2018

A Formally Verified Heap Allocator

Arash Sahebolamri
Syracuse University, asahebol@syr.edu

Scott D. Constable
Syracuse University, sdconsta@syr.edu

Steve J. Chapin
Syracuse University, chapin@syr.edu

A Formally Verified Heap Allocator

Arash Sahebolamri, Scott Constable, Steve J. Chapin
Syracuse University
{asahebol,sdconsta,chapin}@syr.edu

August 14, 2018

Abstract

We present the formal verification of a heap allocator written in C. We use the Isabelle/HOL proof assistant to formally verify the correctness of the heap allocator at the source code level. The C source code of the heap allocator is imported into Isabelle/HOL using CParser and AutoCorres. In addition to providing the guarantee that the heap allocator is free of bugs and therefore is suitable for use in security critical projects, our work facilitates verification of other projects written in C that utilize Isabelle and AutoCorres.[1]

1 Introduction

Up until a few years ago, formally verifying programs written in low-level languages like C was not possible. The flexibility of such languages makes them very powerful for writing systems-level software, however this flexibility comes at the cost of desirable features like type and memory safety. It is all too easy to introduce subtle bugs in programs written in C that can have catastrophic security consequences. The same lack of type and memory safety makes it difficult to formally reason about code written in these languages. There exist languages at the opposite end of the safety spectrum, such as Haskell, which guarantee type and memory safety. A language such as Haskell is not a strong candidate for writing systems software for a variety of reasons, including lack of control over memory layout, and the difficulty of interfacing with the underlying hardware. Thus, low-level systems software is almost always written in languages like C and C++.

The strongest guarantee of trustworthiness in these most critical (and yet, most vulnerable) programs is to formally verify them; in practice, that entails providing machine-checked proofs about their behavior. Isabelle/HOL is a proof assistant capable of checking formally-written proofs in Higher Order Logic. Isabelle/HOL comes with an interactive environment for writing proofs. In addition to a proof assistant, a tool is also required to import the

[1] To access the heap allocator implementation and the proofs, contact the authors.

program source code into the proof assistant so that it can be reasoned about. AutoCorres [2] is such a tool. This tool relies on another tool, CParser [6], that translates C code into a very basic imperative language embedded in Isabelle/HOL called SIMPL. AutoCorres takes the SIMPL imperative code and turns it into a monadic form more amenable to formal verification, and provides correspondence proofs between the SIMPL code and the monadic form, guaranteeing that the translation done by AutoCorres is correct without needing to trust the translation process.

The combination of Isabelle/HOL and AutoCorres has been used for formal verification of the seL4 microkernel. In fact, Data61, the lab behind seL4, developed AutoCorres as part of its seL4 verification effort [4].But seL4 is not the only project that takes advantage of AutoCorres for formal verification; SABLE[2][1] is a secure boot loader developed at Syracuse University that also relies on Isabelle/HOL and AutoCorres for formal verification.

One consequence of committing to provide formal verification of a piece of software is the set of restrictions the commitment imposes on using helper libraries. Because every line of code that the project relies on is brought into the trusted computing base (TCB), use of any library adds an obligation to prove the correctness of the implementation of those libraries. This is true even for basic functions that are often taken for granted such as malloc() and free(), the heap management functions in C.

seL4, being a microkernel with unique memory management needs, did not require a general purpose heap allocator; but nearly every piece of software requires a dynamic heap allocator. The SABLE project mentioned earlier is one such project. Indeed, it was SABLE's formal verification effort that compelled us to create a formally verified heap allocator.

2 Background

CParser represents the heap as a tuple of the raw heap memory, and a tag describing the type(s) associated with each memory location.

type-synonym $heap\_raw\_state$ = heap-mem × heap-typ-desc

heap-mem is a function from memory addresses to stored values; and heap-typ-desc is a function mapping each memory address to the type description associated with that address. This type description is a little involved, since it is capable of representing multiple types associated with a memory location. This multiple type situation arises with the use of arrays or user defined struct types.

This representation of heap memory is very powerful, and allows reasoning about fragments of code that are not type-safe, however, it is not a particularly elegant abstraction to work with when reasoning about code written in a type-safe manner; such code usually constitutes the bulk of systems software projects. Therefore, AutoCorres has built-in techniques to abstract away this low level representation by generating functions to lift heap-raw-state into multiple disjoint heaps, with one heap per type used in the source code [3].

[2]The Syracuse Assured Boot Loader Executive.

Since this lifting into disjoint heaps assumes the heap memory is accessed in a type-safe manner, AutoCorres puts guards around any heap references through pointers in the code asserting that the dereferenced memory location is of the correct type. In practice, these guards create proof obligations when reasoning about the code behavior; proof obligations that are generally trivial to discharge when reasoning about type-safe code, but impossible to satisfy when reasoning about portions of the code that are not type-safe. The following piece of code demonstrates this situation.

```
void init_heap(void *heap, UINT32 heap_size) {
   struct mem_node *n = heap;
   ...
}
```

With the heap abstraction enabled, the above piece of code gets translated into this:

init-heap' *heap heap_size* $\equiv$
$n \leftarrow$ return (ptr-coerce *heap*) :: (mem-node-C ptr);
guard (λs. is-valid-mem-node-C s n);
...

While without it, the translation looks like this:

init-heap' *heap heap_size* $\equiv$
$n \leftarrow$ return (ptr-coerce *heap*) :: (mem-node-C ptr);
guard (λs. c-guard n);
...

The is-valid-mem-node-C predicate requires the heap type description in current state s to agree that the type of n is mem-node-C. Given that we are casting a pointer of type void* to mem-node*, the guard cannot be satisfied if the *heap* pointer was a valid one in the heap type description. The c-guard predicate that replaces is-valid-mem-node-C when heap abstraction is disabled is only a constraint on the value of the pointer (requiring it not to be null, not to wrap around the address space, be properly aligned, etc.), and has no requirements on the current heap state (thus the absence of the state parameter s).

For reasons such as this, reasoning about a heap allocator, which inevitably has portions that are not type-safe, requires reasoning about the raw heap state. On the other hand, reasoning about a type-safe piece of code that takes advantage of the heap allocator can stay within the abstraction of a lifted heap, and as a result be much more convenient.

3 The Heap Allocator

The heap allocator that we implemented is deliberately a simple one, with an alloc() function for allocating memory, and a free() function for deallocating previously allocated memory.

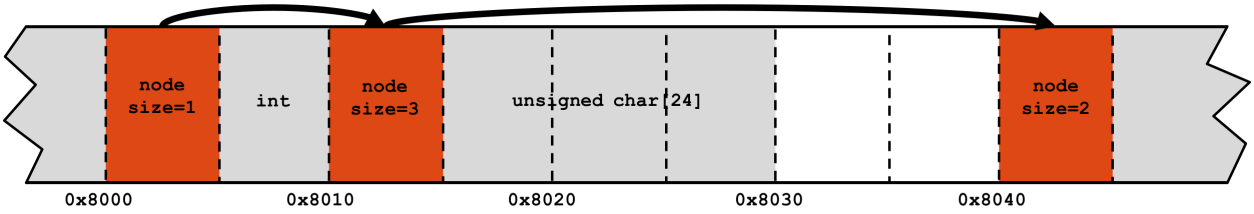

Figure 1: An example heap. Red nodes are the heap nodes; gray nodes are typed memory (allocated memory); and white nodes are free memory.

```
void *alloc(void *heap, UINT32 size) {...}
void free(void *heap, void *obj){...}
```

Prior to use, the heap allocator needs to be initialized by calling the init_heap() function. To allow the flexibility of choosing the heap size, our implementation allocates memory on a memory region that was provided to the init_heap() function.

```
void init_heap(void *heap, UINT32 heap_size) {...}
```

A singly linked list of mem_node objects keeps track of which regions of memory within the heap are free, and which regions are occupied. Figure 1 visualizes the structure of a sample heap.

Each call to alloc() traverses this linked list to find a free region big enough for allocation; and each call to free() traverses the linked list to find the node within the linked list preceding the node that needs to be freed. The preceding node is the one whose next pointer needs to be updated.

4 Verification

As previously mentioned, we used the Isabelle/HOL proof assistant for verification, and the combination of CParser and AutoCorres to import the heap allocator source code into the proof assistant. After importing the code, our main effort involved proving Hoare triples about the three heap allocator functions.

Our first attempt at verification was done on an implementation of the heap allocator that did not support deallocating memory (one with no free() support). This limitation on the heap allocator simplified the code, and reduced the amount of effort required to verify it. More specifically, lack of free() support meant that there would be no loops in the heap allocator traversing the linked list to find a suitable node to allocate memory from. This in turn obviated the need to represent the linked list in the verification effort at all. Not needing to have a representation of the linked list in the proofs reduced the complexity of the invariants predicate that was required for the verification of the alloc() function. Instead of being a recursive function that would scan the entire linked list, the invariants would consist of few constraints on the position of the pointer pointing to the first free location in the

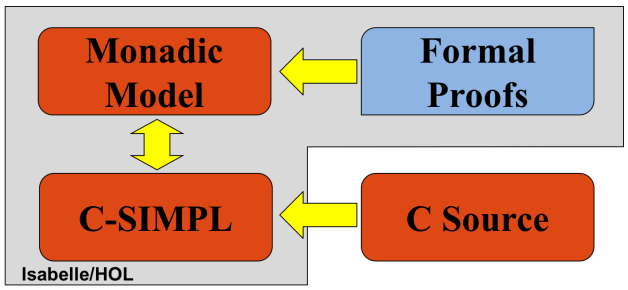

Figure 2: The verification process. It involves importing the C source code into Isabelle/HOL using CParser, abstracting it by AutoCorres, and providing proofs about the abstracted code.

heap, and on the heap state starting from that location being untyped. We would also be relieved from having to deal with notions like a node being reachable from another node, or the path from one node to another.

The reduced complexity kept the size of proofs to around 300 lines of proof (LoP). This version of the heap allocator, and the accompanying proofs, while not general enough to be useful in many projects, provided us with valuable information on how to deal with the C parser/AutoCorres memory abstractions, and on how to structure the proofs.

It is typical of these verification efforts to revolve around finding suitable invariants that capture the behavior of the software, and proving that the invariants are preserved with each call to any of the functions in the code. For the heap allocator, we needed these invariants theorems:

$\{\top\}$	$\{\lambda s.$ heap-invs s *heap*$\}$	$\{\lambda s.$ heap-invs s *heap*$\}$
init-heap' *heap size*	alloc' *heap size_bytes*	free' *heap ptr*
$\{\lambda r\ s.$ heap-invs s *heap*$\}$	$\{\lambda r\ s.$ heap-invs s *heap*$\}$	$\{\lambda r\ s.$ heap-invs s *heap*$\}$

As for the invariants themselves, they were defined as a recursive function that traverses the linked list and checks each node for validity.

The nodeValid predicate puts constraints on the node's size and next fields, and ensures that the next node is located at a higher address than the current node. This monotonicity is both a reflection of the implementation of the heap allocator, and a necessity in proving totality of the heap-invs function.

definition nodeValid s node ≡
let next = node-next s node; size = node-size-masked s node in
c-guard node ∧
unat-ptr node + 8 + unat (size ∗ 8) ≤ (if next ≠ NULL then unat-ptr next else 2 ^ 32) ∧
unat size ∗ 8 < 2 ^ 32 ∧
(node-occupied-flag s node = 0 ⟶ nodeFree s node) ∧
(next ≠ NULL ⟶ next > node ∧ next ≥ node $+_p$ 1)

Dealing with word arithmetic is one of the challenges of verification at the source code level, as seen in the nodeValid definition. However, such reasoning is necessary in guaranteeing lack of subtle bugs caused by integer overflow and other arithmetic errors in low level code.

By far most of the verification effort was spent on proving the invariants theorems and their supporting lemmas. In order to be able to prove the invariants theorems, we needed to introduce two more recursive definitions, reachable, which is a predicate on whether one node is reachable in the linked list from another; and path, which lists all the nodes in the path of the linked list from one node to another.

As an example, here is a lemma that connects heap-invs and path together.

lemma node-in-path-heap-invs-imp-nodeValid-node:
n ∈ set (path s fst-node to) ⟶
heap-invs s fst-node ⟶
nodeValid s n

And here is a lemma involving heap-invs and reachable.

lemma heap-invs-reachable-imp-heap-invs:
reachable s fst-node node ⟹
heap-invs s fst-node ⟹
node ≠ NULL ⟹
heap-invs s node

The lemma that best demonstrates the need for these three recursive definitions is path-nodeValid-reachable-imp-heap-invs. This lemma states that given (1) the nodes in the path from the node heap to the node node are valid, and (2) node is reachable from heap, and (3) heap-invs holds for node, then heap-invs also holds for heap.

lemma path-nodeValid-reachable-imp-heap-invs:
nodeValid s heap ⟶
(∀ p ∈ set (path s heap node). nodeValid s p) ⟶
reachable s heap node ⟶
heap-invs s node ⟶
heap-invs s heap

Functions in Isabelle/HOL, being mathematical functions, are required to be total. This means that each function defined in Isabelle/HOL must have a value for every member of its domain. In practice this means that recursive definitions must be shown to terminate. For recursive functions that are defined by pattern matching on the constructors of algebraic data types like 'a list or nat, the termination proof is trivial enough that it is automated by the proof assistant. But for more complex recursive functions the proof burden is on the proof engineer. In our verification attempt these proof burdens raised interesting challenges. Take the definition of heap-invs as an example. It is logically defined as a conjunction stating

the validity of the node, and recursing to the next node in the linked list.

heap-invs s $heap$ =
nodeValid s $heap$ $\wedge$
$\big($heap-invs (node-next s $heap$) $\vee$ node-next s $heap$ $=$ $NULL\big)$

This simple definition in Isabelle/HOL cannot be shown to be a total function, because syntactically every invocation of heap-invs recurses to the next node, even though logically one can deduce the value of the function for every combination of parameters. Fortunately, there is a facility in Isabelle/HOL for dealing with these situations: splitting the definition of a recursive function into multiple cases.

function heap-invs :: globals ⇒ mem-node-C ptr ⇒ bool **where**
¬ (nodeValid s heap) ⟹ heap-invs s heap = False
|nodeValid s heap ∧ node-next s heap = NULL ⟹ heap-invs s heap = True
|nodeValid s heap ∧ node-next s heap ≠ NULL ⟹ heap-invs s heap = heap-invs s (node-next s heap)

With this split, the recursion only happens when the nodeValid predicate holds and the next node is not null. Since nodeValid requires the next node to be strictly greater than the current node, the termination proof can now establish a *measure function* that is strictly monotonic. This is sufficient to prove termination.

With the invariant proofs in place, the main theorem about alloc() can be proved. This theorem states that a call to alloc() returns a valid pointer to a region of memory for the target type. A valid pointer is one that is not null, is properly aligned, and does not wrap around the address space.

theorem alloc′-hoare:
size-of TYPE('a) ≤ unat size-bytes ⟹
0 < size-bytes ⟹
{|λs. heap-invs s heap-node|}
alloc′ heap-node size-bytes
{|λr s. let ptr = (ptr-coerce r) :: ('a :: mem-type) ptr in
ptr-val r ≠ 0 ⟶ heap-ptr-valid (ptr-retyp ptr (hrs-htd (t-hrs-′ s))) ptr|}

5 Usage

The alloc-hoare theorem establishes the validity of the returned pointer of a call to alloc(). However, it states the validity of the pointer not in the post state of alloc', but in the post state with its heap type description updated to denote the memory at the location of the returned pointer to be of the desired type. Since the heap type description is an auxiliary piece of state with no counterpart in the C code, updates to the heap type description are made possible by special directives in the C code written as comments. These directives are processed by C Parser. The directives are of the form

```
/** AUXUPD: "([guard], [heap type description update function])" */
```

Thus, to be able to take advantage of the alloc-hoare theorem, a directive like this can be inserted in the C code after a call to alloc().

```
int* p = alloc(heap, sizeof(int));
/** AUXUPD: "(True, ptr_retyp (ptr_coerce \<acute>p :: word32 ptr))" */
```

This process can be further automated by adding a number of C macros to the code. For example, a function for allocating objects of type Foo can have the following form.

```
Foo* alloc_Foo(void* heap){
  void* res = alloc(heap, sizeof(Foo));
  /** AUXUPD: "(True, ptr_retyp (ptr_coerce \<acute>res :: Foo ptr))" */
  return (Foo*) res;
}
```

With such a function, it would be trivial to have a version of alloc-hoare specific to type Foo.

6 Evaluation

One purpose of our verification effort was to make sure that our implementation of the heap allocator was free of bugs. We expected that any bugs in the heap allocator would render the theorems unprovable, which would alert us to their existence. In our earlier implementation of the heap allocator, we encountered and fixed two such bugs.

1. Our formula to convert bytes to heap blocks was susceptible to unsigned integer overflow. In this case, the heap would allocate fewer bytes than the caller would have expected to receive. Thus the caller could accidentally overwrite the heap; this could be exploited as a heap overflow attack.

2. One "<" should have been a "<=". Before the fix, the heap allocator could have written beyond the end of heap memory.

In addition to these legitimate vulnerabilities, our verification effort forced us to make changes to facilitate the proofs. These changes, while did not address any bugs, did have the effect of clarifying the code.

7 Conclusion & Future Work

In this work, we presented our effort formally verifying a heap allocator in the proof assistant Isabelle. This work, in addition to providing a guarantee that the heap allocator is bug-free and therefore suitable for security critical applications, paves the way for utilizing the heap

allocator in projects written in C that aim to be formally verified using the Isabelle/HOL framework and AutoCorres.

Since our heap allocator implementation targeted simplicity and proof convenience, it has left room for optimization. We leave that to future work.

Our proofs provide strong guarantees that our implementation of the heap allocator is bug-free. However, the theorems we've proven do not mention the global state. This turned out to be sufficient for our purposes of developing SABLE proofs, but we are aware that to be truly usable in a wide range of scenarios, we need to provide further, stronger theorems about the behavior of the heap allocator. These theorems need to be able to address questions such as

- What happens to the value of a global variable when alloc() or free() is called?
- Is a freshly allocated piece of memory disjoint from all other allocated memory objects?
- Do alloc() and free() guarantee that they do not touch the contents of already allocated memory objects?

To allow a verification effort deal with these kinds of questions, we need to go beyond the theorems presented here. We'll also need a language that is suitable to expressing properties that tackle these kinds of questions. To that end, we are planning on utilizing Separation Logic [5]. Separation Logic is a logic for reasoning about program behavior in presence of shared memory. The most prominent language construct of separation logic is the separating conjunction (*). P * Q asserts that the heap can be partitioned into two disjoint heaps in such a way that P is satisfied in one partition and Q in the other. This simple connective gives us the power to express our desired properties about alloc() and free() succinctly. Versions of these theorems specialized to integers would look like this:

lemma alloc-int'-sep :
{heap-invs-sep h $*$ P}
alloc-uint32' h
{λr. heap-invs-sep h $*$ P $*$ heap-only-at r}

lemma free-int'-sep :
{heap-invs-sep h $*$ heap-only-at ptr $*$ P}
free' h ptr
{λ r. heap-invs-sep h $*$ P}

heap-invs-sep would be essentially the heap-invs predicate, asserting invariants about the heap. Theorem alloc-int'-sep asserts that assuming the heap invariants and an arbitrary P that does not touch the heap structure (since it is joined with heap-invs-sep using the separating conjunction) as preconditions, after the call to alloc(), P will hold, along with the invariants. In addition, existence of heap-only-at r joined using the separating conjunction in the post condition guarantees that the returned pointer is a freshly allocated piece of memory since it is guaranteed to be disjoint from any memory location P might have accessed.

Interestingly, the free() theorem is symmetric with the alloc() theorem, which has the effect of guaranteeing no state changes outside the heap after successive calls to free() and alloc().

We see the power of Separation Logic in allowing us to succinctly formulate theorems capable of providing guarantees expected of a heap allocator, some of which where listed above.

Our work proving the invariants portions of the lemmas presented here provides the foundation on which we'll build to prove the above theorems.

8 Acknowledgments

This research was supported by the Air Force Research Lab and New York State; cooperative work with Critical Technologies, Incorporated, of Utica, NY.

References

[1] S. Constable, R. Sutton, and S. J. Chapin. A high-level overview of sable, a modern secure loader. Technical Report SYR-EECS-TR-2013-08, Syracuse University, 2013.

[2] D. Greenaway, J. Andronick, and G. Klein. Bridging the gap: Automatic verified abstraction of c. *ITP*, 7406:99–115, 2012.

[3] D. Greenaway, J. Lim, J. Andronick, and G. Klein. Don't sweat the small stuff: formal verification of c code without the pain. *ACM SIGPLAN Notices*, 49(6):429–439, 2014.

[4] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[5] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

[6] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In *ACM SIGPLAN Notices*, volume 42, pages 97–108. ACM, 2007.