

Code generation for memory monitoring

...

December 14, 2012

1 Introduction

Memory accesses from a pointer or an array are not safe in C, thus trying to access an element of an array out of valid range or an invalid pointer may cause a segmentation fault during execution.

C compilers such as GCC do not provide detection mechanisms for this kind of errors. We propose an automatic code instrumentation, so that the generated code will perform memory monitoring and will be able to check memory properties during execution (runtime checking). We consider as readable and writable memory: global variables, dynamically allocated memory, and formal parameters and local variables of each function. We decided to use the term “block” to nominate these four cases.

E-ACSL is an executable subset of ACSL [1], a formal specification language for C using source code annotations to express properties. ACSL-annotated C programs can be dealt with FRAMA-C [2], a framework for modular analysis of C. FRAMA-C provides a plug-in generating executable C code from E-ACSL annotations, used by the implementation discussed in this paper. Memory blocks (as defined above) held properties such as their base address and their size. E-ACSL provides five annotations to retrieve useful information about blocks:

\base_addr(p)

returns the base address of the block containing the pointer p

\block_length(p)

returns the size (in bytes) of the block containing the pointer p

\offset(p)

returns the offset between p and $\text{\base_addr}(p)$

\valid(s)

whether reading and writing $*s$ is safe

\initialized(s)

whether the variable stored at the address s has been initialized

Our contribution allowed the E-ACSL plugin to generate executable C code from these five annotations, thus checking these five memory properties during execution.

2 Stored information

For each block, we need to store the following information: the base address (address of the first element within the block), the size (in bytes), the validity status (whether reading and writing the block is safe) and the initialization status for each byte of the block. Figure 1 illustrates the data structure (we call it *block descriptor*) chosen to store this information.

init_ptr is an array of booleans that is dynamically allocated only if needed. If it has been allocated, it contains a boolean for each byte of the block: the n^{th} boolean indicates whether the n^{th} byte has been initialized.

```

1  struct _block {
2      char * ptr;
3      size_t size;
4      int valid;
5      unsigned char * init_ptr;
6      unsigned long init_cpt;
7  };

```

Figure 1: Block descriptor

init_cpt counts the number of initialized bytes within the block. If *init_cpt* = 0 (none) or *init_cpt* = *size* (all) then *init_ptr* is freed. So, when none or all of the bytes have been initialized (the most common cases), the memory space needed for the block descriptor itself is reduced. This consistency is maintained when adding/removing an element.

3 Global architecture

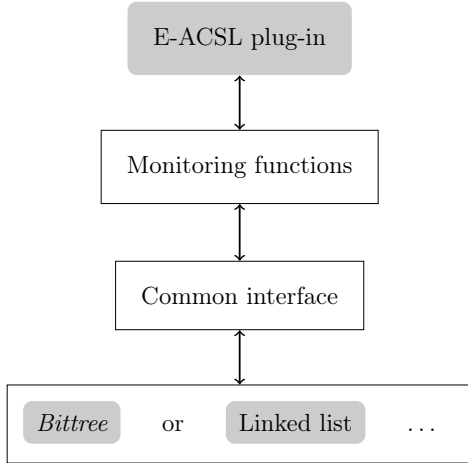


Figure 2: Global architecture

Interactions between the components of our solution are displayed by Figure 2. A data

structure (see Section 4) stores the block descriptors. The library functions (see Section 5) add, remove or modify the block descriptors. The E-ACSL plug-in then perform an instrumentation (see Section 6) to generate executable C code. The generated code uses the monitoring functions of this library.

4 Concrete data structure used: *bittree*

In this section we discuss the choice of the concrete data structure used to store the block descriptors.

4.1 *Patricia tries*

We need a data structure with a good time and space complexity, indeed we may have to often add or remove a block descriptor in the structure. The structure has to be sorted: we want to access to a block descriptor by its base address, and also to its predecessor and successor. Thus, hash-tables will not fit. We also unconsidered the linked lists, due to the linear worst-case complexity. The (unbalanced) binary search trees provide a linear worst-case complexity too when the base address of inserted elements are monotonically increasing, and this may be quite common. Self-balanced binary search trees are dismissed because of the numerous add/remove operations implying numerous costly balancing operations.

We choose to use the *Patricia tries* [3] structure, which is efficient even if the tree is unbalanced. The prefix used by a node (not a leaf) is the greatest common prefix (on 32 or 64 bits) of its two children. The block descriptors are held on leaves. Other nodes just do the routing from the root to a block descriptor. Patricia tries are usually used on strings and characters, so we named “bittree” the structure used in this article. For exam-

ple on 8-bit addresses, Figure 3 shows a bittree storing three block descriptors (identified by their addresses: 0010 0111, 0010 1001 and 0010 1101). The greatest common prefix of 0010 1001 and 0010 1101 is 0010 1 *** and the greatest common prefix of 0010 0111 and 0010 1 *** is 0010 *. The * means that this bit is meaningless. Figure 4 shows that a new node (with a new prefix) is added when a block descriptor is inserted.

Figure 5 shows an example of deletion of a 8-bit block descriptor. Patricia tries being compact prefix trees, a node having an only child is deleted. So 0010 1 *** becomes useless and is replaced by its child 0010 1001. Deleting useless nodes, or only storing useful ones for routing, keeps the tree exploration efficient. A 32-bit (respectively 64-bit) bittree has a worst-case depth of 33 (respectively 65) nodes.

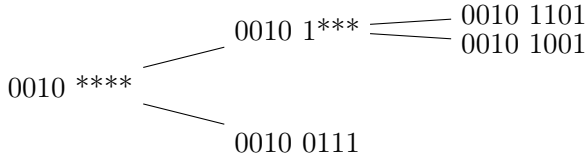


Figure 3: Example of bittree

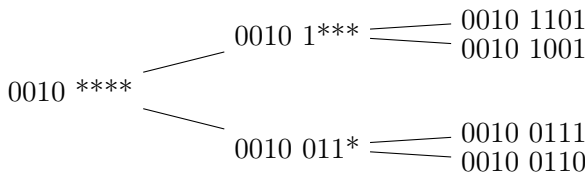


Figure 4: Bittree after adding 0010 0110 into the bittree of Fig. 3

4.2 Greatest common prefix computation

The greatest common prefix of A and B can be naively computed by: $X = \neg(A \oplus B)$

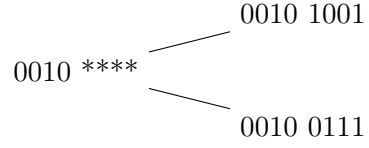


Figure 5: Bittree after deleting 0010 1101 from the bittree of Fig. 3

(where \oplus is the XOR operator) to keep bits in common, then each bit of X on the right side of a 0 is set to 0. The obtained mask is then applied to A or B . For example, considering $A = 0110 0111$ and $B = 0111 1111$, $X = \neg(A \oplus B) = 1110 0111$, setting each bit on the right side of a 0 to 0 we get 1110 0000. We apply this mask to A and get the greatest common prefix of A and B : 011 * *. This algorithm is used by the first version of our implementation and is named *Bittree-naïve* in the experiments.

We optimized this algorithm by firstly computing all of the 65 or 33 different masks (from 0x0 to 0xf...f) and storing them in an array. Then we use a dichotomic search to find the mask corresponding to the greatest common prefix: if A and B have a 32-bit common prefix, do they have a 48-bit common prefix ? Otherwise, do they have a 16-bit common prefix ? And so on. This search takes at most 6 (respectively 5) steps on 64-bit (respectively 32-bit) bittrees. This algorithm is used by the final version of our implementation and is named *Bittree-opti* in the experiments.

4.3 Experiments

These experiments justify the choice of bittrees over linked lists and binary search trees. We implemented the classic *merge sort* algorithm, and added extra allocations/deallocations to put each data structure to the test. The program has been instrumented (see Section 6) to call our monitoring functions (see Section 5).

The execution time of the instrumented program using each data structure has been measured (in micro-seconds) and is plotted against the number of calls to a function *store* (adding an element to the data structure). Figure 6 displays the results of the experiments. The reference time is the execution time of the program without any instrumentation. Bittree-naive and Bittree-opti are using two different versions of the greatest common prefix computation (see Sub-section 4.2).

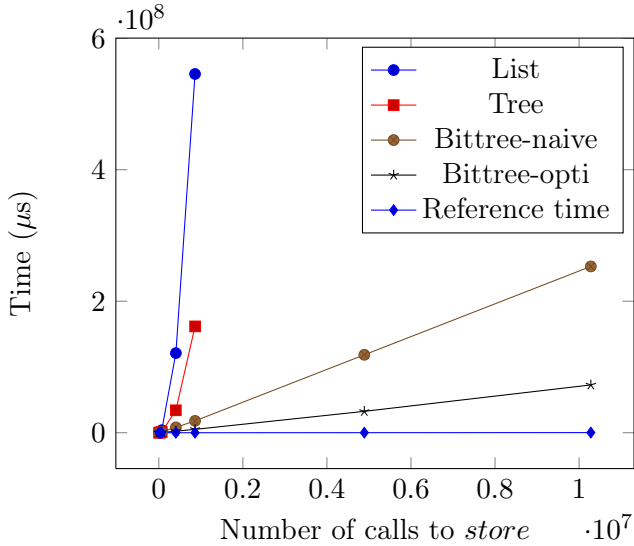


Figure 6: Execution time plotted against the number of calls to *store*

The last but one measurement, the merge sort applied to 50,000 elements (4,889,819 calls to *store*), has a reference time of 0.07s. It runs 32s with the optimized bittree, 118s with the naive bittree, 7 hours 15 minutes with the binary search trees and 19 hours 15 minutes with linked lists. Our last measurement, the merge sort applied to 100,000 elements (10,279,851 calls to *store*) has a reference time of 0.19s. It runs 72s with the optimized bittree, 252s with the naive bittree, but binary search trees and linked lists exceed our 24-hour timeout.

5 Monitoring functions

This section presents the functions used for adding/removing/retrieving in formations about block descriptors into our data structure (bittree). These functions will be automatically inserted in the source code by the instrumentation step (see Section 6).

5.1 Automatic allocation

Automatic allocation functions keep track of all non-dynamically allocated variables (but occupying memory space nevertheless), such as formal parameters, local variables or global variables.

```

1 void * _store_block
2   (void * ptr, size_t size);
3 void _delete_block(void * ptr);

```

Listing 1: Automatic allocation functions

5.2 Dynamic allocation

These functions have to be used instead of those of the standard library (*stdlib.h*).

```

1 void * _malloc(size_t size);
2 void * _realloc
3   (void * ptr, size_t size);
4 void * _calloc
5   (size_t nbr, size_t size);
6 void _free(void * ptr);

```

Listing 2: Dynamic allocation functions

5.3 Initialization

These functions have to be used for each assignment, to update the initialization status of

a block descriptor. `_initialize(ptr, size)` marks the *size* first bytes starting from *ptr* as initialized. `_full_init(ptr)` marks all the bytes of *ptr* as initialized at once. It is designed to avoid multiple calls to `_initialize` whenever it is possible and improves efficiency of the instrumented program.

```

1 void _initialize
2   (void * ptr, size_t size);
3 void _full_init(void * ptr);

```

Listing 3: Initialization functions

5.4 Interrogation

These functions are used to retrieve information about the block descriptors and match the E-ACSL annotations we are trying to support: `\valid`, `\base_addr`, `\block_length`, `\offset` and `\initialized`.

```

1 int _valid
2   (void * ptr, size_t size);
3 void * _base_addr(void * ptr);
4 size_t _block_length(void * ptr);
5 int _offset
6   (void * ptr, size_t size);
7 int _initialized
8   (void * ptr, size_t size);

```

Listing 4: Interrogation functions

`_valid(ptr, size)` returns 1 if it is safe to read/write *size* bytes starting from *ptr*, 0 otherwise. `_base_addr(ptr)` returns the base address of the block containing *ptr* if such a block exists, NULL otherwise. `_block_length(ptr)` returns the size (in bytes) of the block containing *ptr* if such a block exists, 0 otherwise. `_offset(ptr)` returns the offset between *ptr*

and the base address of the block containing *ptr* if such a block exists, -1 otherwise. `_initialized(ptr, size)` returns 1 if the *size* first bytes starting from *ptr* are initialized, 0 otherwise.

6 Instrumentation

This section presents the instrumentation performed by the E-ACSL plug-in to monitor the memory used by a program. The generated instrumented code uses the previously defined functions (see previous section).

For each global variable, calls to `_store_block` and `_full_init` are inserted at the beginning of the *main* function, and a call to `_delete_block` at the end (see Figure 9). For each formal parameter and local variable, a call to `_store_block` is inserted at the beginning of their scope block and a call to `_delete_block` at the end of their scope block (see Figure 10 and Figure 8). Calls to `_full_init` and `_initialize` are inserted on assignments (see Figure 11), and E-ACSL annotations are translated to the corresponding functions which result is tested by an assertion (see Figure 12).

```

1 int * p;
2 p = _malloc(32);
3 _free(p);

```

Figure 7: Dynamic allocation instrumentation

7 Conclusion

We implemented an efficient data structure (bittree) (see Section 4) to store the block descriptors and functions (see Section 5) relying on this structure to perform memory mon-

```

1 {
2   int* p;
3   _store_block(&p, sizeof(int*));
4   ...
5   _delete_block(&p);
6 }

```

Figure 8: Local variable instrumentation

```

1 void f (int i) {
2   _store_block(&i);
3   _full_init(&i);
4   ...
5   _delete_block(&i);
6 }

```

Figure 10: Formal parameter instrumentation

```

1 int g;
2
3 int main() {
4   _store_block(&g, sizeof(int));
5   _full_init(&g);
6   ...
7   _delete_block(&g);
8   ...
9   return 0;
10 }

```

Figure 9: Global variable instrumentation

```

1 int i;
2 i = 4;
3 _full_init(&i);
4
5 int t [10];
6 t[2] = 4;
7 _initialize((t+2), sizeof(int));

```

Figure 11: Assignment instrumentation

itoring. We also defined and implemented into the E-ACSL plug-in the instrumentation (see Section 6) to perform on a C source code to monitor the memory. This allows the runtime checking of the following E-ACSL annotations: `\base_addr`, `\block_length`, `\offset`, `\valid` and `\initialized`.

References

- [1] P. Baudin, P. Cuoq, J. C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. ACSL: ANSI/ISO C Specification Language preliminary design version 1.5. <http://frama-c.com/acsl.html>, 2010.
- [2] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A program analysis perspective. In *Proc. of the 10th International Con-*

ference on Software Engineering and Formal Methods, SEFM '2012, October 2012. To appear.

- [3] W. Szpankowski. Patricia tries again revisited. *J. ACM*, 37(4):691–711, Oct. 1990.

```
1 int p[12];  
2 //@ assert \valid (p+2);  
3 assert(_valid((p+2), sizeof(int)));
```

Figure 12: Annotation instrumentation