ ensures post();
       {
              implementation removed
                     if (!goon)
                     {
                            implementation removed
                     }

              implementation removed
       }

       protected void onDestroy()
       //@ requires Activity_onDestroy_called(currentThread, this, false)
       &*& LooperThread(currentThread, {this}) &*& state();
       //@ ensures Activity_onDestroy_called(currentThread, this, true)
       &*& LooperThread(currentThread, {this}) &*& state();
       {
              implementation removed
              goon = false;
              implementation removed
       }

}

final class MyHandler extends Handler
{
              implementation and annotations removed
}
```

3.3.2 Verification process

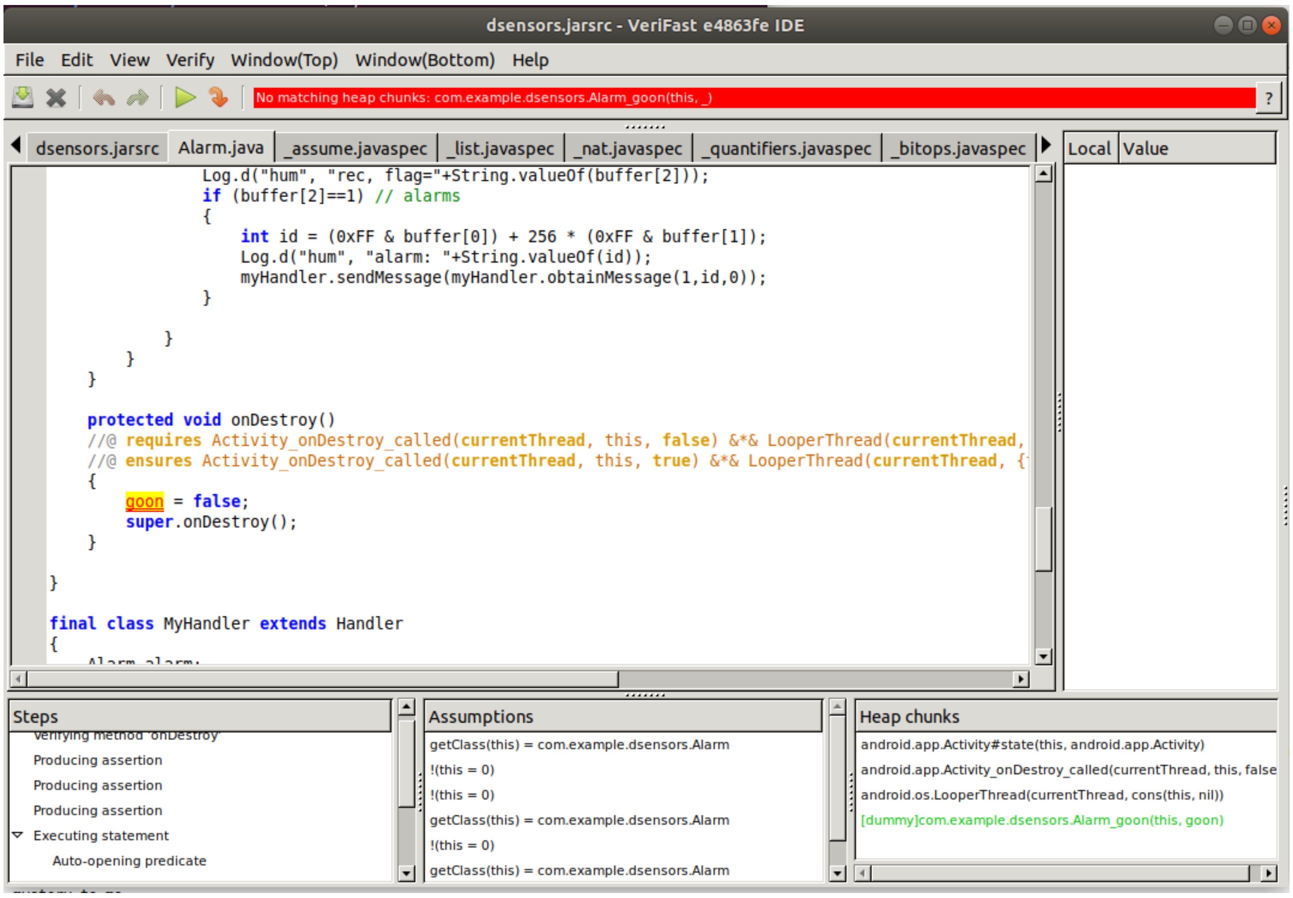
For the verification process of the "Alarm.java" source file, the following command has been used:

```
vfide -disable_overflow_check dsensors.jarsrc -runtime ../rt/rt.jarspec
```

where "`dsensors.jarsrc`" list all java and jar files needed for running verification and `rt.jarspec` points to a set of "javaspec" files that have been properly annotated to allow the verification process to deal with parts of the Android API . The verification lasted about 2 seconds and has been performed on an Intel Core i3 CPU m370 2.4 GHz processor.

3.3.3 Results

The result shown in figure 10 represents the graphical version of VeriFast (vfide) pointing to a race condition issue in the code, with the specific error message "`No matching heap chunks: com.example.dsensors.Alarm_goon(this, _)`".



```
No matching heap chunks: com.example.dsensors.Alarm_goon(this, _)
```

```
                Log.d("hum", "rec, flag="+String.valueOf(buffer[2]));
                if (buffer[2]==1) // alarms
                {
                    int id = (0xFF & buffer[0]) + 256 * (0xFF & buffer[1]);
                    Log.d("hum", "alarm: "+String.valueOf(id));
                    myHandler.sendMessage(myHandler.obtainMessage(1,id,0));
                }

            }
        }
    }

    protected void onDestroy()
    //@ requires Activity_onDestroy_called(currentThread, this, false) &*& LooperThread(currentThread,
    //@ ensures Activity_onDestroy_called(currentThread, this, true) &*& LooperThread(currentThread, {
    {
        goon = false;
        super.onDestroy();
    }

}

final class MyHandler extends Handler
{
```

```
Steps:
  Verifying method 'onDestroy'
  Producing assertion
  Producing assertion
  Producing assertion
▽ Executing statement
    Auto-opening predicate
```

```
Assumptions:
getClass(this) = com.example.dsensors.Alarm
!(this = 0)
!(this = 0)
getClass(this) = com.example.dsensors.Alarm
!(this = 0)
getClass(this) = com.example.dsensors.Alarm
```

```
Heap chunks:
android.app.Activity#state(this, android.app.Activity)
android.app.Activity_onDestroy_called(currentThread, this, false
android.os.LooperThread(currentThread, cons(this, nil))
[dummy]com.example.dsensors.Alarm_goon(this, goon)
```

Figure 10: verification of the Alarm.java source file with the graphical version of VeriFast

This outcome identifies a race condition in the source code that is due to the use of the global `goon` variable in the contexts of 2 different thread:

- an android thread in the set of threads use to manage the android User Interface
- the UDP listening thread.

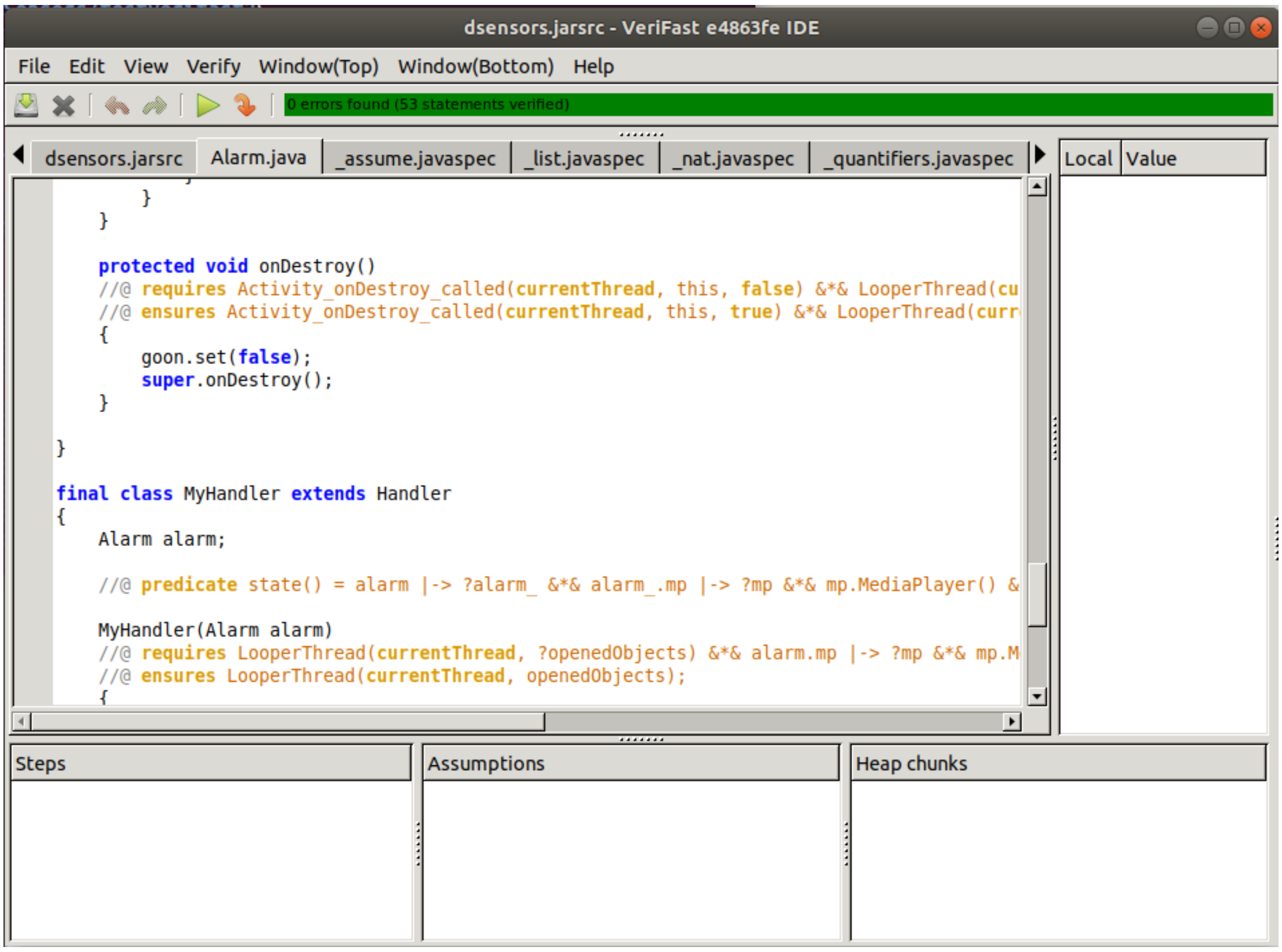
Though malfunctioning had not been experienced in the practical tests of the application, a malfunctioning risk exists.

The source file has been modified in order to replace the global `goon` variable of type `boolean` with a variable with the same name `goon` but of type `AtomicBoolean`.

The `AtomicBoolean` java class has precisely been proposed to circumvent race condition risks.

This modified version of "Alarm.java" has been re-annotated and re-verified with VeriFast software.

Figure 11 shows the graphical version of VeriFast (vfide) that is used to verify this modified version of Alarm.java. No error is detected in this case.



0 errors found (53 statements verified)

```
        }
    }

    protected void onDestroy()
    //@ requires Activity_onDestroy_called(currentThread, this, false) &*& LooperThread(cu
    //@ ensures Activity_onDestroy_called(currentThread, this, true) &*& LooperThread(curr
    {
        goon.set(false);
        super.onDestroy();
    }

}

final class MyHandler extends Handler
{
    Alarm alarm;

    //@ predicate state() = alarm |-> ?alarm_ &*& alarm_.mp |-> ?mp &*& mp.MediaPlayer() &

    MyHandler(Alarm alarm)
    //@ requires LooperThread(currentThread, ?openedObjects) &*& alarm.mp |-> ?mp &*& mp.M
    //@ ensures LooperThread(currentThread, openedObjects);
    {
```

Figure 11 second run of Alarm.java verification, after modification of the source

Chapter 4 Lessons Learnt

4.1 Training phase

The training phase on the software verification tools has taken an important part of the work effort in the analysis process of the 6LoWPAN management platform, since platform developers are not expert in formal methods. In particular, the annotation of the source code required several rounds of work, which was quite non-exhaustive, especially during the preliminary annotation phases.

4.2 ACSL/WP for Code analysis

The work on the ACSL/WP annotations for the MPL code has followed several rounds in order to improve the accuracy of the annotations and get effective analysis results. An example of difficulty in the annotation exercise lied in adapting simple ACSL examples of annotations [3][4], to the case of MPL code, where the code structure includes different types of data structures; many pointers, nested conditional segments, and loop iterations based on pointers. Tthe ACSL annotation work was typically non-exhaustive in the initial phase. It then followed an iterative/empirical approach (annotate, verify, re-annotate) to improve the accuracy of the annotations.

4.3 EVA for Code analysis

EVA analysis tool was very useful for detecting runtime errors when there are too many lines of codes (even with a few hundreds of lines), especially that some of those runtime errors may be due to very simple bugs like the one mentioned in section 3.1.3.2.

4.4 VeriFast for Java code analysis

Software verification with VeriFast allowed to identify risks that practical tests have not identified. Though more extensive practical tests would probably have revealed the risk, exhaustive tests are not achievable in practice. Software verification turned to be very useful for establishing the absence of malfunctioning risks, when combined with practical tests.

Chapter 5 Summary and Conclusion

This document discussed the different results of the source analysis associated to a number of critical functionalities of the CEA use case. This document also highlighted the lessons learnt from the use of different software verification tools (Frama-C WP, Frama-C EVA, and VeriFast).

WP plugin has been used for a particular part of the C code: MPL routing code to verify a number of code-specific properties, whereas EVA plugin has been used to identify potential runtime errors over the whole C code of the 6LoWPAN management platform.

The training phase on the software verification tools has taken an important part of the work effort in the analysis process of the 6LoWPAN management platform.

In addition, the work on the ACSL/WP annotations for the MPL code has followed several rounds in order to improve the accuracy of the annotations and get effective analysis results, whereas EVA analysis tool was very useful for detecting potential runtime errors. Also, Java source code verification with VeriFast allowed to identify risks that practical tests have not identified.

Chapter 6 List of Abbreviations

Abbreviation	Translation
6LoWPAN	IPv6 over Low-Power Wireless Personal Area Networks
EVA	Evolved Value Analysis
GW	Gateway
IoT	Internet of Things
LLN	Low power and Lossy channel Network
MPL	Multicast routing Protocol for Low power and lossy channel networks
WP	Weakest Precondition

Chapter 7 Bibliography

[1] J. Hui et al., "Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks”, IETF standard, RFC 6282, September 2011.

[2] J. Hui et al., “Multicast Protocol for Low-Power and Lossy Networks (MPL)”, Internet Standard, RFC 7731, February 2016.

[3] FRAMA-C WP, https://frama-c.com/wp.html

[4] Patrick Baudin et al., “WP Tutorial”, Online Book, 2012, http://frama-c.com/download/frama-c-wp-tutorial.pdf

[5] Allan Blanchard, “Introduction to C program proof with Frama-C and its WP plugin”, Online Book, June 2019, https://allan-blanchard.fr/publis/frama-c-wp-tutorial-en.pdf

[6] FRAMA-C EVA, https://frama-c.com/value.html

[7] VeriFast, https://github.com/verifast/verifast