Verified Sequential Malloc/Free

Andrew W. Appel
Princeton University
USA
appel@princeton.edu

David A. Naumann
Stevens Institute of Technology
USA
naumann@cs.stevens.edu

Abstract

We verify the functional correctness of an array-of-bins (segregated free-lists) single-thread malloc/free system with respect to a correctness specification written in separation logic. The memory allocator is written in standard C code compatible with the standard API; the specification is in the Verifiable C program logic, and the proof is done in the Verified Software Toolchain within the Coq proof assistant. Our "resource-aware" specification can guarantee when malloc will successfully return a block, unlike the standard Posix specification that allows malloc to return NULL whenever it wants to. We also prove subsumption (refinement): the resource-aware specification implies a resource-oblivious spec.

CCS Concepts: • **Software and its engineering → Formal software verification**; **Functionality**; **Software verification**.

Keywords: memory management, separation logic, formal verification

ACM Reference Format:
Andrew W. Appel and David A. Naumann. 2020. Verified Sequential Malloc/Free. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management (ISMM '20), June 16, 2020, London, UK*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3381898.3397211

1 Introduction

There are now functional-correctness verification tools for C programs [7, 15, 18], in which researchers have proved the correctness of crypto primitives [5, 38], crypto protocols [33] network address translation [39], concurrent messaging systems [24], and operating systems [13, 19]. Many of these programs rely on standard libraries such as malloc/free—so we now verify the correctness of malloc/free. This also serves as a demonstration and assessment of the verification tool.

C's malloc/free library casts undifferentiated bytes to and from the data structures that it uses internally; the client of the library casts to and from its own structs and arrays. In the process, implicit alignment restrictions must be respected. For a formal verification, it is not enough that the program actually respect these restrictions: we want the program logic (or verification tool) to be sound w.r.t. those restrictions—it should refuse to verify programs that violate them.

In fact, the alignment restrictions, object-size restrictions, and integer-overflow properties of C are quite subtle [35]. We want the program logic (and verification tool) to be sound (proved sound with a machine-checked proof) with respect to the operational semantics of C (including alignment constraints, etc.).

Allocation and freeing should be (amortized) constant-time. The usual method is Weinstock's array-of-bins data structure for quickly finding free blocks of the right size [36]. Large blocks must be treated separately; a modern memory allocator can manage large blocks directly using the `mmap` system call.

To do formal verification, we should use a suitable program logic. C programs that use pointer data structures are most naturally specified and verified in separation logic (SL) [29].

There must be a formal specification—otherwise we cannot prove correctness, only weaker properties such as memory safety. The simplest specification of `malloc` says that the function has complete discretion to return `NULL`, *or* it can choose to allocate a block (of at least the right size) and return a pointer. Defensively written C programs should check the return value. But suppose you want to verify that a program actually can complete a task. We provide a *resource-aware* specification that keeps track of the malloc "resource," so that one can prove that resource-bounded programs will always get a block from our malloc, never NULL.

Contributions. Our malloc/free implementation is

1. compatible with the standard API;
2. written in standard C;
3. amortized constant time performance (for small blocks) using an array of bins, and single-system-call performance (for large blocks);
4. proved correct with a machine-checked proof,
5. in a foundationally verified program logic and verifier,

ISMM '20, June 16, 2020, London, UK

ACM ISBN 978-1-4503-7566-5/20/06…$15.00
https://doi.org/10.1145/3381898.3397211

6. with a resource-aware separation-logic specification guaranteeing when malloc *will* allocate a block,
7. and with a "subsumption" proof that the resource-aware specification implies a resource-oblivious specification.

Limitations. In this work we do not address concurrent threads, for which state-of-the art allocators (including snmalloc, mimalloc, tcmalloc, jemalloc, and ptmalloc) provide support by caching per thread (or per cpu) and other techniques [23]. We do not attempt performance enhancements such as shared metadata and splitting/coalescing, or locality-enhancing techniques such as sharding (mimalloc). We use linear spacing of block sizes; some (e.g., dlmalloc, jemalloc) switch to logarithmic spacing for larger blocks. We do not consider extensions of the standard API such as pool allocation (arenas in jemalloc, first-class heaps in mimalloc), profiling, or cooperation with the VM by purging unused dirty pages (jemalloc, mimalloc). See Section 10 for more discussion.

Related work. Marti *et al.* [26] used separation logic in the Coq proof assistant[1] to verify a memory allocator in almost-C; that is, their work had property (4) but not (1, 2, 3, 5, 6, 7). Tuch [32] verified an allocator for the L4 microkernel that was for a simpler API, written in C, with no array-of-bins or other high-performance data structure, with machine-checked proofs in separation logic (and in another style); that is, with properties (2,4) but not (1, 3, 5, 6, 7). Wickerson *et al.* [37] give a proof outline in separation logic for part of Unix Version 7 malloc/free; (1,2) but not (3,4,5,6,7). Zhang *et al.* [40] verified a two-level segregated-fit memory allocator in the Isabelle/HOL proof assistant; it was not a C program but a functional model of a program: properties (3,4) but not (1, 2, 5, 6, 7). Jiang *et al.* [16] verified the model of a buddy allocator; properties (3,4) but not (1, 2, 5, 6, 7).

Automatic garbage collection: Wang *et al.* [34] verified a generational copying garbage collector written in C: properties (3,4,5) but not (1, 2, 6, 7) in the sense of a malloc/free system. Previously, Birkedal *et al.* [6] had verified a copying garbage collector in a toy while-loop language (property 4 only); Gammie *et al.* [11] verified a concurrent garbage collector in a tiny while-loop language but using an accurate x86-TSO memory model (property 4). McCreight *et al.* [28] verified a mark-sweep collector in assembly language using a single free-list (not segregated objects) and only one size of cell (property 4). McCreight *et al.* [27] showed a verified source-to-source transormation that allows C to be garbage-collected (properties 2, 4, 5).

Resource-aware malloc/free: Barthe *et al.* [2] characterize resource-awareness only for allocation, not for freeing, for Java bytecode programs, with an emphasis on analyzing client programs and no verification of the allocator itself. Hofmann *et al.* develop several type systems for resource bounds in functional programs (e.g., [14]), with the aim of certifying resource bounds on compiled code with certificates based on a type system [1] or general program logic [4]. The latter targets an idealized bytecode and restricts the ways memory can be used, but its tracking of available resources inspired our resource-aware specifications. None of the works discussed in this paragraph verify the memory manager itself.

High-performance (unverified) malloc/free: We discuss some modern high-performance (unverified) memory allocators, which improve client locality and support multicore concurrent clients, in Section 10.

Verified C compilers, verified verifiers: Our verification is carried out using the Verified Software Toolchain (VST) [7], which is sound with respect to the C standard and is foundationally verified: it has a machine-checked proof of soundness with respect to the C semantics of the CompCert verified C compiler [22]. Relevant aspects of VST are described in later sections.

Outline of this paper. Section 2 describes the implementation. Section 3 introduces what the reader needs to know about separation logic and gives the resource-oblivious specification. Section 4 describes the resource-aware specification. Section 5 extends the malloc/free interface with additional functions that can be used to ensure that malloc never fails. Section 6 explains subsumption between specifications, which means the code is verified just once though clients can rely on both the resource-aware and -oblivious specs. Section 7 elaborates on what exactly has been proved. Section 8 describes bugs uncovered through verification attempts. Section 9 assesses the verification effort, Section 10 considers future work, and Section 11 concludes.

Code and proofs are available at
https://github.com/PrincetonUniversity/DeepSpecDB
in the `memmgr` subdirectory.

2 The Algorithm and Data Structure

The implementation we describe in this section is meant to be clean, simple, efficient, but not innovative.

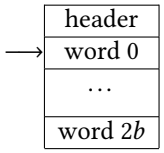
We use *small* blocks, in $BINS = 50$ different size classes, and *big* blocks. We pass the *big* requests directly to the mmap system call, both for malloc and free. This means we don't need to manage a free list of big blocks, and we have no fragmentation problem for big blocks.[2]

Let the word size $W =$ **sizeof**(size_t). In a standard C configuration, the blocks returned from malloc must be aligned at a multiple of $A \cdot W$, where typically $A = 2$.

[1] https://coq.inria.fr

[2] The operating system may be able to avoid fragmentation (for mmap'ed big blocks), by the use of virtual memory remapping.

For size classes $0 \le b < BINS - 1$ we have block sizes of $((b+1) \cdot A - 1) \cdot W$. One word is reserved immediately before each block for its header (to tell free the size of the block).



(assuming $A = 2$)

In a typical C configuration where $W = 8$ and $A = 2$, the small-block sizes are 8 bytes, 24 bytes, 40, ... 792 bytes. Changing the number of *BINS*, and hence the largest small-block size, requires a one-word change to the C program and *no* change to the proof of correctness.

When blocks are on the free list (of a particular size-class bin), we link them together using the field at offset 0 (labeled "word 0" in the diagram above). To initialize or replenish a free-list of size-class b, we ask mmap for a large region $BIGBLOCK = (2^{17} \cdot W)$, and divide it into a linked list of blocks. In the process, to satisfy alignment constraints, one word is wasted at the beginning of the region, and $(b+1)A-1$ words are wasted at the end of the region.

Large objects. We do not have a separate free list for large objects; instead, we outsource each large-object malloc or free to the mmap system call.

No coalescing. We do not coalesce freed blocks, for these reasons: Coalescing is costly (potentially adding a word of overhead to each block for a "footer.") [20, §2.5]. Coalescing can reduce, but cannot avoid, fragmentation, and thus could not improve our resource-tracking specification (see section 4). Some other modern allocators don't coalesce, perhaps for these reasons but also because coalescing would interfere with locality-improving optimizations (such as mimalloc's) and with shared metadata schemes. Splitting would be straightforward to add and to verify, but would complicate the resource-tracking specification.

Look Ma, no hands! Many malloc/free systems have extra safety checks: for example, footer words as well as headers, so that free can detect (in some cases) when there has been a buffer overrun. We deliberately do not. Our verified malloc/free is meant to be used with verified-memory-safe client code (or verified-correct clients, which are memory-safe as a corollary). Client code that is not fully memory-safe may trash the data structures (and thus the invariants) of the malloc/free code, rendering its verification meaningless.

3 Specification in Separation Logic

In this section we present the resource-oblivious specification, and in Section 4 we show the resource-aware specification.

Separation logic is a Hoare logic with judgments of the form {precondition}command{postcondition}, particularly

```
void *malloc(size_t nbytes) {
  if (nbytes > bin2size(BINS−1))
        return malloc_large(nbytes);
  else return malloc_small(nbytes);
}

static void *malloc_small(size_t nbytes) {
  int b = size2bin(nbytes);
  void *q;
  void *p = bin[b];
  if (!p) {
    p = fill_bin(b);
    if (!p) return NULL;
    else bin[b] = p;
  }
  q = *((void **)p);
  bin[b] = q;
  return p;
}

static void *fill_bin(int b) {
  size_t s = bin2size(b);
  char *p = (char *) mmap0(NULL, BIGBLOCK,
            PROT_READ|PROT_WRITE,
            MAP_PRIVATE|MAP_ANONYMOUS, −1, 0);
  if (p==NULL)
    return NULL;
  else
    return list_from_block(s, p, NULL);
}

static void free_small(void *p, size_t s) {
  int b = size2bin(s);
  void *q = bin[b];
  *((void **)p) = q;
  bin[b] = p;
}

void free(void *p) {
  if (p != NULL) {
    size_t s = (size_t)(((size_t *)p)[−1]);
    if (s <= bin2size(BINS−1))
          free_small(p,s);
    else free_large(p,s);
  }
}
```

Figure 1. Core of the allocator.

The computation bin2size(BINS−1) in malloc ought to be optimized to an integer constant; gcc does it, but CompCert's inliner doesn't manage it.

```
static void *list_from_block(size_t s, char *p, void *tl) {
  int Nblocks = (BIGBLOCK−WASTE) / (s+WORD);
  char *q = p + WASTE;
  int j = 0;
  while (j != Nblocks − 1) {
    ((size_t *)q)[0] = s;
    *((void **)(((size_t *)q)+1)) = q+WORD+(s+WORD);
    q += s+WORD;
    j++;
  }
  ((size_t *)q)[0] = s;
  *((void **)(((size_t *)q)+1)) = tl; /* link of last block */
  return (void*)(p+WASTE+WORD); /* link of first block */
}

static size_t bin2size(int b) {
  return ((b+1)*ALIGN − 1)*WORD;
}

static int size2bin(size_t s) {
  if (s > bin2size(BINS−1))
    return −1;
  else
    return (s+(WORD*(ALIGN−1)−1))/(WORD*ALIGN);
}
```

Figure 2. Core of the allocator, continued.

suited to programs that manipulate pointer data structures (and slices of arrays) in which aliases may occur. In conventional Hoare logic, if the assertion $p \mapsto x$ represents the condition that p points to a place in memory where the value x is stored, we might want to prove

$$\{p \mapsto x \wedge q \mapsto y\}\ *\mathsf{p} = \mathsf{z};\ \{p \mapsto z \wedge q \mapsto y\}$$

that is, if before the command p points to a value x and q points to y, then afterwards p points to z and q points to y. But if p and q are aliased ($p = q$), then the postcondition fails to hold (unless $y = z$).

In separation logic, an assertion holds on a particular *footprint* of the memory, and the "separating conjunction" $A * B$ says that A and B hold on disjoint footprints, where the footprint of $A * B$ is the union of the footprints of A and B. Then this judgment is sound:

$$\{p \mapsto x * q \mapsto y\}\ *\mathsf{p} = \mathsf{z};\ \{p \mapsto z * q \mapsto y\}$$

In separation logic, the natural rules for alloc and free are something like,

$$\{\mathsf{emp}\}\ p = \mathsf{alloc}()\ \{p \mapsto \_\}$$

$$\{p \mapsto \_\}\ \mathsf{free}(p)\ \{\mathsf{emp}\}$$

Here emp is the predicate true, considered as having the empty footprint.

Using SL's standard frame rule[3] (and the fact that emp is a unit for $*$), we can conclude by the following inference steps that the newly allocated block is disjoint from any block we could already reason about:

$$\frac{\{\mathsf{emp}\}\ p = \mathsf{alloc}()\ \{p \mapsto \_\}}{\{\mathsf{emp} * q \mapsto 6\}\ p = \mathsf{alloc}()\ \{p \mapsto \_ * q \mapsto 6\}}$$
$$\{q \mapsto 6\}\ p = \mathsf{alloc}()\ \{p \mapsto \_ * q \mapsto 6\}$$

The meaning of $p \mapsto x$, when the type of p is **char** and x is a small integer, is that a single byte of memory at address p contains value x. When p belongs to a structured type τ (struct, array, union, or int>char), we write $p \mapsto_\tau x$ to indicate that the footprint may be several (sizeof τ) bytes of memory, and x is an appropriately structured value. Finally, we write $\mapsto_{(n)}$ to abbreviate that the type is "length-n array of unsigned byte." Then we can write the specification of a multibyte alloc:

$$\{\mathsf{emp}\}\ p = \mathsf{alloc}(n)\ \{p \mapsto_{(n)} \_\}$$

$$\{p \mapsto_{(n)} \_\}\ \mathsf{free}(p, n)\ \{\mathsf{emp}\}$$

where an underscore indicates a don't-care value.

But C's malloc/free does not require a second argument to free; that information comes along with the block. To model that, we use a separation-logic assertion, the "malloc token". The assertion $\mathsf{mtok}(p, n)$ represents the "capability" to free a token of length n at address p, i.e., evidence that the block was obtained from malloc.[4]

Thus our specs take the form

$$\{\mathsf{emp}\}\ p = \mathsf{malloc}(n)\ \{\mathsf{mtok}(p, n) * p \mapsto_{(n)} \_\}$$

$$\{\mathsf{mtok}(p, n) * p \mapsto_{(n)} \_\}\ \mathsf{free}(p)\ \{\mathsf{emp}\}$$

When we write modular programs in VST's separation logic (called *Verifiable C*), module A may have extern global variables (or static global variables) that contain its persistent private data [3]. In correctness proofs of the clients of module A, we represent this abstract package by some suitable separation-logic predicate. For example, consider this simple intserver module:

```
unsigned int s;
unsigned int next(void) {return s++;}
```

The client-side specification of next could be,

$$\{\mathsf{intserver}\}\ \mathsf{j} = \mathsf{next}();\ \{\mathsf{intserver} * \exists i.\ \mathsf{j} \Downarrow i\}$$

and a more refined specification could be,

$$\{\mathsf{intserv}(i)\}\ \mathsf{j} = \mathsf{next}();\ \{\mathsf{intserver}(i+1) * \mathsf{j} \Downarrow i\}$$

[3] From $\{A\}$command$\{B\}$ infer $\{A * C\}$command$\{B * C\}$ for any command and any predicates A, B, C.

[4] The reader can also imagine that $\mathsf{mtok}(p, n)$ represents the header word $p[-1] \mapsto n$, but we want a sufficiently abstract specification that the representation of this capability is up to the malloc/free implementation. The idea has been independently rediscovered and seems to first appear in [30].

```
Definition malloc_spec' :=
 DECLARE _malloc
 WITH n:Z, gv:globals
 PRE [ _nbytes OF size_t ]
   PROP (0 <= n <= Ptrofs.max_unsigned − (WA+WORD))
   LOCAL (temp _nbytes (Vptrofs (Ptrofs.repr n)); gvars gv)
   SEP ( mem_mgr gv )
 POST [ tptr tvoid ] EX p:val,
   PROP ()
   LOCAL (temp ret_temp p)
   SEP (mem_mgr gv;
        if eq_dec p nullval then emp   (* p==NULL ? *)
        else (malloc_token' Ews n p * memory_block Ews n p)).

Definition free_spec' :=
 DECLARE _free
 WITH n:Z, p:val, gv: globals
 PRE [ _p OF tptr tvoid ]
   PROP ()
   LOCAL (temp _p p; gvars gv)
   SEP (mem_mgr gv;
        if eq_dec p nullval then emp   (* p==NULL ? *)
        else (malloc_token' Ews n p * memory_block Ews n p))
 POST [ Tvoid ]
   PROP ()
   LOCAL ()
   SEP (mem_mgr gv).
```

Figure 3. Resource-oblivious specs of malloc and free.

The server-side *definitions* of these predicates would be,

$$\mathsf{intserver} \;=\; \exists i.\; s \mapsto_{\mathsf{uint}} i$$
$$\mathsf{intserv}(i) \;=\; s \mapsto_{\mathsf{uint}} i$$

Our memory manager is no different: in proofs about clients, we represent the private state of the memory manager by a predicate mm. (Section 4 describes a more refined predicate rmm parameterized on a resource vector.)

Thus, the specifications of malloc/free now look like:

$$\{\mathsf{mm}\}\ p = \mathsf{malloc}(n)\ \{\mathsf{mm} * \mathsf{mtok}(p,n) * p \mapsto_{(n)} \_\}$$
$$\{\mathsf{mm} * \mathsf{mtok}(p,n) * p \mapsto_{(n)} \_\}\ \mathsf{free}(p)\ \{\mathsf{mm}\}$$

However, C's malloc function is permitted to return NULL if it wants to. Thus, the spec of malloc should be adjusted to:

$$\{\mathsf{mm}\}\ p = \mathsf{malloc}(n)\ \{\mathsf{mm} * ((p = \mathsf{NULL} \wedge \mathsf{emp}) \vee (\mathsf{mtok}(p,n) * p \mapsto_{(n)} \_))\}$$

The spec of free is also adjusted to allow NULL to be freed.

Verifiable C. is a program logic representable in ASCII, manipulated in the Coq proof assistant within an IDE. Its syntax is more heavyweight than the informal mathematical notation we have been using so far. The function-specifications in Verifiable C notation are shown in Figure 3.

Verifiable C function specs start with the C-program identifier for the function name, e.g. in Fig. 3 DECLARE _malloc. The WITH clause quantifies over variables (in this case n and gv) mentioned in precondition and postcondition. In this case, n (of type Z, or "mathematical integer") represents the *value* of the C parameter _nbytes, and gv gives access to the link-time addresses of whatever global variables the mem_mgr predicate needs to access. Then there is a PRECONDITION and POSTCONDITION.

The precondition is broken down into pure propositions PROP, variable bindings LOCAL, and spatial (memory) predicates SEP. Malloc's PROP precondition says that n must be in a certain range,[5] LOCAL says the function-parameter _nbytes contains an appropriate type-size_t representation of n, and SEP says that the client has access to the memory-manager resource.

The predicate memory_block Ews n p corresponds to what we are writing as $p \mapsto_{(n)} \_$. It includes a "permission share" Ews for concurrency, beyond the scope of this paper. We provide alternate specs in which the mtok and memory blocks are indexed by type, like the notation $\mapsto_\tau$.

Readers who would like to understand the specifications in detail are advised to consult the Verifiable C user's manual.[6]

4 Resource Tracking

One might like to prove that a client program is resource-bounded: the program never uses more than N bytes of memory at a time, so a properly initialized malloc/free system will never run out of memory—malloc will never return NULL.

Hypothetically, we could express this in separation logic with a parameter to the mm predicate. That is, rmm(N) represents a *resource-aware* malloc/free system that has N bytes available to allocate; the Hoare triples specifying malloc and free would look like this:

$$\{N \geq n \geq 0 \ \wedge\ \mathsf{rmm}(N)\}$$
$$\mathsf{p} = \mathsf{malloc(n)};$$
$$\{\mathsf{rmm}(N-n) * \mathsf{mtok}(p,n) * p \mapsto_{(n)} \_\}$$
$$\{\mathsf{rmm}(N) * \mathsf{mtok}(p,n) * p \mapsto_{(n)} \_\}$$
$$\mathsf{free(p)};$$
$$\{\mathsf{rmm}(N+n)\}$$

[5] The upper bound must surely be at most the maximum unsigned int. Also, it cannot exceed the largest block that can be obtained from mmap—which is not specified in the Posix standard, so our spec for mmap uses max_unsigned. Whatever that size is, we must deduct WA+WORD to allow for header and for alignment waste (or else we could add a bounds check in the code and return NULL in this corner case—which would waste cycles).

[6] https://github.com/PrincetonUniversity/VST/raw/master/doc/VC.pdf

Unfortunately, there is the well-known problem of fragmentation. The client might allocate k small blocks of size n; suppose they are allocated consecutively in memory. The client might then free half of them, the "odd-numbered" ones; then allocate $k/2$ blocks of size $2n$. The larger blocks cannot reuse any of the recently freed space.

In a system that cannot move allocated blocks, fragmentation cannot be avoided. C programs in general cannot tolerate the movement of already allocated blocks.

Robson [31] proved that any memory allocator that uses 2^a different block sizes may require, in the worst case, memory that is a times as large as the maximum simultaneous allocated data (times a constant factor). Therefore if we have a simple rmm(N) predicate, with our $2^a \sim 50$ bins ($a = 6$) the best possible coalescing algorithm would consume proportional to aN words of storage. We do no coalescing at all, so our current implementation would not necessarily achieve even that bound.

That is, even with coalescing, fragmentation cannot be avoided, and we could not (efficiently) use a scalar N parameter as proposed above.

To avoid a larger-than-constant-factor waste of memory, we ask the user to do more refined resource tracking. Let V be a *resource vector*: for $0 \le i < BINS$, the slot $V(i)$ tracks the number of available (mallocable) blocks whose size is between bin2Size($i-1$) and bin2Size(i).

For block size n, let $S(n)$ be the bin number, such that bin2Size($S(n)-1$) $< n \le$ bin2Size($S(n)$). Then the specs using the vector-resource predicate rmm are,

$$\begin{array}{c}\{S(n) = i \ \wedge \ V(i) > 0 \ \wedge \ \mathsf{rmm}(V)\}\\ \mathsf{p} = \mathsf{malloc(n)};\\ \{\mathsf{rmm}(V[i := V(i)-1]) * \mathsf{mtok}(p,n) * p \mapsto_{(n)} \_\}\end{array}$$

$$\begin{array}{c}\{S(n) = i \ \wedge \ \mathsf{rmm}(V) * \mathsf{mtok}(p,n) * p \mapsto_{(n)} \_\}\\ \mathsf{free(p)};\\ \{\mathsf{rmm}(V[i := V(i)+1])\}\end{array}$$

The notation $V[i := \ldots])$ denotes update of a mapping.

In practice, even if the V resource is used up, malloc might still be able to return a block. An alternate specification can explain that. In Section 6 we show how to relate several different specifications for the same code, while verifying the code only once.

5 Filling the Resource Vector

The resource vector V allows one to prove that a program (client of malloc/free) stays within its resource bound, and therefore malloc never returns NULL. In turn, this may simplify reasoning about the client, since there is no need for error-checking malloc calls—which is not so hard in itself, the hard part is reasoning about cleaning the abnormal exit. Furthermore, we would like to prove that some programs can keep running indefinitely.

But in the standard case, malloc/free starts with no resources at all, and obtains them by calling mmap from time to time, which may return "no." We therefore augment the API with a new function,

void pre_fill(size_t n, **void** *p);

with the specification,

$$\frac{\begin{array}{c}0 \le n \le N \wedge S(n) = b \wedge \ \mathsf{malloc\_compat}\ B\ p\\ \wedge\ k = (B - W\cdot(A-1))/(W + \mathsf{bin2size}(b))\end{array}}{\{\mathsf{rmm}(V) * p \mapsto_{(B)} \_\}\ \mathsf{pre\_fill}(n,p)\ \{\mathsf{rmm}(V + [b \mapsto k])\}}$$

That is, the client program calls pre_fill with a suitably aligned ("malloc compatible") big block (of size B) at address p, and asks that it be divided into k small blocks (of size n) that are added to the free list.

Clients are expected to obtain such big-blocks either from their own BSS segment, or from calls to mmap at the beginning of their own execution. Such a client, therefore, would start by calling mmap the appropriate number of times; if those calls succeed, then there will be no allocation failures during the rest of program execution.

The spec of pre_fill is not very abstract! It requires clients to know concrete values such as B, A, bin2size(b), and so on. We have also provided a wrapper (try_pre_fill) that simply takes n and k, calculates the right number of calls to mmap, performs those calls, and calls pre_fill on each result. That function has a simpler and more portable specification, and is proved correct solely based on the specifications (not the implementations) of pre_fill and mmap.

$$\frac{\begin{array}{c}0 \le n \le \mathsf{maxSmallChunk} \qquad 0 \le k \le \mathsf{maxInt}\\ S(n) = b\end{array}}{\{\mathsf{rmm}(V)\}\ r = \mathsf{try\_pre\_fill}(n,k);\ \{\mathsf{rmm}(V + [b \mapsto r])\}}$$

This specification says that r blocks of size n have been added to the resource vector. If $r < k$ that is because a call to mmap failed; the client program can therefore take appropriate action *before* entering the (complicated) main body of its computation. It can happen that r exceeds k, by rounding up to a big block.

6 Funspec Subsumption

We now have two different specs for malloc, the resource-aware one using rmm:

$$\begin{array}{c}\{S(n) = i \ \wedge \ V(i) > 0 \ \wedge \ \mathsf{rmm}(V)\}\\ \mathsf{p} = \mathsf{malloc(n)};\\ \{\mathsf{rmm}(V[i := V(i)-1]) * \mathsf{mtok}(p,n) * p \mapsto_{(n)} \_\}\end{array}$$

and the original one using mm:

$$\begin{array}{l}\{\mathsf{mm}\}\ p = \mathsf{malloc}(n)\\ \quad \{\mathsf{mm} * ((p = \mathsf{NULL} \wedge \mathsf{emp}) \vee (\mathsf{mtok}(p,n) * p \mapsto_{(n)} \_))\}\end{array}$$

Clients that do not need resource accounting are more conveniently verified against the mm specification, and those that want stronger guarantees can be verified against rmm.

The same implementation of malloc can satisfy either spec. But we don't want to do two verifications of the same code! In fact we use a third, stronger spec to verify malloc, which uses rmm but accounts for both successful and unsuccesful calls. Fortunately, we can prove that the mm and rmm specs described earlier are implied by the stronger spec (and similarly for free). In the Verifiable C program logic, this relation is called subsumption (funspec_sub) [3]. We proved the C code (of malloc and free, and supporting functions) correct, line by line, with respect to the strong specs. We proved funspec_sub theorems for the alternative specs, so clients can be verified using whichever are convenient.

Our software distribution includes sample clients of both kinds, with end-to-end linking proofs that tie everything together.

7 Formal Guarantees w.r.t. C Standard

Malloc/free systems test the dark corners of the C language definition: casting, alignment, max object size,[7] just-past-the-end-of-an-array pointers, signed integer overflow, and so on. Harmless-looking violations of these rules can confuse compilers into generating unintended code [35]. Correctness verification of a malloc/free system *must* be in a verifier that correctly enforces all of these rules, otherwise it's not worth the trouble.

The Verified Software Toolchain includes a tool for applying the Verifiable C program logic. (Technically, Verifiable C is a higher-order impredicative ghostly concurrent separation logic, but our malloc/free is neither object-oriented nor concurrent so in this verification we have not used the higher-order impredicative concurrent features.) In addition, VST has a machine-checked proof of soundness of Verifiable C with respect to the operational semantics of CompCert C light, which is a source-language specification for correctness of the CompCert verified optimizing C compiler. CompCert C light is a careful and accurate formalization of C11.[8]

Since the CompCert compiler is proved correct (in Coq) with respect to CompCert C (and CompCert C light), one has the guarantee that for any C program proved correct by VST w.r.t. a function specification ϕ, all behaviors of the compiled assembly respect ϕ.

[7] C restricts on the maximum size of a single object, and has additional rules meant to ensure that no object crosses the boundary 0xffffffff to 0x00000000 (e.g., on a 32-bit machine). C's alignment rules are particularly tricky, in order to accommodate legacy architectures on which 64-bit integers could be aligned at 32-bit boundaries.

[8] There are some differences. In particular, CompCert C ascribes defined behavior to sequence point violations and signed integer overflow, and defines parameter evaluation as left-to-right. In these cases, Verifiable C is still sound w.r.t. C11: Verifiable C programs have neither read nor write side effects in expressions, so sequence points and parameter evaluation order are irrelevant; and Verifiable C enforces the absence of signed integer overflow, as a proof obligation for the user.

```
Definition mmap0_spec :=
DECLARE _mmap0
WITH n:Z
 PRE [ 1 (*_addr*) OF (tptr tvoid),
       2 (*_len*) OF tuint,
       3 (*_prot*) OF tint,
       4 (*_flags*) OF tint,
       5 (*_fildes*) OF tint,
       6 (*_off*) OF tlong ]
   PROP (0 <= n <= Ptrofs.max_unsigned)
   LOCAL (temp 1 nullval;
          temp 2 (Vptrofs (Ptrofs.repr n));
          temp 3 (Vint (Int.repr 3)); (* PROT_READ|PROT_WRITE *)
          (* temp 4 is MAP_PRIVATE|MAP_ANONYMOUS *)
          temp 5 (Vint (Int.repr (-1)));
          temp 6 (Vlong (Int64.repr 0)))
   SEP ()
 POST [ tptr tvoid ] EX p: val,
   PROP ( if eq_dec p nullval  (* p==NULL ? *)
          then True else malloc_compatible n p )
   LOCAL (temp ret_temp p)
   SEP ( if eq_dec p nullval  (* p==NULL ? *)
         then emp else memory_block Tsh n p).
```

Figure 4. Specification of mmap0. As the Posix specification of mmap requires unspecified behavior by the client (testing return value for -1 or pointer), we write a wrapper with a different error-return indicator; and we specify here only the subset of the mmap functionality that we need.

Our malloc/free system, like any VST-verified program, can also be compiled with `gcc` or `clang`. In such cases, one still gets strong guarantees of the correctness of the source code.

8 Bugs

The Verified Software Toolchain has a soundness guarantee: if you use Verifiable C to prove that your program satisifes some functional specification, then the program is *safe* (no undefined behavior) and *correct* (inputs and outputs match the functional spec). In contrast, an *unsound* static analyzer can never guarantee the safety of your program—it can only point out places that might be bugs. Unsound static analyzers are sometimes disparagingly called bug finders.

Still, finding bugs is useful enough, and in the process of doing the formal verification we found bugs in our code and a flaw in the Posix standard.

- Function list_from_block builds a linked list of small chunks by iterating over a large block, with variable q pointing to the next small chunk. The loop condition q+s+WORD < p+BIGBLOCK was used in an early version of our code. But the integer sum p+BIGBLOCK

can overflow if the address p is very large. This would cause misbehavior by the allocator and is very unlikely to be found by testing. The logic's "type check" requirements, which ensure defined C behavior, could not be proved for this code.

- An attempt to solve the preceding issue used this loop condition: q < p+(BIGBLOCK−(s+WORD)). But this still could overflow. Again, the undefined behavior becomes evident in a proof attempt but would probably not be found by testing. Of course runtime checks can be used to detect overflow, but the performance cost is undesirable.
- Our first implementation of try_pre_fill had a signed-integer-overflow bug. This was caught during the proof, and might not have been caught during testing.
- Again in list_from_block, the following code sets a link field to point to the link field of the following block:

```
*((void **)(((size_t *)q)+1)) = q+WORD+(s+WORD);
```

  In early versions of the code, the last +WORD was missing, and was found during verification attempt. Our slapdash unit tests did not reveal the bug; thorough testing would have found it.
- We write functional specifications of the system calls we use (mmap and munmap), closely following the Posix documentation. Accordingly, we spec'd mmap to return −1 on failure, and a pointer on success. But the Verifiable C program logic refuses to verify any code that tests p==−1, because that's unspecified in C11.[9] Thus, Posix mmap's API is not very portable. We worked around this problem by writing a shim, mmap0, that returns NULL on failure—which is sufficient for our use of mmap.[10]
- Even if we did not need this shim, the implementation of mmap itself is not verified—we just assume a specification for it. One could address this gap in three ways:
  1. Tolerate the assumption that the OS correctly implements the system call.
  2. The user can avoid using mmap at all, by using our (verified) pre_fill function, using memory from the BSS segment.
  3. One could verify the operating system's implementation of mmap, and verify the system-call interface between the client program and the OS. Mansky *et al.* [25] show how to do this for IO system-calls, and their technique would extend to mmap.
- VST proofs operate on the macro-expanded C code. The proof *may* be portable, but the proof script will need to be rerun for each configuration. Because the values of some Posix flags for mmap differ between Linux and macOS, our proof script was not portable. The mmap0 workaround described above solved this problem too, as we do not need specific values for the flags (Figure 4).
- Before we built a verified malloc/free library, we used VST to verify several programs that are clients of the library: B-trees, hash tables, binary search trees, queues, etc. To do that, we wrote separation-logic specs for malloc and free. The spec for free assumed a non-NULL argument, whereas the Posix standard (and our current version) allows free(NULL). This illustrates a core observation from the DeepSpec project: until you have exercised a specification *from both sides*, you've probably got it wrong ([12], deepspec.org). That is, that specs should be evaluated through use by clients, not just used to verify implementations. However, this particular mismatch was not discovered through use of VST but rather through attempting to write specs in accord with Posix.
- Prior to our work, the specs of malloc and free provided by VST featured the malloc token but were missing a separated conjunct for the memory manager's invariant on internal structure, written mm in this paper. Initial versions of the specs used the malloc token as explained earlier, but failed to include the memory wasted due to alignment and unusable space at the end of big blocks. Once again: specs are not well justified until they have been both implemented and used by clients.

9 Verification Effort

The engineer(s) doing a VST verification proceeds as follows:

1. Write a *functional model* in Coq for the abstract computation. For the malloc/free system, this step is quite simple and we can almost neglect it.
2. Write *representation relations* showing how the functional model relates to data structures in the C program's memory; prove useful lemmas about these representation relations. (This is the "abstractions" line of Figure 5.)
3. Write *function specifications* for each C function, whether part of the API or internal to a module. (This is the "Spec" column of Figure 5.)

[9] That is, §6.3.2.3 of the C11 standard allows comparison of pointers with NULL (that is, 0 cast to a pointer type); but comparing a pointer for equality with (void*)-1 is implementation-defined (by 6.3.2.3.6). The only specification of the behavior is the footnote, "The mapping functions for converting ... an integer to a pointer are intended to be consistent with the addressing structure of the execution environment." Recent research [17] has suggested how the semantics of C integer-pointer casts could be formalized; if CompCert and then VST were to adopt this approach, we could adequately formalize the return value of mmap.

[10] Our current tools cannot verify the correctness of this simple shim in C, for the reasons explained, but we could perhaps verify its assembly-language implementation.

Function	C	Spec	Verif	notes
abstractions		29	1222	a.
mmap0	1	22	n/a	b.
munmap	1	13	n/a	b.
malloc (rmm)	6	20	50	c.
malloc (mm)		13	39	d.
free (rmm)	9	17	42	c.
free (mm)		13	29	d.
pre_fill	5	12	99	
try_pre_fill	20	12	124	
bin2size	3	8	12	
size2bin	6	8	12	
list_from_block	14	11	394	
fill_bin	9	11	60	
malloc_small	15	18	343	
malloc_large	10	13	86	
free_small	6	15	135	
free_large	3	15	86	
Overall	108	49	2733	e.

Figure 5. Lines of code: implementation (C), specification (Spec), verification (Verif).

- Nonblank, noncomment lines of C or Coq code.

a. Definitions of mtok, rmm, mm, and various supporting lemmas.
b. Since we axiomatize these system calls, there is no C code or proof.
c. Includes the rmm-based spec, and function-body proof.
d. Includes the mm-based spec, and only the funspec_sub proof.
e. "Spec" does not include internal funspecs, only the rmm-based specs of malloc, free, and pre_fill.

4. Prove that each C function body satisfies its function-spec, one function-body at a time. This is a forward Hoare-logic proof, written down interactively as a proof script using the VST-Floyd tactics. The tactics automate symbolic execution, but the proof engineer must provide loop invariants, and direct other reasoning of the proof system. (This is the "Verif" column of Figure 5.)

Overall it is a laborious but feasible process. This verification technique can be used for small-to-medium software where extremely high assurance is worth paying for; it is not feasible (yet) for a million lines of code.

Figure 5 shows the size of the C code (88 lines), its specification in separation logic (49 lines), and the proof script in Coq (2733 lines). VST proofs are typically rather verbose (an order of magnitude more lines of proof than lines of code), but in this case the proof is particularly lengthy (compared to the C code). We believe the reason for this is that a malloc/free program is particularly abusive of the C type system: casting undifferentiated memory into linked data structures, returning pointers that are one past the header word, making sure that those pointers are double-word aligned, and so on. All this abuse is *legal* in C11, but needs formal justification. In contrast, the proofs of more "well behaved" C code are a bit more automated by VST's proof tactics [7].

Checking all the proofs (including sample clients) takes under 5min real time, running Coq on a commodity laptop (macOS 2.8 GHz Quad-Core Intel Core i7).

10 Future Work

There are several high-performance implementations of malloc/free; could one of them be verified correct? We take mimalloc [21] as an example.

Like most modern allocators, mimalloc supports multi-threaded applications, and uses thread-local free lists to avoid the need to synchronize on every malloc or free. To transfer a batch of free-list entries from one thread to another, it synchronizes using atomic compare and swap, in a combination of relaxed mode (for failure) and release-acquire mode (for success).

Our VST toolchain supports both semaphores and SC-mode atomics [8, 24]. To verify these synchronizations, we have the choice of three approaches:

1. Convert the atomic CAS to SC mode, which will slightly degrade performance (though not in any of mimalloc's fast paths), and use VST's existing logic. This would be straightforward.
2. Axiomatize in VST the separation-logic rules for relaxed and release-acquire modes developed by Dang *et al.* [9], and use them to prove the mimalloc synchronizations without modification.
3. Import the model of Dang *et al.* from Iris into VST, and use them to prove correctness of the separation-logic rules for relaxed and release-acquire modes, then *foundationally* prove the synchronizations without modification.

Like many allocators, mimalloc uses *shared metadata*: that is, instead of a header word at address p−**sizeof**(size_t), there are no header words: all the objects on a page have the same size; the page number is found by masking out the low-order bits of p, then that is looked up in a table to learn the object size. To verify this in VST, we must address two issues:

1. Can we reason about pointer-integer casts? In CompCert's semantics for C (which we use also in our VST formal-reasoning toolchain), the answer is no. Kang *et al.* [17] show an extension of a CompCert-like memory model that permits such reasoning. One way forward would be to apply this technique in CompCert itself, then extend VST with the corresponding reasoning theorems. A less foundational (but still adequate) approach would be to leave CompCert alone, but extend VST with axioms instead of theorems, relying on Kang's result as demonstration of the consistency of these axioms.

2. Can the same "malloc token" interface mtok(p, n) represent something other than a header word at address p−**sizeof**(size_t)? Yes! Since mtok is abstract (to the client), we can use the expressive power of Verifiable C's separation logic (once extended with pointer-integer casts) to give an alternate representation for mtok, based on shared-metadata calculations. Wickerson *et al.* [37] do something similar with rely-guarantee reasoning; we could use Verifiable C's *ghost state* to accomplish the same thing.

Unlike many allocators, mimalloc has novel techniques to improve the memory locality of the client program. Instead of one free-list per size class, memory is divided into 64-KB *shards*, each with its own free list. A thread's successive mallocs are allocated from within the same shard, which means that related objects will be in the same shard. Furthermore, newly freed objects are not pushed on the head of the same free list that is used for malloc, as that would degrade the locality pattern. All of this should be quite straightforward to reason about in separation logic.

In freshly created shards, mimalloc creates a fully populated free-list in advance, rather than incrementally consuming the region. Our allocator does the same, and for the same reason: it removes one comparison from the fast path of malloc. (mimalloc does it for other reasons as well, regarding address-order of allocation.)

Finally, mimalloc is small: the core library is less than 3500 LOC. This is well within the scale of what is feasible to verify.

Estimated verification effort for 3500 LOC.. L4.verified [19] is an operating-system microkernel, written in 7500 lines of C and verified in Isabelle/HOL (another 1000 lines is not verified). The verification was 200K lines of proof script, taking 25 person-years (of which much was tool development, training, and learning). The authors estimate that (with tools and techniques already developed) a similar project would take 10 person-years.

CertiKOS [13] is a multicore hypervisor kernel, written in 6500 lines of C and verified in Coq using the Certified Abstraction Layers method. The verification is 282K lines of Coq specs and proof scripts, and took about 3.5 person-years (of which, the extension to concurrency comprised 95 kLOC Coq and 2 person-years).

A visiting master's student at Princeton verified an implementation of B+-trees with a comprehensive set of operations. After two weeks learning VST (and a previous knowlege of Coq), verification of 570 (nonblank, noncomment) lines of C code took about 4 person-months, with about 9500 (nonblank, noncomment) lines of proof scripts.

Our own malloc/free verification (reported in this paper) of 108 lines of C took *approximately* 2 person-months, not including the time to learn Coq and VST—it was (mostly) done by the second author, who had never previously used VST. Further down the learning curve, verification effort should be substantially reduced. Furthermore, verification effort was higher for reasoning about code that abuses the C type system (but not illegally) to do the address arithmetic for dividing large pages into smaller blocks of calculated size with headers. Separation-logic reasoning about segregating these into shards should be more straightforward. For a system like mimalloc, we estimate about 36 person-months of effort would be required.

11 Conclusion

We have verified an implementation in C of a malloc/free system using the standard array-of-bins representation for efficiency. The proof is machine checked and establishes correctness of the assembly code produced by the verified CompCert compiler: our verification tool is proved sound with respect to the CompCert's formalization of C11 which includes address alignment, bounded arithmetic, etc. Our code is verified with respect to precise specifications of malloc and free that track the size of free lists. These specs are then proved to subsume simpler specs that formalize the standard API and should be used to verify most clients. We also augment that API with a verified function that pre-fills a free list, and provide alternate resource-aware specs of malloc/free that make it possible to prove malloc never returns null provided the client stays within the bounds it has prefilled. Subsumption connects these specs with the precise ones.

Unless the client uses pre-fill with memory from its own process space (BSS segment), the malloc/free system relies on the mmap system call to obtain large blocks. Our verification is with respect to a formal spec that conforms with the Posix standard, but we know of no verified implementation of mmap.

The current implementation is single-threaded but the specs are already formulated in terms of separation logic "shares", enabling client threads to share ownership of allocated blocks, and making the specifications naturally concurrency-compatible. Verifiable C supports proofs about concurrent shared-memory programs with locks (and has *some* support for the C11 atomics). In future work we plan make the malloc/free system thread safe.

A more challenging goal is to verify a more highly engineered malloc/free system. A promising target is mimalloc [21], which achieves very high performance for a wide range of work loads through free list sharding. Its implementation is relatively compact, in part owing to uniform representation of objects at all sizes. Important system code like this should be verified, foundationally, and we have shown that it is within reach.

Acknowledgments

This work was supported in part by National Science Foundation Grants CCF-1521602 and CNS-1718713. The paper

has been improved thanks to extensive constructive feedback from anonymous reviewers and our shepherd Maoni Stephens.

References

[1] David Aspinall and Adriana B. Compagnoni. 2003. Heap-Bounded Assembly Language. *J. Autom. Reasoning* 31, 3-4 (2003), 261–302.

[2] Gilles Barthe, Mariela Pavlova, and Gerardo Schneider. 2005. Precise analysis of memory consumption using program logics. In *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*. IEEE, 86–95.

[3] Lennart Beringer and Andrew W. Appel. 2019. Abstraction and Subsumption in Modular Verification of C Programs. In *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Proceedings (LNCS)*, Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira (Eds.), Vol. 11800. Springer, 573–590. https://doi.org/10.1007/978-3-030-30942-8_34

[4] Lennart Beringer, Martin Hofmann, Alberto Momigliano, and Olha Shkaravska. 2004. Automatic Certification of Heap Consumption. In *LPAR'04: Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, Proceedings (LNCS)*, Franz Baader and Andrei Voronkov (Eds.), Vol. 3452. Springer, 347–362. https://doi.org/10.1007/978-3-540-32275-7_23

[5] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. 2015. Verified Correctness and Security of OpenSSL HMAC. In *24th USENIX Security Symposium*. USENIX Assocation, 207–221.

[6] Lars Birkedal, Noah Torp-Smith, and John C Reynolds. 2004. Local reasoning about a copying garbage collector. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 220–231.

[7] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *J. Autom. Reason.* 61, 1-4 (June 2018), 367–422. https://doi.org/10.1007/s10817-018-9457-5

[8] Santiago Cuellar, Nick Giannarakis, Jean-Marie Madiot, William Mansky, Lennart Beringer, Qinxiang Cao, and Andrew W. Appel. 2020. *Compiler Correctness for Concurrency: from concurrent separation logic to shared-memory assembly language.* Technical Report TR-014-19. Princeton University Computer Science. https://www.cs.princeton.edu/research/techreps/TR-014-19

[9] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019. RustBelt meets relaxed memory. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–29.

[10] Yi Feng and Emery D. Berger. 2005. A Locality-Improving Dynamic Memory Allocator. In *2005 Workshop on Memory System Performance*. 68–77. https://doi.org/10.1145/1111583.1111594

[11] Peter Gammie, Antony L Hosking, and Kai Engelhardt. 2015. Relaxing safely: verified on-the-fly garbage collection for x86-TSO. *ACM SIGPLAN Notices* 50, 6 (2015), 99–109.

[12] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 595–608. https://doi.org/10.1145/2676726.2676975

[13] Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan (Newman) Wu, Vilhelm Sjöberg, and David Costanzo. 2019. Building certified concurrent OS kernels. *Commun. ACM* 62, 10 (2019), 89–99. https://doi.org/10.1145/3356903

[14] Martin Hofmann and Steffen Jost. 2003. Static prediction of heap space usage for first-order functional programs. In *POPL '03: 30th ACM Symp. on Principles of Programming Languages*. 185–197.

[15] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods Symposium*. Springer, 41–55.

[16] Ke Jiang, David Sanan, Yongwang Zhao, Shuanglong Kan, and Yang Liu. 2019. A Formally Verified Buddy Memory Allocation Model. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 144–153.

[17] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. 2015. A Formal C Memory Model Supporting Integer-Pointer Casts. In *PLDI'15: 36th annual ACM SIGPLAN conference on Programming Languages Design and Implementation*. 326–335.

[18] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Aspects of Computing* 27, 3 (01 May 2015), 573–609. https://doi.org/10.1007/s00165-014-0326-7

[19] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 207–220.

[20] Donald E. Knuth. 1973. *The Art of Computer Programming, Vol. I: Fundamental Algorithms (second edition).* Addison Wesley, Reading, MA.

[21] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. 2019. Mimalloc: Free List Sharding in Action. In *Asian Symposium on Programming Languages and Systems*. Springer, 244–265.

[22] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

[23] Paul Liétar, Theodore Butler, Sylvan Clebsch, Sophia Drossopoulou, Juliana Franco, Matthew J. Parkinson, Alex Shamis, Christoph M. Wintersteiger, and David Chisnall. 2019. snmalloc: a message passing allocator. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management (ISMM)*. 122–135.

[24] William Mansky, Andrew W. Appel, and Aleksey Nogin. 2017. A Verified Messaging System. In *Proceedings of the 2017 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '17)*. ACM.

[25] William Mansky, Wolf Honoré, and Andrew W. Appel. 2020. Connecting Higher-Order Separation Logic to a First-Order Outside World. In *ESOP'20: European Symposium on Programming*.

[26] Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. 2006. Formal Verification of the Heap Manager of an Operating System using Separation Logic. In *SPACE 06: Third workshop on Semantics, Program Analysis, and Computing Environments for Memory Management*.

[27] Andrew McCreight, Tim Chevalier, and Andrew Tolmach. 2010. A certified framework for compiling and executing garbage-collected languages. In *ICFP'10: 15th ACM SIGPLAN International Conference on Functional programming*. 273–284.

[28] Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. 2007. A general framework for certifying garbage collectors and their mutators. In *PLDI'07: 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 468–479.

[29] Peter O'Hearn, John Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL'01: Annual Conference of the European Association for Computer Science Logic*. 1–19. LNCS 2142.

[30] Matthew J. Parkinson and Gavin M. Bierman. 2005. Separation logic and abstraction. In *32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*. 247–258. https://doi.org/10.1145/1040305.1040326

[31] J. M. Robson. 1971. An Estimate of the Store Size Necessary for Dynamic Storage Allocation. *J. Association for Computing Machinery* 18,

3 (July 1971), 416–423.

[32] Harvey Tuch. 2009. Formal verification of C systems code. *Journal of Automated Reasoning* 42, 2-4 (2009), 125–187.

[33] Gijs Vanspauwen and Bart Jacobs. 2017. Verifying cryptographic protocol implementations that use industrial cryptographic APIs. *CW Reports* (2017).

[34] Shengyi Wang, Qinxiang Cao, Anshuman Mohan, and Aquinas Hobor. 2019. Certifying Graph-Manipulating C Programs via Localizations within Data Structures. In *Proceedings of the ACM on Programming Languages (OOPSLA)*.

[35] Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards optimization-safe systems: analyzing the impact of undefined behavior. In *Proceedings 24th ACM Symposium on Operating Systems Principles*. ACM, 260–275.

[36] Charles B. Weinstock. 1976. *Dynamic storage allocation techniques*. Ph.D. Dissertation. Carnegie Mellon University.

[37] John Wickerson, Mike Dodds, and Matthew J. Parkinson. 2010. Explicit Stabilisation for Modular Rely-Guarantee Reasoning. In *European Symposium on Programming (ESOP)*. 610–629.

[38] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. 2017. Verified Correctness and Security of mbedTLS HMAC-DRBG. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*. ACM.

[39] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. 2017. A Formally Verified NAT. In *SIGCOMM'17: Proceedings of the conference of the ACM Special Interest Group on Data Communication*. 141–154.

[40] Yu Zhang, Yongwang Zhao, David Sanan, Lei Qiao, and Jinkun Zhang. 2019. A Verified Specification of TLSF Memory Management Allocator Using State Monads. In *Dependable Software Engineering. Theories, Tools, and Applications*, Nan Guan, Joost-Pieter Katoen, and Jun Sun (Eds.). Springer International Publishing, 122–138.