Technical report — Cerise case study

Bastien Rousseau (ENS Rennes)

August 5, 2022

This file proposes exercises to learn how to use Cerise in Coq. In particular, it proposes exercises to train the definition of specification of a program, prove the specification with the program logic, and use the logical relation to interact with the unknown code. Moreover, it proposes different alternatives to locally encapsulate a code. For each exercise, the specification and the proof have to be implemented in Coq, except when it is explicitly ask to define them of paper.

1 Secret buffer

In this section, we explore the different approaches to implement and specify a program interacting with a memory buffer. The general principle is the following: (1) the program stores a secret value in the buffer, (2) it derives a capability which restricts the permission of the buffer, such that it does not have the authority over the secret data anymore, (3) passes the control flow to an adversary, with a capability to a secret part of the buffer. We propose different version of the case study.

For each version, we define the instructions corresponding to the description of the program, specify the program and prove the specification, at the level of the program logic.

1.1 Buffer

The first version of the program is the base program. Figure 1 shows the instructions of the base code. The program assumes that the register r_1 contains a capability pointing to a memory buffer, and the register r_{30} contains an adversary word. The program writes a secret value in the buffer, restricts the capability of the buffer and pass the control flow to an adversary.

```
init:
lea r1 secret_off
store r1 secret
getb r2 r1
gete r3 r1
add r2 (secret_off + 1)
subseg r1 r2 r3
jmp r30
end:
```

Figure 1: Program restrict buffer

The memory buffer is sensitive to the buffer overflow attacks, but the capability ensures the integrity of the secret value. In the end, we want to formally prove the integrity of the secret data throughout the complete execution of the machine, especially after the execution of the adversary. As a first step, we need to specify the known program, from the first instruction to the last one, before passing the control flow to the adversary.

As the program assumes some hypotheses on the state of the registers before the execution of the program, the precondition of the specification has to express these hypotheses in terms of the program logic.

Exercise 1 — Buffer specification

Complete the following specification on paper.

$$\vdash \left\{ \begin{array}{l} \mathsf{pc} \Mapsto (p_{\mathsf{pc}}, b_{\mathsf{pc}}, e_{\mathsf{pc}}, \text{init}) * \\ \mathsf{r}_1 \Mapsto (p, b, e, b) * \\ \mathsf{r}_{30} \Mapsto w_{adv} * \\ [b, e) \mapsto [0 \cdots 0] * \\ \cdots \end{array} \right\} \rightsquigarrow \left\{ \begin{array}{l} \mathsf{pc} \Mapsto (p_{\mathsf{pc}}, b_{\mathsf{pc}}, e_{\mathsf{pc}}, \text{init} + 6) * \\ \cdots \end{array} \right\}$$

The next step of the specification is to prove the complete specification, *i.e.,* until the machine halt or fail. As a first step, we assume that the adversary code contains only the instruction Halt.

Exercise 2 — Adversary halts

We assume the adversary word is a capability pointing to an adversary program, but the code actually contains only the instruction Halt.

Complete the specification on paper, write the lemma corresponding to the specification in Coq and prove it.

$$\vdash \left\{ \begin{array}{l} \mathsf{pc} \Mapsto (p_{\mathsf{pc}}, b_{\mathsf{pc}}, e_{\mathsf{pc}}, \text{init}) * \\ \mathsf{r}_1 \Mapsto (p, b, e, b) * \\ \mathsf{r}_{30} \Mapsto \cdots * \cdots \end{array} \right\} \rightsquigarrow \bullet$$

Finally, we consider an adversary with unknown code.

Exercise 3 — Unknown adversary

The adversary word is unknown. The program logic and the WP rules are not enough to prove the specification with the unknown, adversary code. Explain why. What is the theorem required to reason with the unknown code ?

Assuming the adversary word w_{adv} be *safe-to-share*, and the memory buffer is initialized with zeroes, complete and prove the specification in Coq. Why does it make sense to assume that the adversary word is *safe-to-share* ?

$$\vdash \left\{ \begin{array}{l} \mathsf{pc} \Mapsto (p_{\mathsf{pc}}, b_{\mathsf{pc}}, e_{\mathsf{pc}}, \text{init}) * \\ \mathsf{r}_1 \Mapsto (p, b, e, b) * \\ \mathsf{r}_{30} \Mapsto w_{adv} * \mathcal{V} w_{adv} * \end{array} \right\} \rightsquigarrow \bullet$$

Exercise 4 — Local encapsulation

We want to share the program with an adversary. We encapsulate the program into a sentry-capability.

1. Define the theorem which ensures that the program is safe to share.
2. What are the required hypothesis ? Explain why.
3. Currently, the theorem is not provable. Explain why.

In the two next sections, we explore different approaches to able the encapsulation of the program into a sentry capability that is *safe-to-share*.

1.2 Closure buffer

We recall that the basic program requires the capability pointing to the buffer in a register before the execution of the program. It enforces a constraint on the execution context. We want to avoid this constraint, because the *safe-to-execute* relation requires a context without any condition.

In order to avoid the constraint on the execution context, we need to obtain the buffer capability during the execution time. One way to do that is to store the capability directly in the memory closure of the program, in a *data* section. In this way, up-front the basic program, we add the instructions which load the capability in the data section of the program. In this configuration, each call to the program stores the secret value at the exact same address in the memory.

Exercise 5 — Closure load

Define the code that loads the capability of the buffer. What is the assumption on the data section of the program ?

Exercise 6 — Closure specification

A good practice when specifying such modular program is to specify each part of the program independently, and to use the sequencing rule to stick the specification together. In this case, we have already proved the specification of the second part of the program in Section 1.1. It only remains to prove the initialization of the context, *i.e.,* the loading instructions.

1. Specify and prove the specification of the loading instructions of the buffer capability.
2. Using the sequencing rules, prove the complete safe execution of the full program.

Exercise 7 — Closure local encapsulation

Define and prove the theorem which ensures that the sentry capability of the closure version of the program is *safe-to-share*.

1.3 Allocation of the buffer

We propose another way to avoid the constraint on the execution context. Instead of storing the capability pointing to the buffer in the *data* section of the program, as we have done in Section 1.2, we dynamically allocate a memory buffer in which the program will store the secret value. The program is fundamentally different from the closure of the buffer. Indeed, on the contrary of the previous way, each call of the program allocates a new memory buffer, and thus the secret value is stored in a different address than the previous calls.

Exercise 8 — Allocation version

The size of the buffer is arbitrary. What is the hypothesis required that relate the size of the buffer and the secret offset ?

1. Define the new program using the allocation, specify and prove that the program executes safely and completely.
2. Prove that the sentry-capability of the allocation-version of the program is safe to share.

Hint: Use the specification of the *malloc* macro, which requires a linking table.

1.4 Call adversary — A full case study

We propose an exercise that use the call macro. It is an alternative of the buffer program, where the control flow is passed to the adversary with the calling convention, instead of the *jmp* instruction. The program splits the memory buffer in two parts: the first addresses are secrets, and store the secret value; and the public addresses, after the secret data. The program passes a capability giving access to the public part, as the previous examples already did, but it also derives a capability giving access to the secret part of the buffer, which is protected from the adversary thanks to the local encapsulation provided by the calling convention. Moreover, we use the assert macro after the call, to prove that the integrity of the secret data is not compromised.

Figure 2 shows the full list of instructions of the program. The instructions from the line 8 to 17 actually correspond to the buffer program in Section 1.1.

Exercise 9 — Call variant of the sub-buffer Define the full specification of the call variant of the program, and prove that the program executes safely and completely. Hints: Use the specification of the *call* macro and the *restore_locals*.

The ultimate step in the properties we are studying is usually to prove them at the bare-level of the operational semantic. Iris provides an adequacy theorem to translate a property of the program logic into a theorem expressed in terms of the operational semantic. On top of the adequacy theorem, Cerise provides different templates in order to simplify its use. In particular, Cerise provides a specialized

```
1   init:
2   ; allocates a region
3   malloc size ; allocation of the buffer
4   mov r7 r1 ; mov the cap of the buffer in r1
5   mov r1 0 ; clear r1
6   mov r7 r8 ; duplicate the buffer cap
7
8   ; stores a secret data in the newly allocated region
9   lea r1 secret_off
10  store r1 secret_val
11
12  ; derives 2 capabilities for both the secret and the public part
13  ; public
14  getb r2 r7
15  gete r3 r7
16  add r2 (secret_off + 1)
17  subseg r7 r2 r3 ; r7 -> ( pc_p, secret+1, e_mem, secret )
18
19  ; secret
20  getb r2 r8
21  getb r3 r8
22  add r3 (secret_off + 1)
23  subseg r8 r2 r3 ; r8 -> ( pc_p, b_mem, secret+1, secret )
24
25  ; calls the adversary and restore the locals after the call
26  call r30 [r8] [r7] ; r8 protected , r7 argument
27  restore_locals r2 [r8]
28
29  ; asserts the integrity of the secret data
30  ; load the value of the secret address in r4
31  load r2 r2
32  lea r2 secret_off
33  load r4 r2
34  ; put the secret value in r5
35  mov r5 secret_val
36  assert r4 r5
37
38  ; halts
39  halt
40  end:
```

Figure 2: Instructions of the call variant of the sub-buffer

version of the adequacy theorem to prove properties established on dynamically allocated memory, using the *assert* macro.

Exercise 10 — Integrity at the operational semantic

1. Why wasn't it possible to apply the adequacy theorem on the previous version of the exercises ?
2. Using the Cerise template of the adequacy theorem, define the end-to-end theorem that ensure the integrity of the secret data throughout the complete execution of the machine. Prove the theorem.

1.5 Read only

Another way to ensure the integrity of a data is to restrict the permission of the capability. Thus, instead of restricting the range of validity of the capability, we can only restrict the permission into RO, such that the adversary code cannot write in the buffer. In this case, the adversary can read the data (and the content of the whole buffer), but cannot modify its value.

Exercise 11 — Read only version, Base

Define the program that correspond to the Read-Only version of the Section 1.1. Specify the program until the *jmp* to the adversary code, and prove the specification. Hint: Assume that the register r_1 contains the capability pointing to the buffer.

Exercise 12 — Read only version, Closure

1. Define the program that correspond to the Read-Only version of the Section 1.2 (*i.e.,* the capability of the buffer is in the closure of the PCC).
2. Specify and prove the complete and safe execution of the program
3. Prove that the sentry-capability of the read-only version of the program is *safe-to-share*.

2 Counter routine with registers

In this section, we propose to explore an alternative of the counter library. In the contrary of the previous counter library in Cerise, the counter is not stored in the memory anymore, but only the registers. The program leverages the local state encapsulation of the *call* macro to save state of the counter, and to protect him against the adversary. Indeed, it will ensures that the code is safe, and the value of the counter is safely saved in the stack.

The assembly code of the program is shown in Figure 3. It is separated in two parts: the initialization code and the increment routine.

```
init:
mov r7 0
mov r9 0
mov r8 pc
lea r8 2                    ; r8 points to call_label

call_label:
call r30 [r7 , r8] [r9]     ; jmp to r30, [r7;r8] local state, r9 parameter

incr_label:
restore_locals r2 [r8 , r7]
add r7 r7 1
mov r9 r7
jmp r8                      ; jumps to call_label

data: (RO, link, link+1, link)
link: (E, bm, em, bm)
end:
```

Figure 3: Program counter variant

The labels `init`, `call_label` and `incr_label` denotes addresses in the memory. r_7 contains the value of the counter internal to the library, r_9 contains the value of the counter exposed to the adversary code and r_8 contains a pointer the label *call_label*. The initialization part puts the counter to 0 and a capability pointing to the *call_label* label in r_8, and then jump into the adversary code using the call macro. The increment part restores the value of the counter from the locals and increments the counter. It then uses the call macro to jump to the adversary code. The combination of the macros `call` and `restore_locals` ensures that the register containing the counter is always the same. `restore_locals` enforces r_7 to be the register containing the counter.

We can see this piece of code in 2 ways:

- A library that initializes a counter, and gives an increment function to the adversary;
- An infinite loop that call an adversary code at each round (not necessarily the same), increment the counter and give the value to the adversary.

Exercise 13 — Loop

The second interpretation highlights the loop in the routine. What reasoning principle do we have to use to prove a specification of a program that contains a loop ?

The calling convention does not enforces the well-bracket control-flow, allowing the adversary code to never re-enters in the loop. However, if the incrementation routine is called by an adversary, the activation record enforces the control flow to do a new loop tour.

Exercise 14 — Specification counter

1. Define and prove the specification of the initialization part.
2. Specify and prove the specification of the increment part, in particular that it safely and completely execute.
3. Prove that the sentry-capability of the program is *safe-to-share*.

Hint: as highlighted in Exercise 13, the proof of the specification of the increment part requires the use of the Löb induction.