CoMeMoV: Collaborative Memory Models for formal Verification

Grant: ANR-22-CE25-0018

D1.a: Selected case studies

Planned Date of Delivery	June 30, 2023
Actual Date of Delivery	September 21, 2023 (presentation); June 1, 2024 (complete report)
Deliverable Security Class	N/A
Editor	Axel FERREOL
Contributors	Téo BERNIER, Nikolaï KOSMATOV, Yani ZIANI

Contents

Executive summary

The objective of this report is to present case studies highlighting the challenges of applying deductive verification due to the memory model used by Frama-C.

This report is the result of two tasks led by TRT. First, Task 1.1 focused on selecting real-world security-critical case studies where some proof difficulties have been observed in the earlier proof attempts, and some new ones to be verified. They will be selected both from proprietary code and open-source code. Secondly, Task 1.2 focused on providing a code base illustrating the problems that need new memory modeling solutions and where the soundness of the verification is difficult to ensure with the currently available tools.

Introduction

This report is a result of work-package WP1 that focused on analysis of requirements for deductive verification of real-life industrial C code. Its results will guide the following technical work-packages. First, a set of real-world security-critical case studies was selected by TRT, including case studies where some proof difficulties have been observed in the earlier proof attempts, and some new ones to be verified. They include the JavaCard Virtual Mathine of Thales and the TPM2-TSS library. Based on the selected case studies, TRT created a list of representative code patterns (on a companion repository, deliverable D1.b) illustrating the problems with the currently available tools. These code patterns will be used by the other partners in technical work-packages.

Code patterns are shared between partners. All the following case studies were tested with Frama-C v27.1.

1 Heterogeneous pointer casts [1]

Heterogeneous pointer casts, a frequent operation in low-level efficient code like in [1], involve converting pointers of one type to another where the alignment requirements are different. Demonstrating the correctness of code involving such casts is crucial. However, the Frama-C Wp plugin currently cannot handle these casts. Recent efforts, notably in [1], have found ways to work around this limitation by rewriting heterogeneous pointer casts with equivalent expressions not containing such casts. Our goal is for Wp to automatically verify code with heterogeneous pointer casts without code rewriting.

1.1 Reading a heterogeneous cast pointer

A typical scenario involves reading from a cast pointer, as illustrated in Fig. 1. In this example, we have created a basic function that takes an `unsigned short` as input, writes it byte per byte into a global `unsigned char` array, casts the array to an `unsigned short` array, and returns its content.

Our objective is to demonstrate that this function correctly returns the value of the unsigned short provided as an argument.

The example in Fig. 1 was tested with Frama-C's default memory model. We noticed that Wp is unable to verify the specification. Indeed, Wp raises a warning related to *l.13* in Fig. 1: *Cast with incompatible pointers types (source: uint8*) (target: uint16*).*

The proof obligation generated by Wp (Fig 2) shows the limits of the *Typed* memory model which can not simultaneously see `array` as an `unsigned char` array and an `unsigned short` array, that is why it does not take into account the writings into `array` at *l.11* and *l.12* and sees the `unsigned short array` as a new array at address `w`.

1.2 Writing to a heterogeneous cast pointer

Heterogeneous pointer casts are also encountered in the target of an assignment expression, as demonstrated in Fig. 3. In this example, we have created a function that casts a global `unsigned char` array into an `unsigned short` array and then writes an `unsigned short` argument into it.

Our goal is to verify that the array indeed holds the value passed as an argument once the function completes.

The example in Fig. 3 was tested with Frama-C's default memory model. We also noticed that Wp is unable to verify the specification. Indeed,

[1] The examples described in this section are contained in the `heterogeneous_pointer_cast` subfolder of the companion repository (deliverable D1.b).

```
1  typedef unsigned char u8;
2  typedef unsigned short u16;
3
4  u8 array[2];
5  /*@
6      requires \valid(array+(0..1));
7      assigns array[0..1];
8      ensures G: \result == v;
9  */
10 u16 read_value(u16 v)
11 {
12     array[0] = v & 0xff;
13     array[1] = (v >> 8) & 0xff;
14     return ((u16 *)array)[0];
15 }
```

Figure 1: Mock function involving reading from a heterogeneous cast pointer. Note that the `u16` is written into the `u8` array respecting the little endian convention, as it is Frama-C's default architecture. This code is available in `heter_read.c`.

```
1 Goal Post-condition 'G':
2 Let x = Mint_0[shift_uint16(w, 0)].
3 Assume { Type: is_uint16(v) /\ is_uint16(x). }
4 Prove: x = v.
```

Figure 2: Proof obligation of the property `G` of function `read_value` of Fig. 1 with Frama-C's default memory model.

```
1 typedef unsigned char u8;
2 typedef unsigned short u16;
3
4 u8 array[2] ;
5 /*@
6     requires \valid(array+(0..1));
7     assigns array[0..1];
8     ensures G: *(u16*)array == v;
9 */
10 u16 write_value(u16 v){
11         *(u16*)array = v;
12         return 0;
13 }
```

Figure 3: Mock function involving writing to heterogeneous cast pointer. This code is available in `heter_write.c`.

```
1 Goal Post-condition 'G':
2 Let x = Mint_0[w <- v][w_1]. Assume { Type:
    is_uint16(v) /\ is_uint16(x). }
3 Prove: x = v.
```

Figure 4: Proof obligation of the property `G` of function `write_value` of Fig. 3 with FRAMA-C's default memory model.

WP raises a warning related to *l.8* and *l.11* in Fig. 3: *Cast with incompatible pointers types (source: uint8*) (target: uint16*).*

The proof obligation generated by WP in Fig 4 shows the limits of the *Typed* memory model which incorrectly gives different addresses (`w` and `w_1`) to the global variable `array`.

1.3 Casts in the specification [2]

The specification of code often involves pointer casts. This is all the more frequent as it occurs in the specification of libraries. A common example is seen in the `memcpy` function where the specification expresses properties over its arguments cast as `char *`, see Fig. 5. However, code may call `memcpy` with `unsigned char*` parameters, as shown in Fig. 6.

WP raises a warning about the cast: *Cast with incompatible pointers types (source: uint8*) (target: sint8*)* and is unable to verify the precondition `valid_src` of `memcpy`.

[2] A real life example can be found in `byte_level.c` and is a copy of `WP1/tpm2-tss-examples/byte_level_operations/byte_level.c`

```
1 /*@
2   predicate valid_read_or_empty{L}(void *s, size_t
       n) =
3     (empty_block(s) ∨ \valid_read((char *)s)) ∧
4     \valid_read((char *)s + (1 .. n - 1));
5  /*
6
7 /*@
8   requires valid_src: valid_read_or_empty(src, n);
9   ...
10 */
11 void *memcpy(void * restrict dest,
12         void const * restrict src, size_t n);
```

Figure 5: Part of the contract of `memcpy`. The parameter `src` is cast to `char *` in the requirement `valid_read_or_empty`. This code can be found in Frama-C default library: `FRAMAC_SAHRE/libc/string.h/memcpy`.

The proof obligation generated by Wp in Fig 7 shows the limits of the *Typed* memory model which incorrectly gives different addresses (`buff_0` and `w`) to respectively `(uint8*)buff` and `(sint8*)buff`.

2 BitFields [3]

Bitfields are extensively utilized in low-level and efficient code, as they offer a cost-effective means of storing information. For instance, in [1], bitfields are used to encode various flags. Verifying code involving bitfields is crucial. However, Wp is unable to deal with bitfields. Recent efforts, notably in [1], have addressed this limitation by introducing ghost variables for each bit and utilizing interactive scripts. Nonetheless, this approach entails annotation overhead and diminishes scalability, particularly on large code bases. Our objective is to enable Wp to automatically verify code containing bitfield structures.

A typical situation arises when one tries to set a field in a bitfield structure, as shown in Fig. 8. In this example, we have developed a function that initializes the first field of a bitfield with a value passed as argument.

We want to prove that the bitfield structure was correctly set.

The example in Fig. 8 was tested with Frama-C's default memory model. We noticed that Wp is unable to reason on bitfields. Indeed, as shown in the proof obligation (Fig. 9), Wp does not take into account the

[3] The examples described in this section are contained in the `bitfields` subfolder of the companion repository (deliverable D1.b).

```
1 /*@
2   requires \valid(offset) ∧ 0 ≤ *offset ≤ UINT8_MAX
      - sizeof(in);
3   requires buff_size > 0 ∧ \valid(&buff[0] + (0 ..
      buff_size - 1));
4   requires *offset ≤ buff_size ∧ sizeof(in) +
      *offset ≤ buff_size;
5   requires \separated(offset, buff);
6
7   assigns *offset, (&buff[*offset])[0..sizeof(in) -
      1];
8
9   ensures *offset ==  \old(*offset) + sizeof(in);
10  ensures \result == 0;
11 */
12 int uint32_Marshal(uint32_t in,uint8_t
     buff[],size_t buff_size,size_t *offset){
13  size_t  local_offset = 0;
14  if (offset ≠ NULL){local_offset = *offset;}
15  in = HOST_TO_BE_32(in);
16  memcpy(&buff[local_offset], &in, sizeof (in));
17  if (offset ≠ NULL){*offset = local_offset +
      sizeof (in);}
18  return 0;
19 }
```

Figure 6: Function from the TPM software stack, calling `memcpy`. The complete code is available in `byte_level.c`

```
1 Goal Definition (Unfold 'shift_sint8'):
2 Let m = Malloc_0[P_in_965 <- 1].
3 Let x = Mint_0[offset_0].
4 Assume {
5   Type: is_uint64_chunk(Mint_0) /\
       is_uint64(buff_size_0) /\ is_uint64(x).
6   (* Heap *)
7   Type: (region(buff_0.base) ≤ 0) /\
       (region(offset_0.base) ≤ 0) /\
8       linked(Malloc_0).
9   (* Residual *)
10  When: null ≠ offset_0.
11  (* Pre-condition *)
12  Have: (0 ≤ x) /\ (x ≤ 251) /\ valid_rw(Malloc_0,
       offset_0, 1).
13  (* Pre-condition *)
14  Have: (0 < buff_size_0) /\ valid_rw(Malloc_0,
       shift(buff_0, 0), buff_size_0).
15  (* Pre-condition *)
16  Have: (x ≤ buff_size_0) /\ ((4 + x) ≤
       buff_size_0).
17  (* Pre-condition *)
18  Have: offset_0 ≠ buff_0.
19  (* Assertion 'rte,mem_access' *)
20  Have: valid_rd(m, offset_0, 1).
21 }
22 Prove: valid_rw(m, shift(w, 0), 4).
```

Figure 7: Proof obligation of `valid_src` of `memcpy` called in Fig. 6 with Frama-C's default memory model.

```
1  struct __struct_bit {
2      unsigned char b0 : 1 ;
3      unsigned char b1 : 1 ;
4      unsigned char b2 : 1 ;
5      unsigned char b3 : 1 ;
6      unsigned char b4 : 1 ;
7      unsigned char b5 : 1 ;
8      unsigned char b6 : 1 ;
9      unsigned char b7 : 1 ;
10 };
11
12 struct __struct_bit flags;
13
14 /*@
15     assigns flags.b0;
16     ensures G: flags.b0 == (v & 1);
17 */
18 int set_b0(unsigned char v){
19     flags.b0 = v;
20     return 0;
21 }
```

Figure 8: Mock function involving a bitfield operation. Note that FRAMA-C does not specify the order of bitfields in memory. This code is available in `bf.c`.

```
1 Goal Post-condition 'G':
2 Assume { Type: is_uint8(v). }
3 Prove: land(1, v) = v.
```

Figure 9: Proof obligation of the property `G` of function `set_b0` of Fig. 8 with FRAMA-C's default memory model.

number of bits of the field `b0` on the assignment of *l.19*, instead it considers `b0` as an `unsigned char`.

We also tried with the Hoare memory model of FRAMA-C unsuccessfully.

3 Union structures [4]

Unions are also frequently used in low-level code because they allow multiple types to be stored at the lowest memory cost. However, WP is unable to reason on unions. Our objective is to enable WP to automatically verify code containing union structures.

A typical situation arises with a union structure comprising a bitfield and a byte. In this scenario, the bits are set via the byte representation, and the bits are accessed through the bitfield view, as shown in Fig. 10. In this example, we have developed a function that initializes the union through its byte-view with a value provided as argument.

Our goal is the verify that the bits (fields) of the bitfield view match the value passed as an argument.

The example in Fig. 10 was tested with FRAMA-C's default memory model. We noticed that WP is unable to reason on union fields. Indeed, it raises a warning: *Accessing union fields with* WP *might be unsound*. In the proof obligation (Fig. 11), we can observe that WP does not take into account the memory view of the union, as it did not add any property related to the assignment *l.27* in the context of the proof.

We also tried with the Hoare memory model of FRAMA-C unsuccessfully.

4 Nested structures [5]

Many industrial codes use nested structures, where structures are nested within other structures. In such contexts, a frequent objective is to demonstrate the preservation of an array of these nested structures. However, automatic provers fail to verify such a goal, particularly with deeply nested

[4] The examples described in this section are contained in the `union_structures` subfolder of the companion repository (deliverable D1.b).

[5] The examples described in this section are contained in the `nested_structures` subfolder of the companion repository (deliverable D1.b) and are copies from `WP1/tpm2-tss-examples/struct_union`.

```
1 struct __struct_bit {
2     unsigned char b0 : 1 ;
3     unsigned char b1 : 1 ;
4     unsigned char b2 : 1 ;
5     unsigned char b3 : 1 ;
6     unsigned char b4 : 1 ;
7     unsigned char b5 : 1 ;
8     unsigned char b6 : 1 ;
9     unsigned char b7 : 1 ;
10 };
11
12 union __union_flags
13 {
14     struct __struct_bit bit;
15     unsigned char byte;
16 };
17
18 union __union_flags flag;
19
20 /*@
21     assigns flag;
22     ensures flag.bit.b0 == v % 2;
23     ensures G: flag.bit.b1 == (v>>1) % 2;
24     //...
25 */
26 int set_byte(unsigned char v){
27     flag.byte = v;
28     return 0;
29 }
```

Figure 10: Mock function involving operations over a union. This code is available in `union.c`.

```
1 Goal Post-condition 'G':
2 Let a = flag_0.F2___union_flags_bit.
3 Let x = a.F1___struct_bit_b1.
4 Assume {
5   Type: is_uint8(v) /\
        is_uint8(a.F1___struct_bit_b0) /\ is_uint8(x).
6   (* Heap *)
7   Type: IsU2___union_flags(flag_0).
8 }
9 Prove: x = (lsr(v, 1) % 2).
```

Figure 11: Proof obligation of the property `G` of function `set_byte` of Fig. 10 with Frama-C's default memory model.

structures, as highlighted in [2]. Recent research, as cited in [2], addressed this challenge by employing manual scripts to assist the prover. Our aim is to enable automatic verification of these goals.

A typical example from [2] consists in a function modifying one cell within an array of a nested structure. We want to prove that, aside from the specific location where the cell was modified, the array remains unchanged.

While the automatic provers struggled to prove the goal generated by Wp, employing a script tactic proved much more effective. This tactic involves recursively unfolding the equalities and verifying them individually through the automatic prover. The efficiency stems from eventually applying the automatic prover on sub-goals rather than directly on the main goal.

5 Dynamic memory allocation [6]

Dynamic memory allocation is frequently used in code. While the contracts of dynamic memory allocator functions can be precisely expressed by ACSL clauses (see Fig. 12), their specification includes clauses that are not supported by Wp. Recent work, as referenced in [2], addressed this issue by representing the heap as a global array in the code. Our goal is for Wp to handle such clauses effectively.

The `malloc` specification in Fig. 12 employs the `\fresh{Old, Here} (\result,\old(size))` clause at *l.8*, which is not supported by Wp, as indicated by the error message: *Allocation, initialization and danglingness not yet implemented.*

[6] The examples described in this section are contained in the `dynamic_memory_allocation` subfolder of the companion repository (deliverable D1.b) and are copies from `WP1/tpm2-tss-examples/dynamic_allocation`.

```
1 /*@ assigns __fc_heap_status, \result;
2     assigns __fc_heap_status \from size,
         __fc_heap_status;
3     assigns \result \from (indirect: size),
         (indirect: __fc_heap_status);
4     allocates \result;
5
6     behavior allocation:
7       assumes can_allocate: is_allocable(size);
8       ensures allocation: \fresh{Old,
           Here}(\result,\old(size));
9       assigns __fc_heap_status, \result;
10      assigns __fc_heap_status \from size,
           __fc_heap_status;
11      assigns \result \from (indirect: size),
           (indirect: __fc_heap_status);
12
13    behavior no_allocation:
14      assumes cannot_allocate: !is_allocable(size);
15      ensures null_result: \result == \null;
16      assigns \result;
17      assigns \result \from \nothing;
18      allocates \nothing;
19
20    complete behaviors no_allocation, allocation;
21    disjoint behaviors no_allocation, allocation;
22 */
23 extern void *malloc(size_t size);
```

Figure 12: Contract of `malloc` in ACSL. This code can be found in FRAMA-C default library: `FRAMAC_SAHRE/libc/stdlib.h/malloc`.

6 Separation of the allocation status of heap and local variables [7]

The *Typed* memory model of Wp uses single tables for different types, regardless of the variable scope. Consequently, global and local variables are represented within the same tables. This causes predicates over global variables to be affected by modification of local variables, hindering simple proofs. Our goal is to enable automatic proofs of predicates involving global variables when they are unaffected by local code.

A typical example, see Fig. 13, illustrates this issue. A predicate over a global variable is set as a precondition, and we want to check its validity at the beginning of a function. Inside the function, a local variable is referenced. Although the predicate is trivially true initially, provers fail to verify it.

A workaround involves rewriting the code to encapsulate problematic variables within an anonymous block and define intermediary variables to keep the function body updated with the changes inside that block, see Fig. 14. Another workaround consists in writing lemmas stating the predicate. It scales up well but requires significant effort in Coq to prove.

Our goal is for Wp to handle this case automatically.

This failure occurs because the predicate involves a `\valid` clause, making it dependent on the allocation table. Additionally, any new variable passed by reference within the function body will have an entry in the allocation table (provided that it is used in properties to be proved), which then gets modified. Consequently, the initial predicate depends on a new, modified allocation table, complicating otherwise trivial proofs.

7 Stack example [8]

Recently, in the context of [1] we worked on verifying code that implements a stack. We encountered several challenges during the proof of its specifications, which we illustrated with a toy example, where words are represented by stamps and function contexts by collections of stamps.

In this example, the code allows adding or removing stamps to or from a collection of stamps and adding or removing collections. Adding or removing a stamp is analogous to pushing or popping from the stack, while adding or removing a collection is similar to calling or returning from a function. To

[7] The examples described in this section are contained in the `separation_variables` subfolder of the companion repository (deliverable D1.b). A real life example is `allocation_table.c` and is a copy from `WP1/tpm2-tss-examples/allocation_table/allocation_table.c`.

[8] The examples described in this section are contained in the `stack` subfolder of the companion repository (deliverable D1.b). Especially, the code of the stack can be found in `stack.c`, and its rewritten version (enabling the proof to succeed) in `stack_rewritten.c`.

```
1 /*@
2 requires linked_ll(rsrc_list, NULL,
     node_to_ll(rsrc_list, NULL));
3 ...
4 */
5
6 int dummyCaller(LIST * rsrc_list, LIST ** out_node)
7 {
8   /*@ assert linked_assert :
9     linked_ll(rsrc_list, NULL,
          node_to_ll(rsrc_list, NULL));*/
10  // Non proved
11  int r;
12  LIST *test_node;
13  r = dummyCallee(rsrc_list, &test_node);
14  return r;
15 }
```

Figure 13: Mock function where the local variable `test_node` is in conflict with the arguments. The complete is available in `sep_var.c`.

```
1 /*@
2 requires linked_ll(rsrc_list, NULL,
     node_to_ll(rsrc_list, NULL));
3 ...
4 */
5
6 int dummyCaller(LIST * rsrc_list, LIST ** out_node)
7 {
8   /*@ assert linked_assert :
9     linked_ll(rsrc_list, NULL,
          node_to_ll(rsrc_list, NULL));*/
10     // Proved
11    int r;
12  {
13    LIST *test_node = NULL;
14    r = dummyCallee(rsrc_list, &test_node);
15  }
16  return r ;
17 }
```

Figure 14: Rewriting version of the function in Fig. 13 to avoid conflicts between local variables and arguments. The complete code is available in `sep_var_rewriting.c`.

```
1 typedef struct RegistryCellTag {
2   uchar CollectionSize; //size of the collection
3   uchar CollectionMaxSize; //maximum size of the
       collection
4 } RegistryCell ;
```

Figure 15: Definition of `RegistryCell` type from `stack.c`.

keep track of previous collections, a second stack (the registry stack) stores local information (registry cells) about the already stacked collections of stamps enabling to restore the context of a stored collections when necessary.

In that example, the registry cell is represented by a structure with two `uchar` fields (see Fig. 15): `CollectionOffset` stores the size of the collection and `CollectionMaxSize` stores the maximum size of the collection. Since each collection directly follows the previous one, knowing the beginning of the latest collection and the size of each collection allows us to locate the start of every collection.

The registry cell is read or written into an array of `ushort` (the registry stack) via a pointer cast, see Fig. 16. As Wp does not support heterogeneous pointer casts, code rewriting is necessary to complete the proof, see Fig 17. In that version, the first field is stored in the first byte and the field in the second byte, we kept that convention for reading too.

Compared to Fig. 16, both writings are made simultaneously in Fig. 17 to facilitate the proof; otherwise (as shown in Fig 18) Wp struggles. In addition, few assertions are necessary to conduct the rest of the proof correctly, see Fig 19.

Ideally, we would like to avoid such transformations.

```
/*Writing*/
((RegistryCell*) NextFreeRegister)->CollectionSize =
 NextFreeCell - CurrentCollectionStart;
((RegistryCell*)
    NextFreeRegister)->CollectionMaxSize =
 CurrentCollectionMaxSize;
/*...*/

/*Reading*/
CurrentCollectionStart -=
 ((RegistryCell*)NextFreeRegister)->CollectionSize;
CurrentCollectionMaxSize =
 ((RegistryCell*)NextFreeRegister)->CollectionMaxSize;
```

Figure 16: Writing and reading a registry cell in the registry stack via pointer cast. `NextFreeRegister` points to the next registry cell available in the registry stack. These are code excerpts from `stack.c`.

```
/*Writing*/
*NextFreeRegister = (NextFreeCell -
    CurrentCollectionStart)
 + (CurrentCollectionMaxSize << 8);
/*...*/

/*Reading*/
CurrentCollectionStart -= *NextFreeRegister & 0xff;
CurrentCollectionMaxSize = *NextFreeRegister >> 8;
```

Figure 17: Rewritten versions of Fig. 16 without pointer cast. These are code excerpts from `stack_rewritten.c`.

```
*NextFreeRegister = (NextFreeCell -
    CurrentCollectionStart)
 + ((*NextFreeRegister) & 0xff00);
*NextFreeRegister = ((*NextFreeRegister) & 0xff)
 + (CurrentCollectionMaxSize << 8);
```

Figure 18: Rewritten version of the writing statement of Fig. 16 where Wp struggles. The complete code is available in `stack_struggle.c`

```
1 //@ assert (*NextFreeRegister & 0xff) ==
      current_collection_size;
2 //@ assert (*NextFreeRegister >> 8) ==
      CurrentCollectionMaxSize;
```

Figure 19: Additional assertions necessary to prove the rewriting in Fig. 17 from `stack_rewritten.c`.

List of Figures

References

[1] Djoudi, A., Hána, M., Kosmatov, N.: Formal Verification of a JavaCard Virtual Machine with Frama-C. In: Proc. of the 24th International Symposium on Formal Methods (FM 2021). LNCS, vol. 13047, pp. 427–444. Springer (2021)

[2] Ziani, Y., Kosmatov, N., Loulergue, F., Gracia Pérez, D., Bernier, T.: Towards formal verification of a TPM software stack. In: Proc. of the 18th International Conference on integrated Formal Methods (iFM 2023). LNCS, vol. 14300, pp. 93–112. Springer (Nov 2023)