Static Analysis for Low Level Programs

Deliverable D2-1

ANR project VECOLIB

September 2016

Abstract

Verifying low level code is an essential step for ensuring correctness of libraries implementing containers. The code used in Ada or C implementations of standard libraries of containers includes complex data structures where the program heap is explicitly managed using pointers and dynamic allocation.

This deliverable reports on two approaches for static analysis of low level C code developed in the Vecolib project. The first approach is implemented in Frama-C and assumes a raw (array like) memory model. The second approach is implemented in an extension of Celia and combines in an original way the raw memory model with record memory model in order to analyse dynamic memory allocators.

1 Introduction

The C language features both low-level and high-level accesses to the memory (respectively via bit manipulations and typed expressions), and exposes the binary representation of high-level memory structures. Those dual views of the memory give more leeway to the programmers for implementing efficient programs, letting them choose the most convenient approach to address different algorithms. However, the interactions (and their restrictions) between the two models can be subtle and must be well understood. In particular, a commonly held view is that variable addresses and pointer values are simply integers, and can be handled accordingly. Even though the standard does not strictly legitimate this idea, a formal verification tool may choose to embrace it, in order to be able to verify the real-world programs that rely on this assumption. It is important for realistic analysers to take into account those peculiarities of the language, and to handle those constructs soundly – if not precisely.

The static analyser of Frama-C, Value [9] as well as its new version Eva [4] are able to deal with such view due to a specific memory model, where the contents of locations in memory are seen as sequences of bytes and pointers are abstracted in both integers and addresses. During the Vecolib project, the abstract domain used by the static analyzer has been extended to precisely capture bit-level operations on integers and pointers, as reported in Section 2.3.

The use of dynamic memory allocation (DMA) is mandatory in implementations of containers. Some libraries use the standard memory allocator (called by `malloc/free` functions available in `stdlib`), others implement their own dynamic memory allocator (e.g., formal containers in Ada). Therefore, it is important to verify that implementations of dynamic memory allocators are safe, e.g., they do not allocate out of the program data segment, they allocate disjoint memory regions, they do not leak memory that is freed or unused by the user. The code used in DMA implementation makes intensive use of low level operations on pointers and memory. During the VECOLIB project, we developed and implemented a static analysis technique based on abstract interpretation that is able to analyse the C code of DMA using the technique of "free list" to keep track of the set of regions available for allocation. This technique has been published in [6] and implemented as an extension of CELIA. It is shortly described in Section 3.

Related Work: Precise analyses exist for low level code in C [12] or for binary code [2]. They efficiently track properties about pointer alignment and memory region separations, but cannot infer shape properties. However, they use different abstract domains than the ones implemented in FRAMA-C and lose precision on bitwise operations. The absence of tracking shape properties avoids the use of the above techniques in the analysis of dynamic memory allocators.

For DMA analysis, [5] proposes an approach based on Separation Logic extended to pointer arithmetic. Another hierarchical analysis of shape and numeric properties has been proposed in [13]. They consider the analysis of linked data structures coded in arrays and track the shape of these data structures and not the organisation of the set of free chunks. Their approach is not based on logic and the invariants inferred on the content of list segments are simpler. [11] defines an abstract domain for the analysis of array properties and applies it to the Minix 1.1 allocator, which is a special class of allocators included in the one we consider.

2 Analysis with Frama-C

2.1 Memory Locations in Frama-C Eva

The static analysis of programs using pointers implemented in FRAMA-C is detailed in [9, 4]. Fundamentally, the static analyser EVA features an intricate memory abstraction able to represent efficiently and precisely both low-level concepts such as unions and bitfields, and high-level ones, such as arrays. This abstract domain is rich, but cannot infer relational properties between e.g. different variables.

EVA relies on a *base separation* hypothesis, where distinct variables are mapped to distinct (and separated) memory blocks. This can be linked to concrete memory addresses in the following way. For a program P, a valid memory layout in VALUE is an injective function $\theta : X \to N$ from variables in X to integer addresses N such that:

- the integer memory address of a variable is strictly positive (the integer 0 is used for the representation of the null pointer): $\forall x \in X, \theta(x) > 0$

- the contents of different variables do not overlap in memory : $\forall x, y \in X, \theta(y) > \theta(x) \Rightarrow \theta(y) - \theta(x) > \texttt{sizeof}(x)$ where the function `sizeof` gives the number of bytes occupied by the type of a variable. The comparison is strict to prevent two variables to be placed contiguously in memory. This is used later to always disambiguate pointers to $\&y$ from pointers to $\&x + \texttt{sizeof}(x)$.

- the content of the variables fits in memory: $\forall x \in X, \theta(x) + \texttt{sizeof}(x) < 2^{\texttt{sizeof}(ptr)}$

```
void *p1 = ..., *p2 = ...;
uintptr_t mask = !c; // c == 0 || c == 1 holds
// Return p1 or p2 without using conditionals useful for
// cryptographic code
r = (void*)(((uintptr_t)p1 & mask) | ((uintptr_t)p2 & ~mask));


// force alignment to 8 bytes
uintptr_t addr = (uintptr_t) p;
addr += 8 - addr % 8;
// another possibility
addr = (addr + (8 - 1)) & -8;
```

Figure 1: Code fragments with pointer masking

For a scalar type τ, an interpretation function is a bijective function φ_τ from any sequence of bytes of size $\texttt{sizeof}(\tau)$ to a value in τ.

A not null pointer value is a pair of a variable x and an integer i such that $0 \leq i \leq \texttt{sizeof}(x)$, written $(\&x, i)$. A memory location is a pointer value $(\&x, i)$ together with a type τ. It represents the consecutive addresses of the $\texttt{sizeof}(\tau)$ bytes in memory starting at the pointer value. The bytes at these addresses form a value of type τ.

2.2 Value Abstraction for Pointer Values

In Frama-C, the abstraction for pointer values is a set of tuples $(\&x_i, o_i^\sharp)$ where x_i is a program variable and $o_i^\sharp$ is an interval abstracting the set of possible values of the pointer offset. This is more precise than the usual abstraction $\overline{\&x} \times o^\sharp$, in which the offsets for different base addresses are coalesced together.

2.3 Bit-masking on pointers

C programs often use bitwise operators ("&" band, "|" bor, "$\wedge$" xor, "$\sim$" bnot) or shift ("<<" lshift, ">>" rshift) to extract parts of integers or pointers. In terms of precision, those operations are usually poorly handled by numerical analysis domain such as intervals or polyhedra. This loss of precision may lead to false alarms, but often remains acceptable in practice.

However, more severe problems occur when those operators are applied to pointers that have been cast into a proper integer type, such as uintptr_t. In this case, some analyzers will stop after reporting an invalid operation, and most others will lose important information about the possible offsets of the pointers. Yet, this form of coding is widespread on low-level C programs. We show in Figure 1 some code fragments that involve masking on pointers.

In the abstract interpreter of Frama-C, casting a pointer into an integer acts as the identity on the abstract value, but all subsequent numeric operations cause the abstract value to degenerate into a special object, called *garbled mix*. Those garbled mix keep track of the addresses they may contain, but nothing else.

```
int x, y, z;
int s = ((uintptr_t)&x + (uintptr_t)&y)-(uintptr_t)&y
// s IN {{ garbled mix of &{x; y} }
*(int *)s = 1; // alarm on possibly invalid pointer +
              // imprecise update of x or y -- but not z
```

Although the loss of precision is not total – the variable z is considered as not modified, the content of s is abstracted in a very imprecise way.

During the Vecolib projet, although we do not propose a solution for the (extreme) example above, we have implemented a new component for the analysis of the code fragments shown in Figure 1. More precisely, we have implemented a new abstract domain that precisely keeps track of *sequences of bits*. The abstract value for an expression of type τ is a sequence of $N \geq 1$ consecutive abstract values v_i that fully covers the size of τ. More formally, if each v_i has a width of s_i bits, then $\sum_{i=1..N} s_i = 8 * \text{sizeof}(\tau)$ must hold.

Let us write $o(k) = \sum_{i=1..k-1} s_i$ the *offset* of the i^{th} value. Given a concretization operator γ for atomic abstract values, the concretization of $\gamma(\overline{v_i})$ is $\sum^{\#}_{i=1..N} \gamma(v_i) \times^{\#} 2^{o(i)}$: we concretize each abstract value, shift it by the proper amount, and sum the results. The concretization of an integer or of a floating-point abstract value is standard. For pointers, the concretization uses the θ operator of Section 2.1.

Abstract operators for bitwise operations are implemented in the obvious way:

- left-shifting by k is implemented by adding k to the offsets, adding the abstract value $0^\sharp$ for the k first bits, and discarding the abstract values that correspond to the k highest bits.

- right-shifting by k is implemented by subtracting k from the offsets, discarding the abstract values that correspond to the k lowest bits, and adding the abstract values $0^\sharp$, $-1^\sharp$ or $[-1..0]^\sharp$ for the k highest bits, depending on the original sign bit.

- binary operators are implemented by splitting the abstract values on each side so that the offsets exactly coincide, then applying the abstract transformer pointwise.

In the static analyser Eva, atomic abstract values are actually more complex because we offer the possibility of using only *some* bits of a standard abstract value. Formally, they are triples $(v, b_{\min}, b_{\max})$ where

- v is a standard abstract value of EVA

- $b_{\min}$ (resp. $b_{\max}$) are the first (resp. last) bit that must be considered.

Thus, the concretization of (v, m, M) is $\gamma(v, m, M) = \{(n\%2^M)/2^m \,|\, n \in \gamma(v)\}$. This choice also makes the implementation of shift operations easier and more precise: if an abstract value of k bits must be shifted by l bits, we can precisely represent the result even when $l \neq k$.

This domain has been integrated into Eva, starting from Frama-C Aluminium. The first example of Figure 1 is already analyzed precisely, provided that c evaluates to either 0 or 1, or that both cases are analyzed separately. We hope to implement involved operations on pointeurs, such as the re-alignment operations at the bottom of the figure, for Frama-C Sulfur.

3 Analysis of Dynamic Memory Allocators

3.1 Motivation

The automated analysis of DMA faces several challenges. Although the code of DMA is not long (between one hundred to a thousand LOC), it is highly optimised to provide good performance. Low-level code (e.g., pointer arithmetics, bit fields, calls to system routines like `sbrk`) is used to manage efficiently (i.e., with low additional cost in memory and time) the operations on the chunks in the reserved memory region. At the same time, the free list is manipulated using high level operations over typed memory blocks (values of C structures) by mutating pointer fields without pointer arithmetic. The analyser has to deal efficiently with this *polar usage of the heap* made by the DMA. The invariants maintained by the DMA are complex. The memory region is organised into a *heap list* based on the size information stored in the chunk header such that chunk overlapping and memory leaks are avoided. The start addresses of chunks shall be aligned to some given constant. The free list may have complex shapes (cyclic, acyclic, doubly-linked) and may be sorted by the start address of chunks to ease free chunks coalescing. A precise analysis shall keep track of both numerical and shape properties to infer specifications implying such invariants for the allocation and deallocation methods of the DMA.

In [6], we proposed a static analysis that is able to infer the above complex invariants of DMA on both heap list and free list. We defined an abstract domain which uses logic formulas to abstract DMA configurations. The logic proposed extends the fragment of symbolic heaps of SL with a hierarchical composition operator, $\oplus$, to specify that the free list covers partially the heap list. This operator provides a hierarchical abstraction of the memory region under the DMA control: the low-level memory manipulations are specified at the level of the heap list and propagated in a way controlled by the abstraction at the level of the free list. The shape specification is combined with a fragment of first order logic on arrays to capture properties of chunks in lists, similar to [3]. This combination is done in an accurate way as regards the logic by including sequences of chunk addresses in the inductive definitions of list segments. The main advantages and contributions of this work are (1) the *high precision of the abstraction* which is able to capture complex properties of free list DMA implementations, (2) the *strong logical basis* allowing to infer invariants that may be used by other verification methods, and (3) the *modularity* of the abstract domain permitting to reuse existing abstract domains for the analysis of linked lists with integer data.

3.2 An example

We demonstrate the core ideas of our method on the C code presented in Figure 2 which is extracted from a DMA in our benchmark, the Aldridge's allocator [1], called LA in the following.

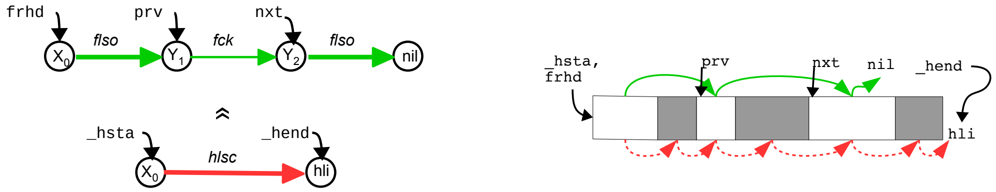
The code declares first an internal data type, `HDR`, used to build both the heap and the free list as follows. The field `size` stores the full size of the chunk (in blocks of `sizeof(HDR)` bytes). In the heap list, this information is used to obtain the start address of the next chunk by adding to the start address of the current chunk its size in bytes. The field `fnx` stores the start address of the next free chunk and it is used to form the singly linked list of free chunks, i.e., the free list. We added the ghost field *isfree* in this data type to mark explicitly free chunks and to simplify the presentation of our method. Lines 7–10 declare several globals variables: `_hsta` and `_hend` represent the first address of the entire memory block and the

```
 1 typedef struct hdr_s {
 2   struct hdr_s *fnx;
 3   size_t size;
 4   //@ghost bool isfree;
 5 }   HDR;
 6
 7 static void *_hsta = NULL;
 8 static void *_hend = NULL;
 9 static HDR *frhd = NULL;
10 static size_t memleft;
11
12 void minit(size_t sz)
13 {
14   size_t align_sz;
15   align_sz = (sz+sizeof(HDR)-1)
16              & ~(sizeof(HDR)-1);
17
18   _hsta = sbrk(align_sz);
19   _hend = sbrk(0);
20
21   frhd = _hsta;
22   frhd->size = align_sz / sizeof(HDR);
23   frhd->fnx = NULL;
24   //@ghost frhd->isfree = true;
25
26   memleft = frhd->size;
27 }
```

(a) Globals and initialisation

```
28 void* malloc(size_t nbytes)
29 {
30   HDR *nxt, *prv;
31   size_t nunits =
32     (nbytes+sizeof(HDR)-1)/sizeof(HDR) + 1;
33
34   for (prv = NULL, nxt = frhd; nxt;
35        prv = nxt, nxt = nxt->fnx) {
36     if (nxt->size >= nunits) {
37       if (nxt->size > nunits) {
38         nxt->size -= nunits;
39         nxt += nxt->size;
40         nxt->size = nunits;
41       } else {
42         if (prv == NULL)
43           frhd = nxt->fnx;
44         else
45           prv->fnx = nxt->fnx;
46       }
47       memleft -= nunits;
48       //@ghost nxt->isfree = false;
49       return ((void*)(nxt + 1));
50     }
51   }
52   warning("Allocation Failed!");
53   return (NULL);
54 }
```

(b) Allocation



(c) Part of the abstract invariant at line 34

(d) Concrete memory where green (resp. red) arrows represent the successor relation for the free (resp. heap) list

Figure 2: Running example with code

address right after the end of memory block respectively, `frhd` stores the address of the head of the free list, and `memleft` counts the number of free bytes in the memory region.

The method `minit` initializes these global variables and makes a reservation for a memory region such that it may store the requested `sz` bytes plus a header value. The memory is reserved due to the call of the system routine `sbrk`, that extends the data segment of the requesting process by the input value and returns the address representing the old limit of this segment. In the initial state, the heap list and the free list start at the same address, the beginning of the memory region reserved, `_hsta`. They contain only one chunk, which is set as free.

The method `malloc` tries to fulfil a request for allocating `nbytes` bytes. For this, it searches a free chunk whose body has size at least `nbytes` using the loop at lines 34–51 which traverses the free list and stops at the first free chunk satisfying this constraint; this way of choosing the free chunk is called the *first-fit policy*. If the free chunk is much larger, then it is split in two parts and the second part, i.e., at the end of the initial chunk, is allocated.

After several calls of allocation and deallocation methods, the memory region will be split into several chunks including free and busy chunks. An intuitive view of the concrete state of the DMA at line 36 is shown in Figure 2(d). The busy chunks are represented in grey. The "next chunk" relation in the heap list (defined using the field `size`) is represented by the lower arrows; the upper arrows represent the "next free chunk" relation defined by the `fnx` field. Notice that the free list is sorted by the start address of free chunks in this example. This fact eases the coalescing of successive free chunks. Indeed, LA implements the *early coalescing policy* which prevents to store in the heap list two successive free chunks. Therefore, the deallocation method of LA (not shown here) merges continuous free chunks into one bigger free chunk and updates both lists accordingly.

Our method abstracts set of states of the DMA using sets of formulas, each formula being a conjunction of predicate atoms. Figure 2(c) gives a graph representation for such a formula that specifies the concrete state represented in Figure 2(d). The heap list satisfying the early coalescing is specified by the atom $\mathsf{hlsc}(X_0, \mathsf{hli})$ where hlsc is an inductively defined predicate. The value X_0 and hli are stored in variables `_hsta` resp. `_hend`, which is represented by arrows sourcing these variables. The free list is abstracted by three atoms building the upper graph starting from X_0 also. The atom $\mathsf{flso}(X_0, Y_1)$ specifies the free list segment from the start of the list `frhd` to the location stored in `prv`, represented by the logic variable Y_1. The atom $\mathsf{fck}(Y_1, Y_2)$ specifies a free chunk at location Y_1 which stores in his *fnx* fiel the location Y_2, stored by variable `nxt`. The last atom $\mathsf{flso}(Y_2, \mathsf{nil})$ specifies another free list segment, suffix of the free list until null. The predicates used in these atoms are specified inductively using an extension of separation logic.

Both graphs specify fully (for the lower graph) or partially (for the upper graph) the same concrete memory region. The upper part highlights only the free list, but all the chunks in the free lists are also chunks of the heap list specified by the lower graph. To compose these two abstractions of the memory region, we introduce a new operator, the hierarchical composition "$\Supset$", which allows to relate the two levels of abstraction while keeping separated properties related with each kind of list used. For example, we are able to express the early coalescing property of the heap lists without interfering with the free list, which is not concerned about this policy. The formula obtained are used as abstract values in order analysis algorithm to represent program configurations. The separation of concerns provided by the hierarchical composition is used by the analysis we propose in order to focus only on the abstraction level required by the statements to be analysed. For example, the loop traversing the free list at lines 34–35 requires to reason only at the free list level. The details on this analysis are provided in [6] (a journal version is submitted).

3.3 Experimental results

We implemented the abstract domain and the analysis algorithm in Ocaml as a plug-in of the Frama-C platform [9]. We are using several modules of Frama-C, e.g., C parsing, abstract syntax tree transformations, and the fix-point computation. The data word domain uses as numerical join-lattice $\mathcal{N}$ the library of polyhedra with congruence constraints included in Apron [7]. To obtain precise numerical invariants, we transform program statements using bit-vector operations (e.g., line 16 of Figure 2(a)) into statements allowed by the polyhedra domain which over-approximate the original effet.

We applied our analysis on the benchmark of free list DMA built from the example above, published by Aldridge [1], our implementation of the DMA proposed by Knuth in [10], the allocator published in the famous book of C written by Kernighan and Ritchie [8]. We were able to discover a bad list traversal in [1], and to infer the policies (and therefore ensure the correctness) in the other allocators. More details are provided in [6].

References

[1] L. Aldridge. Memory allocation in C. *Embedded Systems Programming*, pages 35–42, August 2008.

[2] G. Balakrishnan and T. W. Reps. Recency-abstraction for heap-allocated storage. In *SAS*, volume 4134 of *LNCS*, pages 221–239. Springer, 2006.

[3] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. On inter-procedural analysis of programs with lists and data. In *PLDI*, pages 578–589. ACM, 2011.

[4] D. Bühler. *Structuring an Abstract Interpreter through Value and State Abstractions: EVA an Evolved Value Analysis for Frama-C*. PhD thesis, University of Rennes, 2017.

[5] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *SAS*, volume 4134 of *LNCS*, pages 182–203. Springer, 2006.

[6] B. Fang and M. Sighireanu. Hierarchical shape abstraction for analysis of free-list memory allocators. In *LOPSTR*, volume 10184 of *Lecture Notes in Computer Science*. Springer, 2016.

[7] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, volume 5643 of *LNCS*, pages 661–667. Springer, 2009.

[8] B. W. Kernighan and D. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988.

[9] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015.

[10] D. E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition*. Addison-Wesley, 1973.

[11] J. Liu and X. Rival. Abstraction of arrays based on non contiguous partitions. In *VMCAI*, volume 8931 of *LNCS*, pages 282–299. Springer, 2015.

[12] A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *LCTES*, pages 54–63. ACM, 2006.

[13] P. Sotin and X. Rival. Hierarchical shape abstraction of dynamic structures in static blocks. In *APLAS*, volume 7705 of *LNCS*, pages 131–147. Springer, 2012.