Safety RAM: Protecting variables with one's complement

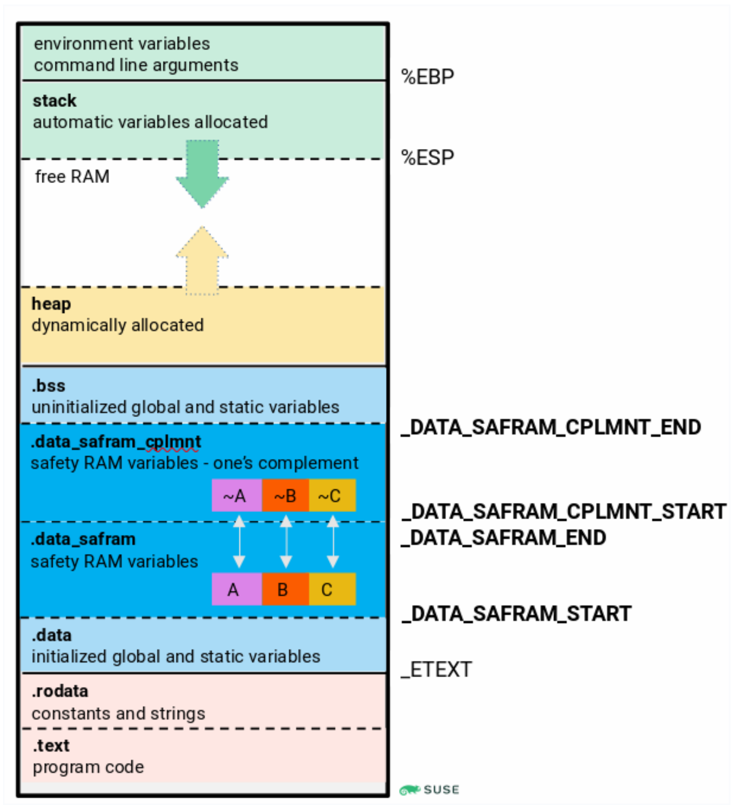
Protecting variables with one's complement

Firstly please read the Safety RAM: Protecting memory section with checksum blog post if you haven't as certain aspects won't be re-explained.

The problem with previous CRC based method was that there shall be no parallelism or interruption during executing of critical section by others, possibly non-safe, processes. So to deal with this limitation we can try to check consistency of the memory every time we read and protect it every time we write. Nice and simple. Obviously checking and protecting the whole safety memory region using checksum every time we read or write would be a massive overkill. Instead we could try to store the same variable in 2 distinct places in memory and check if those are matching when we are reading. But storing the exactly the same value is not smart enough. It would not help us much in likely cases when e.g. the entirely memory blocks are erased. We would not detect any errors if storing the same variable twice in this case. Luckily we can use one's complement for this purpose. Basically we swap 0 and 1 when storing the second complement value. Please also remind yourself that these mechanism are intended against random faults, not malicious cyber attack.

Now let's get our hands dirty. All the source code shown here (and more) can be found on GitHub here.

The logical first step is to define the 2 memory regions as shown on this figure:



This we can achieve by this linker script shown below. Note that unlike the checksum protection we don't need leave any space for the checksum as there is no checksum. The price for that is that we'll need twice as memory.

data_safram.ld - Linker script - define custom memory section

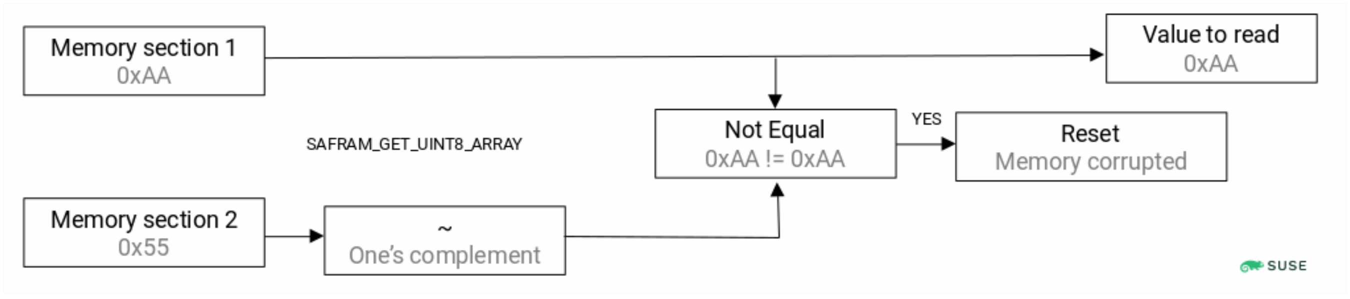
```
SECTIONS
{
    .data_safram : {
        _DATA_SAFRAM_START = .;
        *(.data_safram);
        _DATA_SAFRAM_END = .;
    }

    .data_safram_cplmnt : {
        _DATA_SAFRAM_CPLMNT_START = .;
        *(.data_safram_cplmnt);
        _DATA_SAFRAM_CPLMNT_END = .;
    }
}
INSERT AFTER .data
```

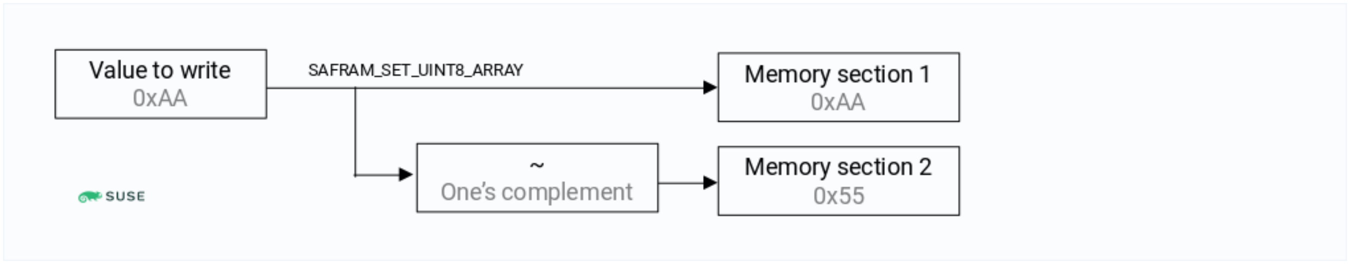
Of course one can take this further and place these section take into account the hardware architecture so e.g. these end up on different chips.

Now we need to check and protect the variables on every single write and read performed.

During reading we basically perform following operation to check the value:



In similar way, we write twice to protect it:



In our example memory section 1 is ".data_safram" and memory section 2 is ".data_safram_cplmnt".

But wait a minute, so I need to do that every time I store or read a value - that's painful and for sure buggy if one needs to write that manually every single time. So let's simplify it by using read/write C macros:

safram_cplmnt.h - Write and read macros for safety variables

```
#define SAFRAM_SET_UINT8_ARRAY(_name, _idx, _value) \
    do { \
        (_name[_idx]) = (_value); \
                (_name##_cplmnt[_idx]) = FCN_UINT8_CPLMNT(_value); \
    } while(0)

#define SAFRAM_GET_UINT8_ARRAY(_name, _idx, _failcall) \
        ((_name[_idx]) == FCN_UINT8_CPLMNT(_name##_cplmnt[_idx])) ? ( _name[_idx]) : \
            (_failcall, (_name[_idx]))
```

The one's complement function is defined as:

safram_cplmnt.h - One's complement macro

```
#define FCN_UINT8_CPLMNT(_value) ((uint8_t) ~(_value))
```

And as as this setup relies on specific names of complementary variables, we can also define C macros to help with the definition:

safram_cplmnt.h - Definition macros for safety variables

```
#define SAFRAM_DEF_UINT8_ARRAY(_name) __attribute__((section(".data_safram"))) uint8_t _name[]
#define SAFRAM_DEF_UINT8_ARRAY_CPLMNT(_name) __attribute__((section(".data_safram_cplmnt"))) uint8_t
_name##_cplmnt[]
```

Note that the example defines these macros only for arrays of UINT8 but similar macros can be created for any other basic type. The only catch could be with floating values where we need to convert their bit representation to unsigned integer when storing and checking the complement.

Then we can use the macros in our main program.

th_safram_cplmnt.c - Safety variables with complements

```
SAFRAM_DEF_UINT8_ARRAY(testarray) = {255, 254, 253, 252, 251, 250, 249, 248};
SAFRAM_DEF_UINT8_ARRAY_CPLMNT(testarray) = {0, 1, 2, 3, 4, 5, 6, 7};

void periodic_task(char **argv, uint8_t count)
{
        /* No specific safety critical section */
    uint8_t value;
    for(uint8_t i = 0; i < 8; i++) {
        value = SAFRAM_GET_UINT8_ARRAY(testarray, i, restart(i));
        if (i == 1) {
            printf("testarray[%u]=0x%02X, one's complement=0x%02X\n", i, value, FCN_UINT8_CPLMNT(value));
            value += i;
            SAFRAM_SET_UINT8_ARRAY(testarray, i, value);
        }
    }
        sleep(1);
}
```

Then we can use the same memory corruption program as in here to emulate memory corruption. The critical section is still there but limited in size. The critical moment happens when we check consistency and return the safety variable value as this operation is not atomic in this implementation (e.g. our values still could get corrupted after the memory check is evaluated and thus wrong value is returned without detection). Yet the probability of this happening is much lower as variables are protected and checked as we go.

Process 1 - th_safram_cplmnt

```


./th_safram_cplmnt
This is an example of protecting RAM variable with one's
complement! Run #0!
pid=30687
.data_safram , start_address=0x601068, length=8
.data_safram_cplmnt, start_address=0x601070, length=8
testarray[1]=0xFE, one's complement=0x01
testarray[1]=0xFF, one's complement=0x00
testarray[1]=0x00, one's complement=0xFF
testarray[1]=0x01, one's complement=0xFE
testarray[1]=0x02, one's complement=0xFD
testarray[1]=0x03, one's complement=0xFC
testarray[1]=0x04, one's complement=0xFB
testarray[1]=0x05, one's complement=0xFA
testarray[1]=0x06, one's complement=0xF9
testarray[1]=0x07, one's complement=0xF8
testarray[1]=0x08, one's complement=0xF7
This is an example of protecting RAM variable with one's
complement! Run #1!
pid=30687
.data_safram , start_address=0x601068, length=8
.data_safram_cplmnt, start_address=0x601070, length=8
testarray[1]=0xFE, one's complement=0x01
testarray[1]=0xFF, one's complement=0x00
testarray[1]=0x00, one's complement=0xFF
testarray[1]=0x01, one's complement=0xFE
testarray[1]=0x02, one's complement=0xFD
testarray[1]=0x03, one's complement=0xFC
testarray[1]=0x04, one's complement=0xFB
testarray[1]=0x05, one's complement=0xFA
This is an example of protecting RAM variable with one's
complement! Run #2!
pid=30687
.data_safram , start_address=0x601068, length=8
.data_safram_cplmnt, start_address=0x601070, length=8
testarray[1]=0xFE, one's complement=0x01
testarray[1]=0xFF, one's complement=0x00
testarray[1]=0x00, one's complement=0xFF
testarray[1]=0x01, one's complement=0xFE
testarray[1]=0x02, one's complement=0xFD
testarray[1]=0x03, one's complement=0xFC
...
```

Process 2 - th_safram_corruption (from "crc" folder)

```
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
>./th_safram_corruption 30687 0x601070
8
Opening /proc/30687/mem, address is
0x601070
Data at 0x601070 in process 30687 is:
0
F6
2
3
4
5
6
7
Adding one to the first byte!
...
> ./th_safram_corruption 30687
0x601068 8
Opening /proc/30687/mem, address is
0x601068
Data at 0x601068 in process 30687 is:
FF
6
FD
FC
FB
FA
F9
F8
Adding one to the first byte!
...
```

The conclusions are very similar to the one's for checksum protection really. You can see for yourself (well, at least on openSUSE Leap 15) that the process 1 is restarted whenever we corrupt the memory from process 2. But as said before we did NOT really achieve any ASIL rated memory protection and separation (spatial freedom from interference if you want) compliant with ISO26262. Besides others (e.g. no coding practices required by ISO26262 - Part 6 were followed) the process restart mechanism is not safe. This of course brings a question what part of Linux kernel we would need to make safe in order to be able to rely on this. But that's beyond the scope of this blog post (and I don't know really at the moment).

To sum up this dual-storage of one's complement approach comes handy if we need to protect global static variables on applications where our process can be preempted by or has other less safety critical software components running in parallel. This is the main advantage really. The disadvantage is that we need twice as much as memory for our safety variables and the critical section did not disappear completely, it is still there, but limited in size. Please stay tuned if you wonder about what protection mechanism are implemented in Linux's memory management.

And as said, all the code is here: https://github.com/llansky3/safram in the "cplmnt" folder.

Memory corruption problems

And please allow me one correction to the last Safety RAM: Protecting memory section with checksum blog post that explains why it was not possible to corrupt the memory on Ubuntu 21, Debian 11 and possible others.

Firstly and the most importantly this was a bug in the my memory corruption program. The start address should have 64 bits and not just 32 bits. Sorry and thanks to my colleagues for pointing this out.. This is fixed now by:

```
uint64_t start_address = strtoul(argv[2], NULL, 16);
```

This demonstrated itself especially on Ubuntu and Debian where address space layout randomization (ASLR) is enabled and gcc builds with PIE enabled by default. This is not the case in openSUSE Leap 15 and Tumbleweed and thus the addresses land in lower addresses that can be accommodated by 32bit pointer.

But also you may have noticed the "ptrace restrictions are effectively disabled" line in the program output and related part of the code:

```
#ifndef PTRACE_NOT_ALLOWED
printf("ptrace restrictions are effectively disabled\n");
prctl(PR_SET_PTRACER, PR_SET_PTRACER_ANY);
#endif
```

So now let's explore what happens if we define the PTRACE_NOT_ALLOWED preprocessor macro. In this particular case this can be done by making everything with:

```
make DEFS=-DPTRACE_NOT_ALLOWED
```

You may observer (as I do) that the memory corruption still works fine. This is the case on openSUSE Leap 15 or Tumbleweed, but in other distributions you may get following error when attempting to corrupt the memory:

```
Opening /proc/15222/mem, address is 0xa51e9010
Data at 0xa51e9010 in process 15222 is:
  0
  0
  0
  0
Adding one to the first byte!
Error while writing
```

So what stops the memory corruption in these cases? The answer is Yama - Linux Security Module. The "ptrace_scope" is by default it is set 1 = restricted there. The openSUSE Leap 15 or Tumbleweed doesn't have this module enabled by default. That's why there is no issue in corrupting memory there. Of course if you are running the corruption program with "sudo" then you will not run into any of these problems.

You can check if you have this module in your distribution and the active ptrace scope by:

```
cat /proc/sys/kernel/yama/ptrace_scope
```

This page describes how to disable that if you are interested. By the way similar restrictions can be also achieved using AppArmor or SELinux.