Code-Specify-Test-Debug-Prove: Flexibly Integrating Separation Logic Specification into Conventional Workflows

With Supplementary Material

ZAIN K AAMER*, University of Pennsylvania, USA
RINI BANERJEE*, University of Cambridge, UK
HIROYUKI KATSURA*, University of Cambridge, UK
DAVID KALOPER-MERŠINJAK*, University of Cambridge, UK
DIMITRIOS J. ECONOMOU, University of Cambridge, UK
KAYVAN MEMARIAN, University of Cambridge, UK
DHRUV MAKWANA, University of Cambridge, UK
NEEL KRISHNASWAMI, University of Cambridge, UK
BENJAMIN C. PIERCE, University of Pennsylvania, USA
CHRISTOPHER PULTE, University of Oxford, UK
PETER SEWELL, University of Cambridge, UK

We seek to enable more flexible use of rich specifications in a variety of ways that smoothly extend conventional software development practice. We show how a single specification language, based on separation logic to capture the subtle ownership disciplines of systems code, can be used for runtime assertion checking, for property-based testing, and for formal machine-checked proof—and how each of these complements and supports the others. We demonstrate all this on a challenging example: a component of a production hypervisor, running both stand-alone at user level and *in situ* in the hypervisor.

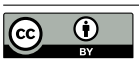
CCS Concepts: • **Software and its engineering → Formal software verification**; **Empirical software validation**; • **Theory of computation → Program reasoning**; **Automated reasoning**; **Separation logic**.

Additional Key Words and Phrases: C, specification, runtime testing, property-based testing, separation logic, refinement types, verification, pKVM, Android

ACM Reference Format:
Zain K Aamer, Rini Banerjee, Hiroyuki Katsura, David Kaloper-Meršinjak, Dimitrios J. Economou, Kayvan Memarian, Dhruv Makwana, Neel Krishnaswami, Benjamin C. Pierce, Christopher Pulte, and Peter Sewell. 2026. Code-Specify-Test-Debug-Prove: Flexibly Integrating Separation Logic Specification into Conventional Workflows: With Supplementary Material. *Proc. ACM Program. Lang.* 10, PLDI, Article 200 (June 2026), 39 pages. https://doi.org/10.1145/3808278

*These authors contributed equally.

Authors' Contact Information: Zain K Aamer, zaamer@seas.upenn.edu, University of Pennsylvania, Philadelphia, USA; Rini Banerjee, rb2018@cam.ac.uk, University of Cambridge, UK; Hiroyuki Katsura, Hiroyuki.Katsura@cl.cam.ac.uk, University of Cambridge, UK; David Kaloper-Meršinjak, dk505@cl.cam.ac.uk, University of Cambridge, UK; Dimitrios J. Economou, dje45@cl.cam.ac.uk, University of Cambridge, UK; Kayvan Memarian, Kayvan.Memarian@cl.cam.ac.uk, University of Cambridge, UK; Dhruv Makwana, dcm41@cam.ac.uk, University of Cambridge, UK; Neel Krishnaswami, nk480@cl.cam.ac.uk, University of Cambridge, UK; Benjamin C. Pierce, bcpierce@seas.upenn.edu, University of Pennsylvania, Philadelphia, USA; Christopher Pulte, christopher.pulte@cs.ox.ac.uk, University of Oxford, UK; Peter Sewell, Peter.Sewell@cl.cam.ac.uk, University of Cambridge, UK.



ACM 2475-1421/2026/6-ART200
https://doi.org/10.1145/3808278

1 Introduction

The conventional software development lifecycle—code, test, debug—is a successful methodology for building useful systems. However, it manifestly fails to make systems robust or secure enough for today's adversarial environment. Ever since the never-ending "software crisis" arrived in the late 1960s [57], researchers and practitioners have developed a wide variety of approaches to try to make software more reliable, varying along many axes.

Among the most important of these axes is whether a given approach smoothly extends conventional practice or whether it requires radically different methods. Many bug-finding tools (including static analysers like Infer [11] and CBMC [19] and dynamic analysers like Valgrind [58] and ASan [69]) can be added to conventional workflows as push-button tools that can be used without much specialist expertise. This is enabled by their focus on implicit and generic specifications, in particular the absence of particular classes of memory-safety flaws or other undefined behaviours. By contrast, traditional formal verification tools support—and require—writing much richer specifications, including "intermediate" specifications like loop invariants, that capture specific properties of the code. Historically these have been used solely for machine-checked proof, which gives high assurance but requires expertise that is still uncommon. Conventional wisdom is that one has to develop a system hand-in-hand with its specification and proof—John Reynolds once remarked that there are two kinds of programs: those developed in concert with their correctness proofs and those that cannot be verified at all. In particular, discovering the intermediate specifications required to verify existing code is difficult, especially when feedback comes only from failed proof attempts in a specify-code-prove workflow. All this has made formal verification hard to deploy at scale and hard to retrofit to existing systems.

More generally, most work aiming at more rigorous software development is siloed, focussed on a single approach and tool. This can of course be necessary to make progress, but it leaves things disconnected, and it is regrettably culturally embedded—in particular, the communities working on testing and on proof have been largely disjoint.

In this paper, we seek to enable *more flexible* use of rich specifications, in ways that do smoothly extend conventional practice. We show how a single specification language based on separation logic, designed to capture the subtle ownership disciplines of systems code, can be used in runtime assertion checking, in property-based testing (PBT), and in formal machine-checked proof—in complementary and mutually supporting ways. And, more generally, we hope to prompt deeper integration of testing and proof research.

At a high level, we advocate an approach to incrementally increasing assurance: improving both code and specifications with feedback from testing and property-based testing, which check that the code and specification are consistent in concrete executions, and only then (if more assurance is desired) attempting proof. This is a simple and obvious idea that should have become routine practice long ago, but somehow has not. Related work, including a long history of lightweight formal methods, has advocated aspects of it and put some of the pieces in place, but it has not been part of main-line thinking, and, especially for production systems code with complex ownership patterns, it has not previously been practical – there has not been tooling to support it. It could be done either code-first, incrementally adding *partial* specifications for existing code, as we do here; or specification-first, as in some classic formal-methods approaches; or co-developing both code and specification. It could be done either where both code and specification are human-written, as we do here; or, we conjecture, to provide strong feedback loops for AI tooling.

For code-first development, we envisage an incremental workflow with three intertwined steps. The first is incrementally developing comprehensive specifications and internal invariants by starting with partial specifications and testing whether they hold, in conventional execution of

automatically instrumented code. This lightweight process gives quick feedback to the specification developers (which in our case were our research team, but could be the same as the code development team). The second step is refining these specifications with property-based testing. Runtime assertion checking is limited by the code coverage of the tests one has, but PBT can generate many tests that better exercise the code and specification. (Note that it is not desirable to begin with property-based testing: if pre-conditions and invariants are too incomplete, then PBT will likely generate too many false positives.) The final step, once an experimentally well-validated specification is in place and if higher assurance is desired, is trying to formally prove that the program meets it. This might well require further specification changes (to find sufficiently strong inductive invariants), and testing and PBT remain valuable, to automatically test revised specifications, and to test lemma statements, before and during the costly step of full proof.

We demonstrate the viability of this approach with a real-world case study—in several ways, a maximally challenging one, albeit of modest scale: the *hyp memory allocator* component of the pKVM hypervisor developed by Google for Android. This is production systems code, written in C by a conventional development team, running bare-metal at a hypervisor privilege level, with nontrivial ownership patterns, with rich specifications including ownership, that was not written with formal specification or verification in mind. Reynolds' remark was surely true when he made it, but we believe that it is now both necessary and possible to falsify it.

Crucially, we are meeting conventional software development where it is. The choice of C, the development of tools that can work with it (for specification, testing, test generation, and proof), the emphasis on partial specification, and the iterative process of specification development, are all there to provide a smooth path to increased assurance for conventional code and developers. Of course, there is still some way to go, from our academic tooling (that can be used on production code by us), to production-quality tooling that could be used by non-expert developers, but we aim to provide a substantial step in the right direction.

To make this possible, we developed both improvements to existing tools and new tools. Our starting point was the CN separation-logic specification language [64] and the Fulminate runtime testing system [4]. Fulminate translates C with CN specifications into C with runtime checks, using reified ghost state to check ownership properties. Then:

- We extended Fulminate to support partial specifications, added runtime checking of loop invariants and of CN lemmas, and more corner-case C features, substantially improved performance, and enabled specification testing for the hyp allocator *in situ* – with the allocator and the Fulminate-generated specification instrumentation compiled and run as part of the pKVM hypervisor, booting and running at EL2 below the Android kernel in QEMU.
- We added support for ghost arguments to the CN specification language, Fulminate runtime testing, and CN proof.
- We improved CN proof performance.
- We developed a new PBT tool, Darcy, that uses SMT solvers to generate test inputs for the complex specifications that arise for such code. Darcy supports a much wider range of preconditions than prior work. We evaluate Darcy's bug-finding ability to demonstrate how it can be used to assist specification writing.
- We developed new *tree-carver* tooling to suppress unused parts of the large Linux header files recursively included by the code.
- We demonstrate the workflow of testing-first development of specifications, initially for a stand-alone version of the pKVM hyp allocator. This step quickly found many specification bugs, and four bugs in the production hypervisor implementation. (These bugs can also be

independently useful, as a realistic benchmark suite for evaluating future separation-logic-based testing tools.)
- We use these specifications to carry out a CN correctness proof of the allocator implementation. Our experience was that the above testing, and re-testing during proof development, substantially eased the proof. Because well-tested specifications do not contain many trivial specification bugs, we could focus on the essential difficulties of the proof, such as proving inductive lemmas—which was eased by lemma testing.

We do all this for systems code written in C, which remains very widely used for critical code, but the underlying ideas should apply equally to Rust or other languages. Indeed, there is a recent proposal to add Fulminate-style contracts to Rust [41].

We start with an overview of the hyp allocator example and the application of Fulminate, Darcy, and CN proof to it in a standalone version (§2). We describe what was necessary to make specification testing work *in situ* in the pKVM hypervisor (§3), and summarise our overall experience (§4). We then describe the various technical advances that have made this possible: improvements to Fulminate (§5), ghost arguments (§6), the Darcy PBT tool (§7), and improvements to CN proof (§8). Finally we discuss some of the extensive related work (§9), and conclude (§10).

We demonstrate these capabilities on existing, production systems code, essentially unchanged. However, a number of limitations remain for future work. The CN specification language and associated tools do not yet support concurrency (the hyp allocator datastructures are protected by simple coarse-grain locking, so this is not a big limitation for that code). There is a long tail of C features, including unions and exotic GCC/Clang extensions, that are not needed for this code and which we do not support. There is a mismatch in the handling of null pointers between CN proof (which follows the ISO C standard in distinguishing null pointers and zero pointer values) on one hand and Fulminate and our PBT tools (which follow common usage in identifying the two) on the other—interestingly, this discrepancy was discovered by PBT. The hyp allocator relies on an underlying memcache layer, mapping physical memory to virtual, which we do not address. CN does not yet support Rocq export of lemmas that involve ownership, and CN proof, while usable, remains slower than one would like. Overall, one would not claim that this is production-ready tooling; our research goal is rather to explore and demonstrate the viability of the approach.

The CN-annotated pKVM hyp allocator is available online in both stand-alone and in-situ forms, along with our tools, all open-source, as in the Data Availability Statement at the end of the paper. Online supplementary material gives additional details of Fulminate and Darcy performance evaluation, Fulminate support for loop invariants, and LLDB tooling integration.

2 Overview

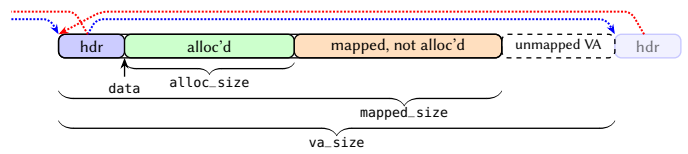
We begin with an overview of our approach using an example taken from our real-world hypervisor case study, demonstrating how separation-logic runtime testing (Fulminate) and PBT (Darcy) help identify specification bugs in a realistic setting, and highlighting some key ideas.

2.1 Code: The pKVM hyp allocator

The pKVM hypervisor is developed by Google to enforce memory isolation between the Android Linux kernel and guest virtual machines that handle sensitive user data, protecting the latter from kernel compromises, and vice versa. It is deployed in Android and upstreamed to Linux. It runs at the high-privilege Arm Exception Level 2 (EL2), where much conventional runtime infrastructure is not available, including the C standard library allocator and I/O. We focus on pKVM's *hyp allocator*, a memory allocator used to manage memory chunks of various sizes. The hyp allocator is used both by pKVM and by third party vendors, so its clear specification and correctness are crucial.

```
static struct hyp_allocator {
  struct list_head chunks;
  unsigned long start;
  u32 size;
  hyp_spinlock_t lock;
} hyp_allocator;

struct chunk_hdr {
  u32 alloc_size;
  u32 mapped_size;
  struct list_head node;
  u32 hash;
  char data __aligned(8);
};
```




```
struct list_head {
    struct list_head *next, *prev;
};
```

Fig. 1. Top level and chunk list datastructures of the allocator

The allocator manages a large, fixed, contiguous region of virtual address space, and its (partial) mapping to physical memory, by partitioning it into smaller chunks on demand. These are represented with an intrusive doubly linked circular list of chunk headers of type `struct chunk_hdr`, called a *chunk list*, shown in Fig. 1. Unlike many allocators, the hyp allocator needs to manage which virtual address ranges are mapped to physical memory and which are not. To this end, each chunk has two fields: `alloc_size` and `mapped_size`, representing the size of the memory in the chunk that is allocated to users (0 when freed) and the size of the physically mapped memory.

In this paper, we focus on two main API calls of the allocator: `hyp_alloc` and `hyp_free`. The former takes as input the size of the requested memory, tries to allocate a chunk of memory from the managed region, and returns a pointer to the allocated memory if successful or `NULL` if it fails. The latter takes as input a pointer to the memory chunk to be freed and marks it to be reused for future allocations. The hyp allocator relies on several internal helper functions, such as searching for a free chunk by iterating over the chunk list (`get_free_chunk`), splitting and recycling a portion of a large chunk (`chunk_recycle`), merging adjacent free chunks (`chunk_merge`), validating chunk headers by computing their hash (`chunk_hash_validate`), and so on. Furthermore, the allocator inherently involves bit-level manipulations to handle various alignments, as well as complicated pointer arithmetic that may lead to arithmetic overflows. The standalone version of the allocator code targeted in this section's specification, testing, and proof is 913 lines of C code, including the required dependencies but excluding comments, blank lines, and specification annotations.

2.2 Spec: Specifying `chunk_install` using CN

We use the internal function `chunk_install`, invoked by `hyp_alloc`, as a running example—see Figs. 2 and 3. (For brevity, we omit the full implementation and specification of `chunk_install`; they are included in the supplementary material.) The `chunk_install` function takes a pointer to the new chunk, the size of the memory to be allocated, a pointer to the preceding chunk, and a pointer to the allocator's auxiliary metadata; it carves out the new chunk from the previously unused memory region of the preceding chunk (`prev`). Given these inputs, the function inserts a new chunk into the chunk list while preserving the global invariant of the allocator: it updates the metadata of both chunks and invokes `chunk_list_insert` to perform the pointer operations of the underlying doubly linked list, ensuring that the new chunk `chunk` is correctly positioned right after the previous one `prev` in the allocator's chunk list.

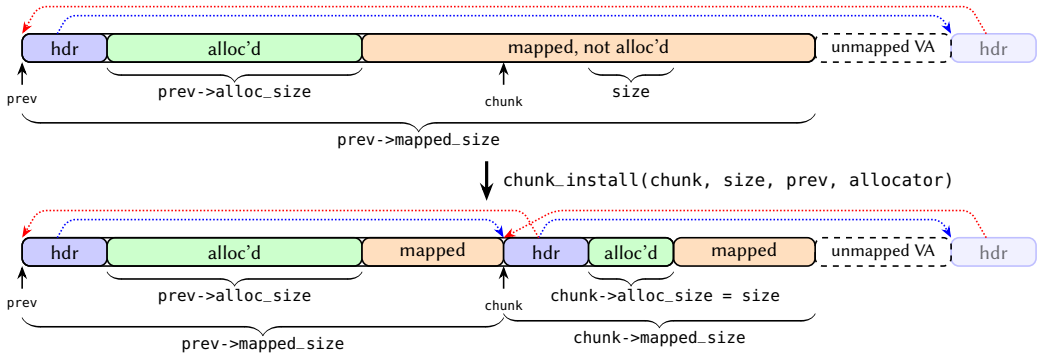


Fig. 2. `chunk_install` implementation behaviour

```
static int chunk_install(struct chunk_hdr *chunk, size_t size,
     struct chunk_hdr *prev, struct hyp_allocator *allocator)
/*@
requires
  take ha = RW<struct hyp_allocator>(allocator);
  take P = chunk_hdr(prev, ha);
  (u64)chunk + hdr_size() + size
    <= (u64)prev + (u64)P.mapped_size;
ensures
  ... omitted...
@*/
{
  size_t prev_mapped_size = prev->mapped_size;
  prev->mapped_size =
    (unsigned long)chunk - (unsigned long)prev;
  chunk->mapped_size =
    prev_mapped_size - prev->mapped_size;
  chunk->alloc_size = size;

  chunk_list_insert(chunk, prev, allocator);
  return 0;
}
```

Fig. 3. Implementation and initial specification of `chunk_install`

The correct behavior of `chunk_install` is described by several preconditions, invariants, and postconditions—e.g., the given new chunk must lie within the memory range of the previous chunk. To formally specify, test, and verify these requirements, we use the *CN specification language* [4, 64], which can capture and reason about the complex memory ownership structures of C programs through *separation-logic ownership predicates (points-to tokens)*, *separating conjunctions*, *iterated separating conjunctions*, and *recursively defined predicates*. An initial candidate precondition for `chunk_install` in CN is shown in the `requires` clause of the magic comment (delimited with `/*@` and `@*/`) on the left in Fig. 3. The specification statement

```
take ha = RW<struct hyp_allocator>(allocator)
```

asserts ownership of a memory region located at `allocator`, of the size of `struct hyp_allocator`. This corresponds to a points-to token with a quantifier of the form $\exists$`ha`. `allocator` $\mapsto$ `ha` in standard separation logic notation. We can also define predicates describing specific ownership disciplines such as `chunk_hdr(prev, ha)`. Semicolons `;` denote *separating conjunctions* (with appropriate scoping of quantified variables), expressing the disjoint ownership of parts of memory: a statement of the form `take A =` $P_1(\ldots)$ `; take B =` $P_2(A, \ldots)$ ensures that the memory regions owned by `take A =` $P_1(\ldots)$ and `take B =` $P_2(A, \ldots)$ do not overlap. Finally, we can impose constraints on values, as seen in the last two lines of the precondition, ensuring that the requested chunk ends within the physically mapped region of the previous chunk (the beige box in the top of Fig. 2, before the function call).

2.3 Spec-Test-Debug: Fulminate runtime specification testing of `chunk_install`

Once we have an initial specification for `chunk_install`, we want to quickly runtime-check these ownership and functional properties to check that our guessed specification properties are consistent with the implementation. If we instrument this program with Fulminate and run the result with a simple handwritten driver that exercises `hyp_alloc`, we get the error message:

```
************************ Failed at ************************
function __list_add, file main.pp.exec.c, line 1322
Store failed.   ==> 0x1308081c0[0] (0x1308081c0) not owned
```

Fulminate reports an ownership error—specifically, that writing to the address `0x1308081c0` has failed inside the underlying Linux `__list_add` function (operating on the structs in Fig. 1) because the address is not owned. This indicates that there is a missing ownership specification for `0x1308081c0` in `chunk_install`. We know that this error must be due to `chunk_install`'s ownership specification being incomplete, rather than a missing ownership specification somewhere else in the call stack, because Fulminate now supports *fragmentary* specifications: completely unannotated functions are now treated by Fulminate effectively as if they were inlined, so they do not require explicit ownership specifications to pass Fulminate's checks while nested within or themselves calling annotated functions (see §5.1 for details). This is what allows us to start by specifying a function like `chunk_install`, which, at this early stage of specification-writing, is both a caller of one unannotated function (`chunk_list_insert`) and a callee of another (`hyp_alloc`), without having to "fill in the gaps" for ownership specifications in other functions straight away.

The Fulminate-inserted runtime instrumentation is designed so that running LLDB on the instrumented code gives meaningful output. Here, what we deduced from Fulminate's runtime ownership error is corroborated by running LLDB and looking at the backtrace:

```
frame #2: ... `cn_failure(failure_mode=CN_FAILURE_CHECK_OWNERSHIP, spec_mode=C_ACCESS) at utils.c:86:3
frame #3: ... `c_ownership_check(access_kind="Store", generic_c_ptr=..., expected_stack_depth=1) at utils.c:602:5
frame #4: ... `__list_add(new=0x130808158, prev=0x130808008, next=0x1308081b8) at main.pp.exec.c:1322:2
frame #5: ... `chunk_list_insert(chunk=..., prev=..., allocator=...) at main.pp.exec.c:3968:3
frame #6: ... `chunk_install(chunk=..., size=..., prev=..., allocator=...) at main.pp.exec.c:4792:3
frame #7: ... `hyp_alloc(size=...) at main.pp.exec.c:7640:17
```

This tells us which sequence of function calls led to this error, including the concrete values that triggered it. We omit most of these for brevity, but note that in frame #4, the concrete `next` argument to `__list_add` (`0x1308081b8`) looks very close to the bug-triggering address `0x1308081c0`, and is indeed 8 bytes smaller than the latter.

In fact, debugging with LLDB is not just limited to inspecting C states, and we can even debug the CN specifications within the same LLDB session. Since Fulminate translates CN specifications to C code, LLDB can already inspect the underlying Fulminate specification representations out of the box, without any special support. To improve that debugging experience, we developed LLDB extensions that pretty-print boxed CN values and records. One could further refine the toolchain tailored to the specific target by extending LLDB itself, thus avoiding the problem of having to learn a new different toolchain. For more details of our LLDB extensions, see §A.3. For the hyp allocator, for example, we can inspect the CN specification corresponding to frame #6 with `cn_print`, a general-purpose printer for CN values:

```
(lldb) frame select 6
(lldb) cn_print P_cn
{ Hdr = { ... }, Node = { next = 0x00000001308081b8, prev = 0x0000000100040068 } }
```

From this, we see that the address Fulminate claims it does not own is tracked in the CN specification and accessible via `P.Node.next`, though we have not yet specified ownership of it.

Next, we can set LLDB breakpoints in `chunk_list_insert` and `__list_add` and step through the Fulminate-instrumented functions, shown below. For presentation purposes, we show the uninstrumented version of `chunk_list_insert` as the error is thrown in its callee, `__list_add`. The latter has had all of its loads and stores transformed to calls to Fulminate's `CN_LOAD` and `CN_STORE` macros, which perform the necessary ownership check and then load from or store to the provided address.

```
void chunk_list_insert(struct chunk_hdr *chunk,
                       struct chunk_hdr *prev,
                       struct hyp_allocator *allocator)
{  ...
   struct list_head *new = &chunk->node;
   struct list_head *head = &prev->node;
→  __list_add(new, head, head->next);
   ...
}
```

```
void __list_add(struct list_head *new,
                struct list_head *prev,
                struct list_head *next)
{  /* Fulminate transforms
      next->prev = new;
      into the following */
→  CN_STORE(CN_LOAD(next)->prev, CN_LOAD(new));
   ...
}
```

Since LLDB exits after the ownership check for the store to `next->prev`, this must be the source of the original runtime error. This matches what we found from the LLDB backtrace, since the pointer to the `prev` member of some `struct list_head` will be 8 bytes greater than the pointer to the struct itself, from the definition of `struct list_head`. Morally, we can see from Fig. 2 and the provided definitions that `chunk_list_insert` updates the `prev` pointer of the next chunk of `prev` via `__list_add`, and thus ownership of that next chunk is also required; we add another ownership specification to the precondition of `chunk_install` to fix this specification bug:

```
take Next = RW<struct chunk_hdr>(P.Node.next);
```

2.4 Test generation-Test-Debug: Darcy property-based testing of `chunk_install`

Even after fixing the ownership declaration issue that Fulminate detected, the specification is still incorrect, failing to ensure that the new chunk is located after the allocated region of `prev`. Fulminate only exercises the execution paths of a handwritten test driver, so, to find this bug, we would have to manually write a driver that exercises that path, which the current driver does not.

We can do better using property-based testing. Unfortunately, the existing PBT tool for CN specifications, Bennet [1], does not work well with complex constraints on pointers, failing to generate any inputs for `chunk_install`. Instead, we developed Darcy, a solver-based PBT tool for CN specifications. Darcy gathers multiple constraints, gives them to an SMT solver, and builds a test case from the model found.

When we apply Darcy to the revised `chunk_install`, it immediately finds another error and puts us in a debugger before a call to the function with the failing input.

```
frame #0: tests.out`cn_failure(failure_mode=CN_FAILURE_CHECK_OWNERSHIP, spec_mode=C_ACCESS) at utils.c:86:3
frame #1: tests.out`c_ownership_check(access_kind="Store",generic_c_ptr=4,expected_stack_depth=1) at utils.c:604:5
frame #2: tests.out`chunk_install(chunk=0x0, size=0, prev=0x12ab00738, allocator=...) at alloc.exec.c:5509:9
```

Looking at the address of `chunk` in Frame #2, we see that it is located at `0x0`, which is outside the allocator's memory region, even though `chunk` is expected to be in the memory region of `prev`, as explained in §2.2. This happens because the lower bound of the address of `chunk` is unconstrained in the precondition of `chunk_install`, in contrast to its upper bound (the last precondition in Fig. 3). This can be fixed by adding one more constraint to `chunk_install`'s precondition: `(u64)prev + hdr_size() + (u64)P.alloc_size <= (u64)chunk`.

In §4.5, we look at Darcy's ability to find 12 specification bugs that Fulminate with our manual test driver found and 12 specification bugs that it did not.

```
1  type_synonym cn_alc = {
2    pointer head,
3    pointer first, pointer last,
4    va start, u32 size
5  }
6  type_synonym cn_hdr = {
7    va header_address, u32 alloc_size,
8    u32 mapped_size, u32 va_size
9  }
10 datatype cn_hdrs {
11   Chunk_nil{},
12   Chunk_cons { cn_hdr hd, datatype cn_hdrs tl }
13 }
```

```
14 predicate [rec] (datatype cn_hdrs) Cn_hdrs
15      (pointer p, pointer prev, cn_alc ha, ...) {
16   if (ptr_eq(p, ha.head)) {
17     return Chunk_nil{};
18   } else {
19     let haddr = array_shift<byte>(p, -(offsetof(chunk_hdr, node)));
20     take raw_hdr = RW<struct chunk_hdr>(haddr);
21     assert(ptr_eq(raw_hdr.node.prev, prev));
22     take tl = Cn_hdrs(raw_hdr.node.next, p, ha, ...);
23     let cn_hdr = {header_address: (va) haddr,
24                   alloc_size: raw_hdr.alloc_size, ...};
25     ...
26     return Chunk_cons{hd : cn_hdr, tl : tl};
27   }
28 }
```

Fig. 4. Ghost state in CN

2.5 Ghost state and global invariants

Testing and debugging specifications solidify an understanding of the source code, allowing us to carve out the abstraction and invariant of the concrete state, which we call the *ghost state* of the `hyp_allocator`. The type synonyms `cn_alc` and `cn_hdr` and data type `cn_hdrs` on the left of Fig. 4 define the ghost states of the `hyp_allocator` as a whole, a chunk header, and a list of chunks, respectively. Compared with the C structs in Fig. 1, these ghost states abstract away unnecessary implementation details such as `hash`, and add fields that are useful for verification but not explicitly represented in the C structs, such as `va_size`, which expresses the size of the virtual memory region owned by each chunk, as illustrated in Fig. 1.

We represent the doubly linked list of chunks using a standard cons-list data type, named `cn_hdrs`, which can be constructed by the predicate `Cn_hdrs` shown on the right of Fig. 4. The predicate `Cn_hdrs` is a recursively defined predicate that walks through the chunk list by following the `next` pointer until it reaches `ha.head`, the head node of the chunk list stored in the `hyp_allocator` `ha`. During this traversal, the predicate not only constructs the ghost state but also enforces several global invariants of the `hyp_allocator` and its chunk list. For example, it asserts that the previous pointer of the current node matches the one observed previously (line 21). More interestingly, we still keep track of the actual pointer values of the chunks in `cn_hdr`. They are all important when reasoning about chunk splitting (recall the `chunk_install` operation in Fig. 2) or chunk merging, which, in effect, involves "raw pointer manipulation" beyond the standard ownership discipline.

2.6 Prove: CN proof of `chunk_install`

After several iterations of a specify-test-debug cycle using Fulminate and Darcy, one might get a "correct" specification in the sense that those tools no longer report any violations, and one might have found many specification and implementation bugs. If one wants to go beyond that to the high assurance of full proof, one can formally prove that the specification is correct for all possible cases, using CN proof. This sometimes requires additional CN proof annotations and other manual effort, such as proving lemmas that CN cannot automatically prove.

```
void LemmaCreateNewChunk(char *base, size_t size1,
                         size_t size, size_t size2) /*@
requires
  let size_all = size1 + size + size2 + hdr_size();
  take C_pre = barray(base, size_all);
ensures
  let chunk_hdr = array_shift<byte>(base, size1);
  let chunk = array_shift<byte>(base, size1 + hdr_size());
  let rem = array_shift<byte>(base, size1 + size + hdr_size());
  take C1 = barray(base, size1);
  take chunk_hdr = W<struct chunk_hdr>(chunk_hdr);
  take C2 = barray(chunk, size);
  take C3 = barray(rem, size2);
@*/{ /* proof omitted for brevity */ }
```

For example, the memory region owned by a chunk is expressed using an ownership predicate defined with a quantifier over addresses, an *iterated separating conjunction*, of the form `each(`*u64 i; i < owned_size*`){RW<byte>(`*prev + hdr_size() + alloc_size + i*`)}`. After the function call, however, the owned region is divided into four parts: (i) the shrunk allocated region of `prev`, (ii) the header of the new chunk, (iii) the newly allocated region of `chunk`, and (iv) the remaining owned region. CN intentionally does not automate proof about the splitting of iterated separating conjunctions, in order to preserve the predictability of its proof automation. Therefore, to complete the proof, we must manually prove a lemma (on the right) by induction. (Here, `barray(base, size)` is a shorthand for `each(`*u64 i; i < size*`){RW<byte>(`*base + i*`)}`.) Fulminate and Darcy also help us invent, debug, and prove such lemmas: CN lemmas are expressed as C functions with pre- and post-conditions but no body, and they can be tested like normal functions.

3 Runtime specification testing *in situ* at EL2

We developed our hyp allocator specification on a stand-alone version, manually extracted from the pKVM sources, before the in-situ machinery we now describe was in place: we took the original `alloc.c` and `alloc.h` and replaced their includes with hand-crafted minimal header files to make it build and execute in isolation as user code, exercised by a simple test harness or by PBT. For an academic experiment that makes perfect sense, but for real-world use it would be quite impractical—one has instead to deal with the code *in situ*: in a version of the full source tree, with the production build and runtime environment. To demonstrate that the approach scales to that (and to discover what is required to do so), we copied our stand-alone hyp allocator specifications back into a fork of the Android Linux tree, used Fulminate to instrument the hypervisor binary, and exercised them in *in situ* execution, of the hypervisor and Android Linux kernel, in QEMU (as used by the pKVM developers).

The hypervisor presents an execution environment which is radically different from user space, and its compilation environment is also rather specific. A good measure of the difference in scale between the stand-alone (user-level) and *in situ* (in the hypervisor) versions of the hyp allocator is the difference in lines of code after pre-processing. The target `alloc.c` source file contains 849 lines of original (non-specification) code. After fully pre-processing in the stand-alone environment, it expands to 1083 lines (again without the specification), a 1.28× expansion. After pre-processing the same file in the Linux build environment, the result contains over 130 000 lines, an expansion above 150×. This is the result of including the supporting headers and expanding the relevant macros, and approximately half of this is (transitively) referenced by the allocator code. While Linux in particular makes heavy use of complex macros and certain non-standard C features, we measured expansions of about 20× in other C projects. Thus, real-world scenarios are of a completely different scale to stand-alone situations. Beyond the basic fact that our goal is producing tooling which seamlessly runs in realistic environments, this serves as a stress test for the entire CN pipeline.

3.1 Instrumenting the source

The Linux kernel has its own configuration (*Kconfig*) and build (*Kbuild*) system, supporting the kernel's high degree of configurability, and driven by around 5000 configuration and build files. Instead of the C standard library, the kernel has its own support library, and the hypervisor code selectively uses parts of this.

The first step is to configure and build Linux, generating parts of the build system and some of the headers. We start with the `compile_commands.json` generated by the build system, and script the extraction of the precise build command, a 2KB string giving the particular combination of pre-processor and compiler directives, from it. We use this to pre-process the allocator.

We develop and use a Clang-based *tree-carver* tool to drastically reduce the amount of code processed by all stages of the CN pipeline. Given root functions in a file, it carves out their transitive dependency, and comments out all functions, types, and struct/union fields not depended upon by the root functions. It leaves pre-existing comments, including CN specifications, untouched. This reduces the code size by roughly a factor of 2, and reduces the number of non-standard extensions which we needed to support (especially useful at the outset).

Running Fulminate on the result forced us to improve its front-end support as described in §5.3, and to fix a number of smaller bugs in Fulminate. As our tool chain still lacks support for a few relevant non-ISO-standard features (attribute annotations, `__auto_type`, and compiler built-ins for directing compile-time evaluation), we use the pre-processor to erase their uses just before instrumenting the file. At present the resulting instrumented file needs slight manual modifications. We do not support assignment of compound literals, which occurs exactly once, and remove its instrumentation. The instrumentation slightly interferes with the compile-time dead-code removal that Linux relies on, so we direct Fulminate to not instrument one function, and we have to delete two further lines from the output. And finally, we manually re-add attributes on the three per-cpu globals.

3.2 Runtime

The runtime environment of the hypervisor is exotic in many ways: it directly manages its own page tables, mapping and unmapping physical pages into its virtual memory; deals with multiple address spaces (its own, the host's, and the physical addresses); has a limited stack size; and has very limited I/O capabilities. On top of this, it has no general-purpose `malloc` implementation which can acquire additional memory on its own. The more limited allocator that it does contain, is exactly the target of our instrumentation, so we cannot use it.

The initial Fulminate design had this bare-metal environment in mind, and the Fulminate runtime proved fundamentally compatible with it, requiring no fundamental changes. We replaced `malloc` with a simple lower-level allocator, and we disabled all of the features which rely on the standard library (or even an operating system), such as printing. The work on the efficiency of the ghost state (§5.4) proved critical: without it, the tests ran for hours.

To fully integrate Fulminate into pKVM, we had to extend pKVM with a few lines. We add calls to initialise the Fulminate runtime early in the hypervisor startup sequence, just before the allocator initialisation. We also add calls into the Fulminate runtime registering the ownership of 6 global variables, which Fulminate cannot do on its own without a single well-defined entry point (the hypervisor provides an exception-handler interface; it does not have a `main`). The 3 per-cpu globals are given wildcard ownership (§5.1) because of complex indirection machinery they use; and a further one is given wildcard ownership because it is hidden from the allocator code (and specification) behind macros. In addition, we register ownership of the host-provided memcache structure at the beginning of the appropriate hypercall, and we register the ownership of pages unmapped from the host and remapped at EL2 in the corresponding internal function.

Lastly, at startup, we take ownership of the allocator's virtual address range. The range is known ahead of time, but physical pages are only mapped into it on demand, making most of the range not backed for most of the time. This range is central to the specification of the allocator, which describes its ownership. Since we specify the ownership, but there is often no memory underlying this range, this allocator construct sits at the very edge of what our logic can express. It fits a slightly different notion of purely *abstract* ownership, and requires the ability to operate on the abstract ghost state alone; Fulminate's new optimisations over iterated separating conjunctions make this possible (§5.4).

```
void *hyp_alloc(unsigned long size) /*@
requires
  take HA_pre = hyp_allocator(&hyp_allocator);
ensures
  take HA_post = hyp_allocator(&hyp_allocator);
  take Allocated = Maybe_barray(return, size);

  let actual_size = ALIGN(size == 0u64 ?
              MIN_ALLOC() : size, MIN_ALLOC());
  take Extra = Maybe_barray_with_offset
            (return, actual_size - size, size);
@*/
```

```
void hyp_free(void *addr) /*@
requires cn_ghost u64 size;
  take HA_pre = ValidAllocatorAndAddr(&hyp_allocator, addr);
  take U = barray(addr, (u64)size);
  take V = barray_with_offset
            (addr,
            (u64)HA_pre.lseg.chunk.alloc_size - size,
            size);
  size <= (u64)HA_pre.lseg.chunk.alloc_size;
ensures
  take HA_post = hyp_allocator(&hyp_allocator);
@*/
```

Fig. 5. Specifications of `hyp_alloc` and `hyp_free`

4 Experience

We summarise our experience applying the Code-Specify-Test-Debug-Prove workflow.

4.1 Specification and invariant of the hyp allocator

We specified the two main API calls of the hyp allocator: `hyp_alloc` and `hyp_free`, as shown in Fig. 5. These use the fundamental predicate, `hyp_allocator`, which captures the ownership structure of the allocator (recall Fig. 1) and its global invariants—that all the allocated chunks are physically mapped, that the chunk list forms a well-formed doubly linked list, that the chunks in the list are sorted in ascending order of their virtual addresses, etc.

Using that predicate, the specification of `hyp_alloc` states that it returns a pointer to (and ownership of) allocated memory of the requested size, while preserving the invariant of the hyp allocator. Since `hyp_alloc` rounds the size up to meet an alignment requirement and the minimum chunk size, it may allocate a slightly larger chunk than requested. To handle this, it also yields `Extra`, the ownership of the extra memory.

Conversely, the specification of `hyp_free` takes as input a pointer to the memory chunk to be freed, along with its ownership, and releases it while preserving the ownership and invariant of the hyp allocator. However, `hyp_free` does not have direct access to the size originally requested by the client. Thus, we introduce a ghost argument (§6) to the specification that carries the size information from `hyp_alloc` to `hyp_free`.

4.2 Wildcard ownership

During runtime testing (including at EL2), wildcard ownership (§5.1) allows us to bypass unnecessary ownership checks, and focus on the parts of the allocator we verify. As noted in §1, we do not consider the physical-memory-mapping functionality `memcache`, which is orthogonal to the core logic of the hyp allocator. Nevertheless, Fulminate previously required us to declare ownership of every memory chunk touched by these functions, and would otherwise get stuck on such ownership checks. In particular, the addresses used by `memcache` are determined dynamically at runtime, making it difficult to statically suppress ownership checks for these functions. Wildcard ownership allows us to disable these checks cleanly and proceed with testing.

4.3 Applying Fulminate to the hyp allocator

For our stand-alone version of the hyp allocator, manually extracted from the pKVM source tree, we implemented a mock for `memcache` and other dependencies, to run the allocator at the user level, with small test cases. One test case we used during our specification development is as follows:

```
void test(void) {
        int *p = hyp_alloc(400); assert(p);
        int *q = hyp_alloc(400); assert(q);
        hyp_free(p);
        int *r = hyp_alloc(300); // will invoke `chunk_recycle'
        assert(p == r);
        hyp_alloc(80);
}
```

Fulminate by itself uncovered more than 20 specification bugs (counting only those we recorded in commit messages; the actual number was higher). At first glance, this may be a surprisingly high number, given that Fulminate is doing testing rather than general proof. The explanation is that humans often make simple mistakes when writing specifications—incorrectly declared ownership, flipped logical conditions, off-by-one errors, and so on. Fulminate serves as a lightweight sanity checker for specifications, catching these simple mistakes quickly, as shown in §2.

This would not have been possible without support for fragmentary specifications, as described in §2, which allows us to test specifications even when they are incomplete. To demonstrate Fulminate's capability and establish a baseline for future tools, we created a benchmark (available online) of 7 partial specifications of the hyp allocator in which some parts are intentionally omitted. We derived these specifications by omitting certain parts (e.g., specifications for top-level functions or leaf functions) from the complete specifications we developed for the hyp allocator. Fulminate handles all incomplete specifications in this benchmark. Specifically, if a specification bug introduced into the complete specification is detectable by Fulminate, then the same bug remains detectable after being introduced into a partial specification in the benchmark.

4.4 Real bugs found during the workflow

We have also found *four implementation bugs* in the hyp allocator so far, all of which have been confirmed and fixed by the pKVM development team after our reports. We found the first two bugs while writing partial specifications with Fulminate, and the remaining two while refining and proving the specification of `hyp_alloc` with CN proof.

4.4.1 Suboptimal chunk selection in `get_free_chunk`. We discovered that `get_free_chunk`, which is intended to find the smallest free chunk that fits the requested size, did not always select the optimal chunk. Specifically, when multiple free chunks are available for a requested size, the function fails to properly initialise the `best_available_size` variable on the first candidate. As a result, subsequent comparisons against an uninitialised value may leave a suboptimal chunk selected, leading to inefficient memory utilisation.

4.4.2 `hyp_alloc(0)` returns a non-null pointer. In the previous version of the hyp allocator, `hyp_alloc` incorrectly returned a valid pointer when called with a size of zero. If a client subsequently frees this pointer, it can lead to a use-after-free error. According to ISO C, the `free` function must work correctly when given a pointer previously returned by a memory management function, provided that it has not already been deallocated. In this sense, the behavior of the pKVM hypervisor allocator was inconsistent with the expectations set by standard C library semantics.

4.4.3 Incorrect type for a local variable in `hyp_alloc`. A local variable, named `missing_map`, in `hyp_alloc` was incorrectly declared as an `int` instead of `size_t`, even though other parts of the code handle the size as `size_t`. This discrepancy can lead to integer overflow in the middle of `hyp_alloc` and writing many zeros to the out-of-bounds memory.

4.4.4 Integer overflow in `hyp_alloc`. The previous `hyp_alloc` was also susceptible to integer overflow due to the lack of proper guards of its argument. For example, `hyp_alloc(0xfffffffffffffff9)` wrapped around to `hyp_alloc(0)` at the beginning of `hyp_alloc`, which consequently returned a valid pointer to a chunk of size 0.

4.5 Applying Darcy to the hyp allocator

After developing the specifications and proofs using Fulminate and CN proof, we identified room to apply PBT, and developed Darcy, as described in §7.

To evaluate Darcy's bug-finding effectiveness on the hyp allocator, we created a benchmark (available online) of realistic specification bugs by collecting patches to the specifications in our repository and retroactively applying them to the complete specification. Some of these patches were actually fixes for bugs discovered during attempted proofs, since we were experimenting with testing and proving in parallel. We extracted 24 bugs—12 that were found with Fulminate and 12 found through failed proof attempts—and applied Darcy to the affected function in each case. We performed these tests on a M3 MacBook Pro with 32GB RAM.

With default settings, 15 of these bugs were found by Darcy within the first 10-20 inputs. For all of these bugs, except one in `chunk_install`, they were found in 7.3 seconds, averaged across 10 trials. For the remaining bugs, many of them in `chunk_install`, we turned to coverage reports for insight. We presented a simplified version in §2, but the real `chunk_install` involves a relaxed precondition, where invalid `chunk` values are handled by runtime checks. Looking at the coverage reports, we saw that nearly all tests were hitting these checks.

After strengthening the precondition to forbid these paths, we found that Darcy failed to generate test cases that involved setting up the first chunk. Looking closer, we saw that the hyp allocator's page size is 4096, so we increased the maximum array length to 4096 and reran Darcy with the unmodified spec. This discovered 5 more bugs consistently and 1 flakily. All of these took at least 45 seconds and sometimes several minutes to detect the bug. The remaining 3 bugs were in functions that call `chunk_install`, where it is not as obvious how one would strengthen the precondition to get better performance. However, two of these were found with Fulminate.

From this experience, we built a more formal bug-finding study, running 10 trials for each bug, with a 3-minute timeout, with both Darcy and Bennet. Bennet finds 3 of the bugs, which are simpler preconditions. It fails to generate a single valid input for the rest of the functions. Also, running Darcy for another 10 trials, we see it fail once for one of the bugs that we had instantly discovered. Inspecting the results, we can see that the rest of the trials found the bug in 5 inputs or less, but the failing trial had 57 test inputs. Shi et al. [71] showed that some bugs are more likely to be found at smaller sizes. Darcy gradually increases its test case sizes based on QuickCheck's algorithm [18]. If the bug isn't caught at the small sizes, the larger inputs are less likely to trigger it.

The full results are given in §A.4.2.

4.6 Applying CN proof to the hyp allocator

After refining the specifications via testing, we verified the correctness of `hyp_alloc` and `hyp_free` using CN proof. The proof development consists of 1846 lines of specifications and 735 lines of proof annotations, for 913 lines of C code excluding comments and blank lines, totalling 2.8x the code size. Note that each proof was carried out by manipulating ghost state through C functions; therefore, no Rocq proofs were required. Due to the current limitations, as noted in the Introduction, we trust two API functions for `memcache` without proofs. We also made light changes to the original code: we added temporary variables to hold intermediate values for proofs, curly braces to delimit scopes for proof annotations, and an unused variable to the `chunk_hdr` struct to explicitly capture padding due to alignment. See the supplementary material for details.

There is a trade-off between the number of proof annotations and the efficiency of automated proof checking, which we can tune as explained in §8. An illustrative example is the `barray` predicate used to express ownership of memory region of a chunk (see §2.6). In CN, this ownership is represented using an iterated separating conjunction (§2.6), which introduces complex arithmetic constraints for the SMT solver. However, in the hyp allocator, the array behaves as a contiguous memory region for nearly all of the proof, so such fine-grained resource reasoning is often unnecessary. We thus treated `barray` as an opaque predicate in CN, which significantly improved the proof checking performance, at the cost of writing a few manual unfolding annotations.

4.7 Testing and performance of the hyp allocator *in situ*

We exercise the *in situ* Fulminate-instrumented allocator with a test written as a Linux kernel module exercising a few pKVM hypercalls: we top the allocator up with memory, create a virtual machine, and create a virtual CPU, exercising the primary usage points of the allocator. Our test of the Fulminate-instrumented code completes in 0.05 seconds, in QEMU emulating AArch64, making the checking sufficiently fast to run during whole-system execution; though (as expected) too slow to permanently enable in production builds. Finally, we confirmed that manually re-introducing some of the specification bugs aborts the execution, as expected.

This provides good evidence that the stand-alone experience can scale to this real-world use: one could develop and test CN specifications directly on production systems, even in this very challenging hypervisor setting.

5 Fulminate improvements

We now discuss the various improvements to Fulminate that enabled runtime specification testing of the hyp allocator, both stand-alone and *in situ*. Fulminate translates CN separation-logic specifications into C runtime assertions, instrumenting the source with executable preconditions that "take" ownership on entry to a function, executable postconditions that "put back" ownership at every point of return from a function, etc. CN's restrictive syntax, without arbitrary existentials and points-to, makes executing `RW` (CN's points-to operator) computationally feasible. Fulminate tracks ownership by mapping ranges of addresses to their stack depth wherever their ownership is asserted in the specification, checking and updating against a global ghost stack depth counter, and by instrumenting all C memory accesses with ownership checks.

5.1 Partial specifications with Fulminate

For flexible development of specifications, one wants to be able to develop them incrementally, and thus one wants to be able to test specifications that are *partial* in various ways: (a) specifications that capture only some functional and/or ownership properties, not full correctness, or where preconditions and the code do not imply postconditions; and/or (b) mixtures of annotated and unannotated functions in the same call stack, rather than strictly down-closed annotation; and/or (c) covering code where conventional separation logic cannot fully express the necessary properties.

Fulminate previously already supported partial specifications that capture a subset of the full functional and ownership properties of a given function (case (a)): it checks just that the stated specifications hold. Ownership specifications need to typically be written in pairs (whatever memory has ownership taken for it in a function's precondition needs to have ownership put back in the corresponding postcondition), and loop invariant ownership specifications rely on the existence of some precondition ownership specification over the same memory, but there are no other constraints on how much specification the user needs to write per-function for Fulminate to run successfully. We have added new CLI flags that allow the user to toggle which *intermediate* specifications are checked by Fulminate, including enabling/disabling loop invariant checks, loop

invariant leak checks and lemma checks (see §5.2); and allowing users to only instrument certain functions, and/or skip instrumenting others.

As mentioned in §2.3, Fulminate now supports *fragmentary* specifications (case (b)), effectively treating functions with no specifications as inlined and expecting no explicit ownership specifications to be provided for them. Unannotated functions are now only instrumented with access checks, no longer incrementing/decrementing Fulminate's ghost stack depth counter. This change makes it possible to run Fulminate on programs where the call stack is a mix of annotated and unannotated functions, which is the typical state during the specification-writing process.

Further, Fulminate now supports wildcard ownership (case (c)), where a memory region marked as "wildcard" in the ownership ghost map can be ignored by Fulminate when checking ownership—i.e. all ownership checks for such specially-marked memory pass automatically. This is particularly useful when an annotated function accesses memory that is from higher up in the call stack and needs to already be owned at the correct stack depth, but taking ownership for this memory is tricky. For example, at EL2, standard CN separation-logic does not always cover what is going on: some regions need their ownership asserted in the early stages of pKVM initialisation, before their values are even allowed to be accessed, and others (e.g. per-CPU variables) cannot be dereferenced directly, which is needed for asserting ownership in Fulminate. Wildcard ownership can also be useful in less critical scenarios: the user can mark a memory region as wildcard if they wish to leave writing its ownership specification for later.

These simple extensions turn out to make a big difference in the user experience.

5.2 Support for more CN features

Loop invariants. Fulminate now supports runtime checking of loop invariants. For a given CN-annotated loop, Fulminate translates the loop invariant into C runtime checks and injects them into the source such that they are executed just before the loop condition is evaluated at each iteration. Functional properties are checked in the usual way. Ownership is asserted by first bumping *down* the recorded stack depth in Fulminate's ghost state, both for (a) pointers enclosed in explicit calls to `RW` in the loop invariant, and (b) the addresses of function arguments and all block-local variables that have been declared so far. (CN tracks the latter ownership implicitly, regardless of whether there is any user-provided loop invariant.) Fulminate then performs a leak check, and if that succeeds, it puts back ownership by bumping up the recorded stack depth of the aforementioned addresses (all the while leaving the global ghost stack depth counter unchanged). For more details, see §A.2.

To enable and encourage partial loop-invariant-writing, Fulminate does not translate CN's implicit ownership checks unless a user-provided loop invariant exists. In addition, the loop leak check is disabled by default and a CLI flag has been added to explicitly enable it. We note that CN does not yet support framing for loop invariants, and so Fulminate's loop leak check will fail if *any* in-scope variable does not have its ownership asserted in the loop invariant—which is one reason we disable this check by default, to pose less of a barrier to partial specification writing and checking. We have also added a CLI flag for disabling runtime loop invariant checks altogether.

Lemmas. Supporting lemmas in Fulminate was a relatively straightforward extension: they consist of a `requires` and `ensures` clause like CN pre- and post-conditions, a name and an optional list of ghost arguments. Fulminate translates a lemma to a C function with an empty body, taking the Fulminate representations of the lemma's ghost arguments as its list of arguments (see §6) and with pre- and post-condition corresponding to the `requires` and `ensures` clauses, translated to C in the way Fulminate already does. For partial-specification-writing, a CLI flag has been added to toggle whether user-provided lemmas are checked at runtime (as mentioned in §5.1).

5.3 Support for more C features

Better C control-flow support. Fulminate now has better support for C control-flow statements: namely, for `goto`s into and out of blocks, and `continue`. For `goto`s, this amounted to unmapping the addresses of all in-scope variables from Fulminate's ownership ghost state just before a `goto` statement, and mapping in the addresses of all in-scope variables on entry to a label. The approach is similar for `continue` statements, where all variables that were declared between the control-flow statement being jumped back to and the `continue` statement itself are unmapped from the ghost state just before the `continue`.

GCC-statement-expressions. A GCC statement-expression is a sequence of statements enclosed in curly braces, and then parentheses, where the value of the final statement's subexpression (if the final statement is of the form of an expression followed by a semicolon) is the value of the entire statement-expression, and otherwise `void` [24]. These can include variable declarations, the addresses of which need to be mapped to and unmapped from Fulminate's ownership ghost state. In the case where the value of the statement-expression is not `void`, we store the final subexpression in a ghost variable, unmap the addresses of all the statement-expression locals, and then add the ghost variable followed by a semicolon at the end, so that the statement-expression still has the right value. In the `void` case, we just unmap the addresses at the end, as is done in blocks.

Unions. The hyp allocator contains a single instance of a union: its spinlock. For applying Fulminate to the *in situ* hyp allocator, we needed a way of supporting unions in the CN frontend (which is shared between proof, runtime checking and PBT) and in Fulminate. As CN does not yet support concurrency or unions in general, we have not yet written any specifications involving the hyp allocator's spinlock, so we work around this by representing unions as arrays of bytes, which is good enough for testing of the allocator.

Support for `goto`s was needed for both the stand-alone and *in situ* hyp allocator, while support for GCC-statement-expressions and unions was only needed for the *in situ* allocator, whose headers use a larger subset of C features than the simplified headers of the stand-alone allocator.

5.4 Performance improvements

The initial version of Fulminate [4] used naive data structures and memory management for ghost state to demonstrate the general approach, but its performance was not good enough for the hyp allocator. We replaced the existing ownership ghost-state data structures with more asymptotically efficient designs. Initially, the ownership ghost state was represented using a hash table, fully materialising the per-byte mapping from addresses to stack depths. The new representation is based on radix tries, and centres on ranges. Under big-endian key orientation, all nodes of a radix trie represent contiguous intervals, whose size shrinks to 1 towards the leaves. We exploit this by pruning the trie at the nodes whose corresponding interval is mapped to the same depth in its entirety. This gives us a data structure which semantically maps individual addresses to depths but compresses contiguous intervals. To extract range-based information, we make it parametric in a monoid m, and a function $inj : depth \to m$. The range-based queries are then a variant of the *fold* operation, returning results of the shape $inj(v(a_0)) * \ldots * inj(v(a_n))$ for an address interval $[a_0, a_n]$, monoidal operation $*$, and representing the value-at function with v. We pre-compute partial query results and store them in interior nodes, yielding a structure that supports querying ranges in a time independent of their size, and we further optimise it by using level compression, as usual for similar structures. In the C implementation, we express the parameterisation with pre-processor inclusion, and instantiate the structure to keep track of ownership-depth minima and maxima.

Combined with tracking the sets of addresses that loop invariants take ownership of (§5.2), this significantly reduces the time complexity of ghost state operations. For instance, ownership checks and updates—performed at every function entry and exit—were linear in the size of the memory region, while leak checks—performed at every function exit and every loop iteration—were linear in the size of the entire tracked memory; both kinds of operations are now done in constant time.

In addition, we implemented several optimisations for runtime iterated separating conjunctions. Fulminate now checks whether the constraints for a separating conjunction over `RW` are simply lower and upper bounds over the index variable (i.e. the region is contiguous), in which case the entire region's ownership is asserted via a single call before the relevant pointers are looped over and dereferenced. We also implemented an analysis over the results of these conjunctions, represented as maps in CN and C hash tables in Fulminate; Fulminate no longer loops over the pointers to construct a map if the result of the conjunction is never used.

We also introduced a bump allocator. The bump allocator performs allocations (bumping a pointer) and deallocations (resetting it) in constant time, and is used for managing local ghost state. Fulminate makes use of the known lifetimes of different types of specification: for function-local specification, it stores the bump allocator's current pointer p on entry to every annotated function, and resets its pointer back to p on function exit. Similarly, for loop invariants, Fulminate resets the bump counter at the end of each loop iteration and on exit from the loop (§5.2).

As a result of these changes, Fulminate is now able to scale up to much larger examples than before. For example, for a representative test size of 64 allocations, the standalone Fulminate-instrumented hyp allocator runs in only 0.47s and uses 41MB space, eminently usable in an interactive workflow. Additionally, we compared this new-and-improved Fulminate to the previous version from [4] using various CN-annotated examples, which demonstrates it uses substantially less space, both in constant factor and scaling. For more details, see the supplementary material (§A.1).

6 Ghost arguments

To ensure decidable type inference as well as executable specification, the CN specification language uses `take` bindings to restrict existentials to computable outputs of resources. This restricted form of quantification is extended by allowing the user to prove lemmas and instantiate quantifiers. A ghost argument of a function is a universal quantification that must be instantiated when the function is called. By requiring ghost arguments to be supplied by the user, we extend the domain of quantification while easily maintaining dynamic checking and decidable static checking.

For example, the C string library function `strcat` concatenates two `'\0'`-terminated strings by copying the second string into the unused space of the first string's buffer. This is safe only if the buffer has enough unused space, but the buffer size cannot be computed just from the given string pointer. With ghost arguments the specification can be parameterised on passed-in sizes:

```
char *strcat(char *dest, const char *src);
/*@ requires ghost u64 dest_size, u64 src_size; // declarations of ghost arguments
      take srcIn = String_Buf_At(src, src_size); take destIn = String_Buf_At(dest, dest_size);
      (u128) string_len(srcIn) + (u128) string_len(destIn) < (u128) dest_size; // < for '\0'
    ensures
      take srcOut = String_Buf_At(src, src_size); take destOut = String_Buf_At(dest, dest_size);
      srcIn == srcOut; destOut == string_buf_concat(destIn, srcIn);
      string_len(destOut) == string_len(srcIn) + string_len(destIn); ptr_eq(return, dest);    @*/
```

As explained in §4, ghost arguments allow us to write a more natural specification for the hyp allocator. We have implemented ghost arguments for functions in CN proof and runtime testing, though not yet for property-based testing. To simplify the implementation, the user of CN

must supply ghost arguments to functions requiring them. Because of this, their type checking is straightforward and closely follows that of computational (C) arguments.

Extending runtime checking with ghost arguments is a bit subtler. Instead of changing the API of C functions with ghost arguments in the instrumented code, we use the heap to pass around ghost arguments. A global ghost frame stack is instrumented for this. For each function specification and call site, the list of ghost argument types is represented by an enum. At each call site, an array of these ghost arguments and its enum tag is pushed onto the ghost frame stack. When the called function begins executing, the specification is compared against this tag, the ghost arguments are loaded from the array into local variables, and then the ghost frame stack is popped.

7 The Darcy property-based testing tool

To apply property-based testing to the hyp allocator, we need to be able to generate inputs that satisfy the CN preconditions of its constituent functions. The only existing PBT tool for CN, Bennet [1], uses random backtracking search, which performs well when constraints are easy to satisfy and mostly local, but wastes time on values that satisfy local constraints but not constraints encountered later in the search. It cannot handle any of the preconditions involving the allocator's chunk list, which excludes all the interesting functions in this experiment. So we built a new tool, Darcy, that uses an SMT solver to generate random test inputs satisfying CN preconditions, in the form of synthesised standalone C programs that require only Z3 [21] and can be executed by Fulminate. Darcy supports nearly all of the functions in the hyp allocator. It reuses the user interface and internal generator DSL of Bennet.

For a given CN precondition, Darcy goes through four main stages to generate a test input: (1) selecting a random path through the precondition, (2) recording all the constraints along that path, (3) asking the SMT solver for a model, and (4) building the input from the solver's model.

Selecting a Path. Darcy begins by selecting a random path through the precondition. At each branching point (if-else statements), it chooses uniformly between the branches. For example, in the `cn_hdrs` predicate in Fig. 4, it would choose randomly between the recursive case and the base case. This ensures the eventual SMT query uses only constraints along a single path. To guarantee termination when dealing with recursive predicates, we use "sized" generation, inspired by QuickCheck [18]. Darcy selects a test case size for each generation attempt, which constrains the depth of recursive calls, following prior work [1]. For example, consider the `cn_hdrs` predicate, and suppose we were generating an input of size 3. We would normally randomly choose between the branches of the `if`, but, if we took the recursive branch 4 times, we would exceed the depth limit and choose the base case.

Since the path chosen is randomly selected, it is possible to select a path that is impossible (resulting in unsatisfiable constraints). Unlike Bennet, which cannot distinguish between constraints that are unsatisfiable and ones that are just difficult to satisfy, Darcy can detect UNSAT. To avoid repeating such mistakes, Darcy supports test-time path pruning, maintaining a trie data structure that remembers which sequences of choices lead to unsatisfiable constraints.

Gathering Constraints. Once a path through the precondition has been chosen, Darcy traverses the predicate once more, generating a fresh symbol for each value that must be generated. This allows the solver to disambiguate between the `raw_hdr` fields (in `cn_hdrs`) for each element. We generate values for the function's arguments and any time an `RW` is used (for the value stored in memory). Darcy also stores each logical and resource constraint along the path. Resource constraints are intervals of the first byte (`p`) and last byte (`p + size`). So in the case of `cn_hdrs`, for the first node, we add a new resource constraint for `RW(haddr): Resource(haddr, haddr + sizeof(struct chunk_hdr))`.

Assertions are stored verbatim, so here we would remember `ptr_eq(raw_hdr.node.prev, prev)`, where `prev` is substituted with the argument to `cn_hdrs`.

To efficiently handle arrays, Darcy treats resources of the form `each(..){RW(..)}` as contiguous arrays. It then generates a variable for each potential element, up to a fixed max array length. (This optimization does make Darcy incomplete, as it cannot generate arrays with intertwined ownership—for example, claiming ownership of an array without one element, whose ownership is claimed separately, later in the precondition. Extending Darcy to allow more general but more expensive representations is left to future work.)

Querying the Solver. The constraints are then given to the SMT solver. The translation is similar to CN's translation for proof, with some more information given to the solver. For recursive functions in the CN specification language, CN treats them as uninterpreted functions. Instead, we give the entire definition to the solver to avoid spurious models (those that do not truly satisfy the precondition). For pointers, we allocate a buffer and constrain all owned memory to appear within it. This gives the solver freedom to assign pointers to any value in the range, while guaranteeing it will be mapped memory. To guarantee separation, for each pair of blocks of $(\mathtt{p_i}, \mathtt{size_i})$ and $(\mathtt{p_j}, \mathtt{size_j})$, we assert that $\mathtt{p_i} + \mathtt{size_i} <= \mathtt{p_j} \vee \mathtt{p_j} + \mathtt{size_j} <= \mathtt{p_i}$.

To control the distribution of models from the solver, we turn our query into a MAX-SMT optimization problem [23]. In addition to "hard" constraints that we expect the model to satisfy, we can also add "soft" constraints, of which the solver will satisfy as many as possible. For each value `x` that we ask for, we generate a random value $\mathtt{x_{skew}}$, and add a soft constraint that $\mathtt{x} == \mathtt{x_{skew}}$. This biases the solver's output distribution based on the distribution we sample $\mathtt{x_{skew}}$ from. The likelihood of a given model is the number of invalid assignments for which it is the nearest solution. This distribution, in turn, is configurable, defaulting to a truncated uniform distribution, `[0, size)` for unsigned bitvectors and `(-size, size)` for signed ones. (For small sizes, Darcy also generates the maximum and minimum representable value.) Darcy also supports using uniform distributions, as well as disabling skewing entirely. For complex pointer-manipulating programs, it can also shuffle the order of pointers in memory via soft assertions that one pointer is less than another.

We then ask the solver if the constraints are satisfiable. If the result is `UNSAT`, we select a different path. For our example with `cn_hdrs`, the model would look like: `{ raw_hdr1 = ..., raw_hdr2 = ..., raw_hdr3 = ..., ... }`, where `raw_hdr1` is the `struct chunk_hdr` value of the first node.

Concretising the Model. If the solver finds a model, Darcy will traverse the precondition one last time. This time, whenever a value is needed, we get the value from the SMT solver. We also evaluate memory locations, to store the values at their appropriate locations in memory. In the case of `cn_hdrs`, Darcy would request the value of `raw_hdr1` from the solver, evaluate what `haddr` should be, and store the value of `raw_hdr1` there.

Feedback. Darcy provides two kinds of user feedback. First, by virtue of using Fulminate to determine whether a function works, we get Fulminate's error messages for free. Second, Darcy provides code coverage information via `lcov`, so that users can evaluate the performance of their testing. We provide a CLI flag that causes debuggers to break before a failing input is executed, allowing the user to easily see the failure in action. We also synthesise programs that reproduce a failing input, using Bennet. Unfortunately, the current prototype only preserves basic memory layout invariants, so for any preconditions involving pointer arithmetic beyond arrays, one may need the debugger.

We compare Darcy against Bennet via two sets of experiments. First, we ran the bug-finding case studies from the Bennet paper [1]. These include BSTs, AVL trees, a ring queue, and more, where particular bugs were injected. For all of them, Darcy was slower than Bennet at finding bugs (§A.4.1).

We also did an extensive study on the time it takes to generate a single input, for Bennet and Darcy across various test case sizes, so that we can see what the tradeoff of backtracking and using an SMT solver is, at various scales (§A.4.3).

8 CN proof

The hyp allocator proof required three main changes to CN to improve proof performance. First, CN's resource inference includes automatic folding and unfolding of resource predicates, defined as if-then-else "disjunctions" of guarded cases: CN's previous resource inference automatically unfolded such definitions whenever it could prove that a predicate instance's case was uniquely determined, in a scheme inspired by [79]. However, this required CN to issue many repeated queries to the SMT solver, to determine the provability of case guards (i.e. whenever adding a new assumption to the proof context that could affect provability, it would need at least one SMT query per resource unfolding candidate). To reduce the number of SMT queries, we changed CN to only automatically unfold predicate definitions without case disjunctions, at the cost of some manual predicate unfolding by the user. Second, we extended CN so users can choose, per predicate definition, to treat it as opaque, which is useful when the unfolded definition would incur expensive resource inference. Finally, CN's SMT automation previously used incremental solving (in which the solver is incrementally updated with new constraints learnt along the analysed control-flow path). Unlike smaller examples, in the hyp allocator incremental solving performs worse, so we extended CN to support both incremental and non-incremental solving. Combined, these changes improved proof performance for the hyp allocator by a factor of 7.5.

9 Related work

There is a long line of work on lightweight formal methods, e.g., from [34, 37, 80] to [8, 9, 54].

More specifically, the idea of using a common language for testing and proof is still under-appreciated but dates back to at least the 1970s, with the Euclid system [44]. The Euclid experience (along with Z [33, 76]) went on to influence the design of Larch [25], which supported a much richer (and only partially executable) specification language based on multisorted first-order logic. Meanwhile Meyer [55] popularised the idea of design-by-contract and using custom behavioural interface specification languages to specify runtime assertions. Both of these fed into the design of systems like JML [46] and Spec# [5], combined verification and runtime testing specification languages for Java and C# respectively. More recently, the Frama-C E-ACSL [6, 73–75] plugin provides highly optimized support for runtime assertion checking in C.

All of these systems were based on various fragments of first-order logic, and they all had different *ad hoc* approaches to account for aliasing and pointers. Euclid, the earliest system, forbade aliasing of mutable data altogether, and its successors had somewhat more liberal sets of restrictions. Fulminate [4] is distinctive in that its runtime assertions are specified in a fragment of separation logic, which makes the specification of aliasing much smoother, albeit at the cost of needing more instrumentation to dynamically track ownership.

For pKVM, [54] develops an executable specification for many of the top-level hypercalls and other exception handling. This spec is embedded in C, which means it needs no special tooling or expertise, but cannot be directly used for property-based testing or formal verification—an extreme version of lightweight formal methods. Porting these specifications to CN's spec language will be a good stress test for its expressiveness, and will also enable proof and randomised testing.

Property-based testing, popularized by QuickCheck [18], is a testing approach in which structured inputs are repeatedly generated and tested. A common technique involves backtracking search, which has seen great success [16, 17, 26, 43, 51] due to its lightweight nature, particularly in combination with lazy instantiation of values [2]. Most relevant is Bennet [1], which performs

backtracking search to find inputs that satisfy CN preconditions. While this works well for recursive data structures (including complex ones like AVL trees) [71], the hyp allocator makes extensive use of arithmetic constraints, which Bennet is unable to handle. By contrast, Darcy is able to generate inputs for most of the hyp allocator, but with the overhead of an SMT solver.

Other tools have applied constraint solving to PBT before, with FocalTest [13, 14] and Target [67], which uses an SMT solver, like Darcy. Luck [42] is a language that integrates backtracking search and constraint solving approaches.

Constraint solvers are also used in testing based on symbolic execution [10, 27, 28, 63, 68], which focus on systematic path exploration, without placing much emphasis on the distribution of values (besides preventing repeated inputs). Instead, Darcy uses MaxSMT queries to skew test inputs towards random assignments. This ensures diversity of test inputs, favoring inputs that are far from other precondition-satisfying inputs (as they are "close" to more invalid assignments).

The (dynamic) symbolic execution tools Java StarFinder [61, 62] and Concolic StarFinder [60] both use SMT solvers to generate inputs that satisfy separation logic formulas. They restrict their predicates to a decidable fragment of separation logic [45] and use an SMT solver to generate inputs. Due to targeting Java, the lower level details of pointer arithmetic and arithmetic constraints over pointers do not appear. Whereas we target systems code, they focus on data structures, like Bennet.

A related approach is to test against a reference implementation: for example, Cutler et al. [20] developed a production implementation in Rust, and a model implementation with proofs in Lean, and used differential random testing to connect the implementation with the model.

A novelty of CN is that the same specifications can be used both for runtime assertions and generating property-based tests. We believe that this lends itself to the incremental development of specifications, because if a spec is too partial then PBT generates too many false positives, but once it is complete enough then PBT is very effective for discovering the corner cases and edge conditions that must be handled by a genuinely accurate specification.

Another community again focusses on runtime verification [31], often with custom temporal-logic specification languages. Meanwhile, classic formal verification has many recent high-profile successes, including BlueRock [53], CertiKOS [29, 30, 72], CompCert [7, 48], F* [59, 77], Hyper-V [47], IronFleet [32], SeKVM [49, 50, 78], and seL4 [39, 40], and separation-logic proof tools including Iris [38], VST [12, 81], VeriFast [35], Viper [56], Prusti [3], Gillian [52, 66], and FCSL [70]. Some have led to widely deployed verified code—a big achievement for semantics and verification research—but adoption in conventional systems software development projects remains a challenge.

10 Conclusion

Specification has a cost, of course: one needs to explain in machine-readable language what the code achieves, and, for fine-grain specification, how it does so. Sometimes specification can abstract substantially from the code, especially top-level specifications, but the implicit reasons why the code works, when made explicit, can often be longer than the code. The redundancy between code and specification is both good and bad: it forces one to think about the system in two complementary ways, and that, along with testing and/or proving the correspondence between them, can be invaluable in finding and excluding errors. But specs have to be written, and have to be maintained.

To justify this cost, the benefits have to be commensurate, and, for wide adoption, we believe they have to be incremental: developers have to be able to get significant benefit from initial, partial specifications, whether or not they ultimately aim for high-assurance full verification. And they need tools with shallow learning curves. The integration of specification, runtime checking of those (rich ownership-enabled) specifications, property-based testing leveraging the specifications to generate test inputs, and proof (including runtime checking also of lemma statements), is a step towards this—and hopefully a step towards more robust software infrastructure.

Data availability statement

The artifact for this paper is available in Zenodo at https://doi.org/10.5281/zenodo.18937570. For the latest source code, see the following repositories:

- CN, Fulminate, and Darcy: https://github.com/rems-project/cn
- Cut-out hyp allocator with specifications and proofs: https://github.com/rems-project/cn-pKVM-hyp-allocator
- Android Linux kernel with in situ Fulminate-instrumented allocator: https://github.com/rems-project/linux/tree/pkvm-6.12-pldi26-artefact

Acknowledgments

We thank the pKVM development team (Will Deacon, Keir Fraser, and their colleagues) for discussions and their support. We thank Ben Laurie and Sarah de Haas (Google) for their support.

This work was funded in part by Google. This work was funded in part by a Google DeepMind Scholarship (Banerjee). This work was funded in part by UK Research and Innovation (UKRI) under the UK government's Horizon Europe funding guarantee for ERC-AdG-2022, EP/Y035976/1 SAFER (Sewell). This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 789108, ERC-AdG-2017 ELVER, Sewell). This work was supported in part by a European Research Council (ERC) Consolidator Grant for the project "TypeFoundry", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 101002277, Krishnaswami). This work was supported in part by a Royal Society University Research Fellowship (URF\R1\241195, Pulte). This work was supported in part by the National Science Foundation under grant *SHF: Medium: Usable Property-Based Testing, NSF #2402449*. The authors would like to thank the Isaac Newton Institute for Mathematical Sciences, Cambridge, for support and hospitality during the programme Big Specification, where work on this paper was undertaken. This work was supported in part by EPSRC grant EP/Z000580/1.

Claude Code was used in the development of Darcy, in parts of Fulminate's bump allocator and for minor CLI improvements, and in drawing Figs 1 and 2.

A Supplementary material

A.1 Fulminate performance evaluation

In this section, we discuss Fulminate's performance when applied to the hyp allocator, to assess its usability for that case study, and also compare the performance of the Fulminate presented in this paper with that of the original Fulminate paper [4] (**new** and **old** respectively), on the benchmarks presented there, to assess the effects of our performance improvements. The original Fulminate cannot run on the hyp allocator, both for feature-coverage and (for the in situ version) performance reasons, so we cannot compare the two versions on that. Throughout this section, t represents the runtime of the executable in seconds; s represents the space usage in MB; the *instr* subscript denotes binaries instrumented with Fulminate checks; and *uninstr* denotes binaries without any instrumentation.

We ran our benchmarks on a 2024 MacBook Pro with M4 Pro chip and 48GB RAM. This is different hardware from that used in [4]. To compare between the two versions of Fulminate, we therefore re-ran the same benchmarks from the original Fulminate paper on this machine, and also disabled loop invariant and lemma checking in the new Fulminate. The time taken to run uninstrumented binaries for all the examples here was negligible on this machine (in contrast with the runtimes reported in that paper that were a minimum of 0.20s). Hence, we present absolute results for runtimes of instrumented binaries rather than ratios or percentages.

A.1.1 Usability of new Fulminate on the hyp allocator. The performance metrics for the hyp allocator are presented in Figs 6 and 7. Here, we parameterise on the number of allocations (`#` allocs) that are requested from the hyp allocator, looping and calling `hyp_alloc`; we later loop the same number of times to free the allocated pages via `hyp_free`. We also report the number of calls to Fulminate's internal function that asserts ownership (`#` own. calls); in the original Fulminate paper, these corresponded exactly to calls to `Owned` (CN's former syntactic name for `RW`) in the CN specifications; now, due to the changes to iterated conjunctions described in §5.4, this records a single ownership call rather than several wherever conjunctions are optimisable.

The time and space metrics reported in the third and fifth columns of both tables are reasonably small. For the hyp allocator with full checks, only after 128 allocations does the binary take more than a second to run, and without loop invariant and lemma checking, it takes twice as many allocations for this to happen. Fig 8 shows the linear relationship between the number of requested allocations and $\log_2$ of the runtime of the Fulminate-instrumented binary, both with full specification checks (the blue line corresponding to Fig 7) and without (the red line corresponding to Fig 6). (We exclude data points where the runtime is negligible). The gradient of both lines is similar, but the blue line has a steeper gradient and is diverging from the red; i.e. the Fulminate-with-full-checks hyp allocator consistently takes longer to run and the gap between the two runtimes increases with the number of allocations. The Fulminate-with-full-checks hyp allocator is 3 units higher than the version with fewer checks once it reaches 256 allocations, incurring a $2^3 = 8\times$ slowdown.

A.1.2 Comparing old and new Fulminate on the pKVM buddy allocator. The buddy allocator is another allocator within pKVM [22] that is a good target for verification and testing. It was previously used as the principal real-world case study for the CN [64] and Fulminate [4] papers. Fig 9 shows the performance metrics from applying the old and new versions of Fulminate to the buddy allocator, varying the number of pages. The fourth column shows the runtime improvement of the new version of Fulminate over the old version as a percentage, and the final column shows the multiplicative factor of improvement in space performance.

The main takeaway here is that while the old Fulminate runs out of memory after 8 pages, the new Fulminate is able to tackle far bigger sizes. The final column of Fig 9, although only made up

# allocs	# own. calls	t_{instr} (s)	s_{uninstr} (MB)	s_{instr} (MB)
2	3076	0.01	17.6	26.8
4	5660	0.02	17.6	26.9
8	12,758	0.02	17.6	27.1
16	33,606	0.03	17.6	27.8
32	104,026	0.06	17.6	29.5
64	357,500	0.16	17.6	35.4
128	1,317,590	0.57	17.6	56.3
256	5,048,486	2.10	17.6	135
512	19,747,072	8.22	17.8	437
1024	78,097,760	33.0	18.0	1620
2048	310,613,176	133	18.4	6090

Fig. 6. Performance metrics for the pKVM hyp allocator instrumented with Fulminate checks, with loop invariant and lemma checking disabled. Exercised using different numbers of requested allocations (# allocs) from 2^1 to 2^{11}.

# allocs	# own. calls	t_{instr} (s)	s_{uninstr} (MB)	s_{instr} (MB)
2	4756	0.01	17.6	26.8
4	8845	0.02	17.6	27.0
8	20,649	0.02	17.6	27.3
16	60,213	0.04	17.6	28.2
32	227,065	0.11	17.6	31.1
64	1081183	0.47	17.6	41.2
128	6,250,077	2.66	17.6	78.5
256	41,438,581	17.6	17.6	221
512	299,292,739	129	17.8	775

Fig. 7. Performance metrics for the pKVM hyp allocator instrumented with full Fulminate checks, including loop invariant and lemma checks. Exercised using different numbers of requested allocations (# allocs) from 2^1 to 2^9.

of 3 data points due to the limitations of old Fulminate, shows that the space performance for the buddy allocator is substantially better, both in constant factor (11× for 2 pages) and in scaling (30× for 8 pages).

A.1.3 Comparing old and new Fulminate on the tutorial examples. Fulminate was also previously evaluated against 54 examples from the CN tutorial [65], a set of examples designed to teach people how to use CN, Fulminate and Bennet, as well as a subset of the VeriFast C tutorial [36] ported to CN. These are small mathematical examples, designed for understanding CN's specification language and toolchain, and are smaller in scale and complexity than the real-world pKVM examples evaluated so far. The overall benchmarks and comparison between old and new Fulminate are in Fig 11. The mean runtime of these examples is negligible for both old and new Fulminate, and there is a negligible (< 2%, 0.02MB) decrease in space performance in new Fulminate.

The original Fulminate paper also evaluated on another iterated example, a stack example taken from Software Foundations, Vol. 6 [15], with stacks of increasing size. The comparison metrics for this are reported in Fig 12. Up to 64 stack elements, the runtime performance between old and new Fulminate is difficult to compare, as even with instrumentation it is negligible for both versions; for the 3 data points (128, 256 and 512 elements) where useful metrics for both Fulminate versions exist, we note that new Fulminate's runtime performance is either the same or marginally better than

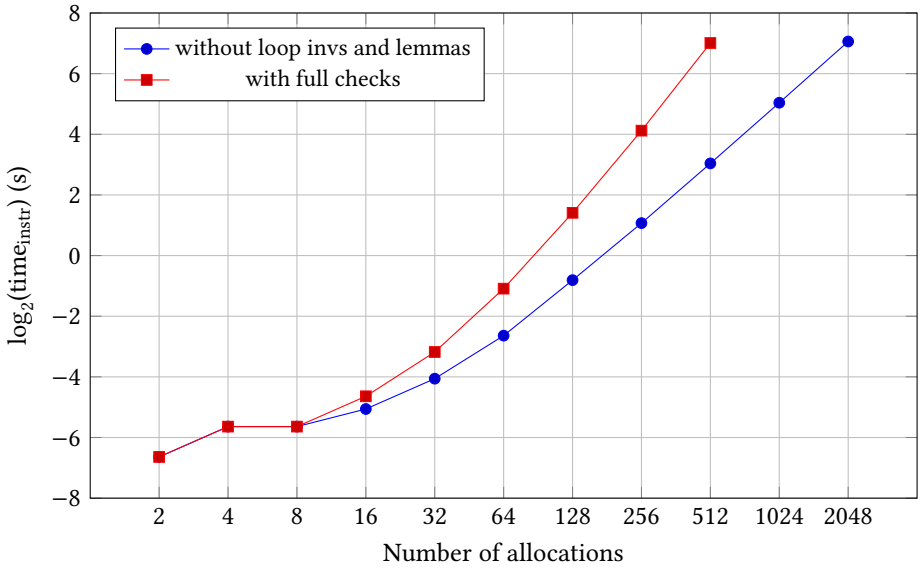


Fig. 8. Number of allocations requested from the hyp allocator plotted against $\log_2$ of the runtime of the instrumented binary, for the hyp allocator instrumented with Fulminate's full checks enabled and with loop invariant and lemma checks disabled.

# pages	**old** t_{instr} (s)	**new** t_{instr} (s)	$t_{\text{improvement}}$ (%)	**old** s_{instr} (MB)	**new** s_{instr} (MB)	$s_{\text{improvement}}$ (×)
2	0.02	0.01	50.0	56.1	5.06	11.1
4	0.07	0.05	28.6	187	10.4	17.9
8	0.34	0.21	38.2	796	26.6	29.9
16	oom	0.77	-	oom	45.8	-
32	oom	2.95	-	oom	94.6	-
64	oom	11.81	-	oom	194	-
128	oom	47.74	-	oom	392	-

Fig. 9. Performance metrics for the pKVM buddy allocator, exercised using different numbers of pages (from 2^1 to 2^7). Entries marked 'oom' signify where the old version of Fulminate ran out of memory and failed to complete the whole execution.

before. As with the buddy allocator, the last column of the table shows a substantial improvement in Fulminate's space performance. This is illustrated in Fig 13.

A.2 Details: Loop invariants in Fulminate

In this section, we provide more details on the design and implementation of loop invariants in Fulminate, as first described in §5.2.

Consider the function on the top left of Fig. 14. Its CN pre- and post-condition state the required ownership properties over *n* and a functional property over the return value. The loop invariant, indicated by the `inv` keyword, asserts ownership of *n*, some functional properties, and that the address *n* itself remains unchanged. The code on the bottom left and right is the Fulminate automatically instrumented version of the while loop, with C comments added and error-message-updating statements removed for clarity. The loop invariant checks are injected just before the instrumented

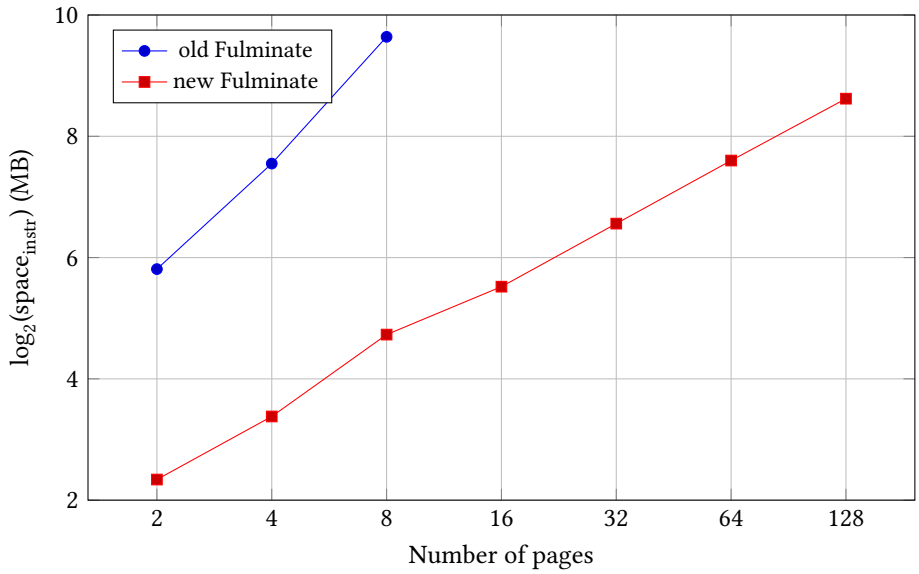


Fig. 10. Number of pages requested from the pKVM buddy allocator plotted against $\log_2$ of the space usage of the instrumented binary.

Set of examples	**old** t_{instr} (s)		**new** t_{instr} (s)		**old** s_{instr} (MB)		**new** s_{instr} (MB)		$s_{\text{difference}}$ (%)
	mean	SD	mean	SD	mean	SD	mean	SD	
CN tutorial	< 0.01	0.0	< 0.01	0.0	1.12	0.07	1.14	0.02	1.54
Verifast examples	< 0.01	0.0	< 0.01	0.0	1.13	0.03	1.14	0.01	1.42

Fig. 11. Performance metrics for tutorial examples.

# elems	**old** $t_{\text{instr}}(s)$	**new** $t_{\text{instr}}(s)$	$t_{\text{improvement}}$ (%)	**old** s_{instr} (MB)	**new** s_{instr} (MB)	$s_{\text{improvement}}(\times)$
2	< 0.01	< 0.01	•	0.112	0.112	1
4	< 0.01	< 0.01	•	0.144	0.112	1.29
8	< 0.01	< 0.01	•	0.240	0.112	2.14
16	< 0.01	< 0.01	•	0.544	0.128	4.25
32	< 0.01	< 0.01	•	1.62	0.160	10.1
64	< 0.01	< 0.01	•	5.55	0.240	23.1
128	0.01	0.01	0	20.6	0.480	41.9
256	0.04	0.04	0	79.2	0.848	93.4
512	0.19	0.18	5.26	311	1.50	207
1024	oom	0.76	-	oom	2.80	-
2048	oom	3.23	-	oom	5.73	-
4096	oom	13.36	-	oom	10.9	-
8192	oom	54.76	-	oom	20.8	-

Fig. 12. Performance metrics for a stack example taken from Software Foundations, Vol. 6 [15], exercised using different numbers of stack elements (from 2^1 to 2^{13}). Entries marked 'oom' signify where the old version of Fulminate ran out of memory and failed to complete the whole execution. • means no reasonable comparison could be made between two negligible numbers.

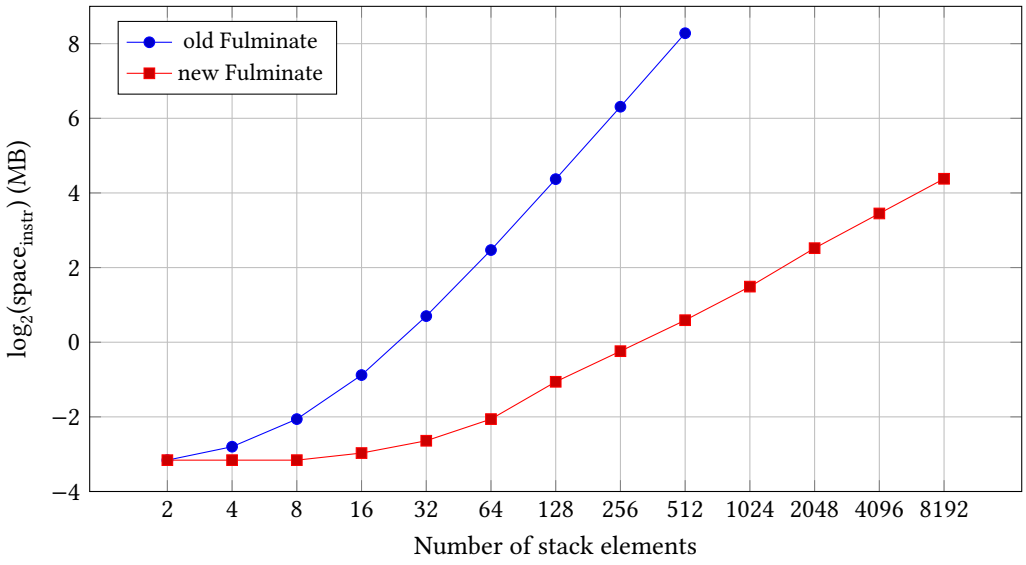


Fig. 13. Number of stack elements plotted against $\log_2$ of the space usage of the instrumented binary for a stack example taken from Software Foundations, Vol. 6 [15] (using both old and new Fulminate).

CN-annotated source

```
unsigned int f(unsigned int *n)
/*@
  requires take X = RW<unsigned int>(n);
  ensures take Y = RW<unsigned int>(n);
@*/
{
    unsigned int i = 0;
    while(i < *n)
    /*@ inv take Z = RW<unsigned int>(n);
        Z == X && i <= Z;
        {n} unchanged; @*/
    { i++; }
    return i;
}
```

Fulminate-instrumented loop

```
{
  /* 1. Variable declarations */
  cn_bump_frame_id __cn_bump_count_a_616;
  struct loop_ownership *a_587;
  cn_bits_u32 *O_i2;
  cn_pointer *O_n2;
  cn_bits_u32 *Z;
  while (({
    /* 2. Set up bump allocator and state for tracking owned addresses */
    __cn_bump_count_a_616 = cn_bump_get_frame_id();
    a_587 = initialise_loop_ownership_state();
    /* 3. Runtime loop invariant checks (first two inserted by CN) */
    O_i2 = owned_unsigned_int(i_addr_cn, LOOP, a_587);
    O_n2 = owned_unsigned_int_pointer(n_addr_cn, LOOP, a_587);
    Z = owned_unsigned_int(O_n2, LOOP, a_587);
    cn_assert(cn_bool_and(cn_bits_u32_equality(Z, X_cn), \
      cn_bits_u32_le(O_i2, Z)), LOOP);
    cn_assert(cn_pointer_equality(O_n2, n_cn), LOOP);
    /* 4. Check for leaks and put back ownership */
    cn_loop_leak_check();
    cn_loop_put_back_ownership(a_587);
    0;
    }), /* 5. Check loop condition */
    CN_LOAD(i) < CN_LOAD(*CN_LOAD(n)))
      {
          CN_POSTFIX(i, ++);
          /* 6. Free bump allocator before next iteration */
          cn_bump_free_after(__cn_bump_count_a_616);
      }
      /* 7. Free bump allocator once loop condition is false */
      cn_bump_free_after(__cn_bump_count_a_616);
}
```

Fig. 14. Top left: A CN-annotated function that loops while incrementing a counter i, stopping and returning it once equal to the value of the pointer n passed in. Bottom left and right: the Fulminate-instrumented loop.

loop condition as a GCC-statement-expression, enclosed in the parentheses followed by curly braces.

We considered two schemes for loop invariant checks: one could either (a) treat the loop as a tail-recursive function with the invariant as both its pre- and post-condition, or (b) insert the

runtime loop invariant checks on entry to the loop, just before the evaluation of the loop condition, and handle ownership in a new manner (described below). The first is conceptually simpler, but the second preserves the structure of the source loop body rather than transforming it, leaving the Fulminate-instrumented output closer to the source; and checks the functional properties only once as required (in (a), these would have been checked twice, unnecessarily).

We show how runtime loop invariants are checked in Fulminate for the example in Fig. 14. First, the loop is enclosed in a block and several variables are declared: a variable for recording the pointer to Fulminate's internal bump allocator on loop entry, a linked list for tracking owned addresses in the invariant, and three ghost variables (lines 2–7). Next, we initialise the bump allocator frame ID and loop-ownership-tracking variables (lines 9–11) and then perform the runtime loop invariant checks (lines 12–18): taking ownership amounts to mapping addresses to one less than the current ghost stack depth in this scheme (lines 13–15), and functional properties are checked in the same way as usual (lines 16–18). After this, Fulminate performs a leak check (line 20), checking if any addresses are still mapped to the current ghost stack depth, and if this succeeds, traverses the linked list of loop-owned addresses and maps each address back to the current ghost stack depth (line 21). Finally, the instrumented loop condition is checked (line 24); if true, the instrumented loop body is entered and the bump allocator is freed before re-entering the loop (line 28), and if false, the loop is exited and the bump allocator is still freed (line 31).

A.3 LLDB Tooling Integration

Although Fulminate offers high compatibility with existing debuggers such as GDB and LLDB by embedding separation-logic assertions directly into C code and leveraging standard DWARF debug information, debugging an instrumented binary remains challenging. Primitive data types, such as pointers in the specification language, cannot be mapped directly to native C types. Furthermore, algebraic data types in the specification language are compiled into combinations of C structs and unions, which makes analysing their values laborious: one must manually traverse multiple fields and layers of unions and structs within a debugger.

To overcome these difficulties, we implemented two categories of LLDB commands: (a) a general-purpose printing command for inspecting ghost states embedded in the binary, and (b) specialised commands tailored to the pKVM hyp allocator.

The general-purpose printer `cn_print` unpacks and prints primitive ghost-state data types, such as `cn_pointer`, in a human-readable form. This command accepts a pointer expression as its argument, evaluates it using LLDB's native evaluation mechanisms, determines its type, and if it is a boxed primitive type, extracts and prints the underlying value. When the argument is a CN struct, the command recursively unpacks its fields and formats the result for readability. For example, whereas invoking the standard LLDB print command on a ghost-state struct produces a raw view of the boxed data and pointer values, the `cn_print` command presents the unpacked and dereferenced data directly in an intuitive form:

```
(lldb) print *C_post_cn
(a_11184_record) {
  alloc_size = 0x0000000102c30f68
  header_address = 0x0000000102c31038
  mapped_size = 0x0000000102c30f6c
  va_size = 0x0000000102c31040
}
```

```
(lldb) cn_print C_post_cn
{
  alloc_size = 104
  header_address = 5771395072
  mapped_size = 136
  va_size = 136
}
```

Second, we developed commands specific to the pKVM hyp allocator to inspect both the ghost-state and the real state of chunk lists. To display the ghost-state chunk list, our command interprets

the embedded `cn_chunk_hdrs` data structure and prints the list in a concise and comprehensible format.

```
(lldb) print *HA_post_cn->lseg->after
(cn_hdrs) {
  tag = CHUNK_CONS
  cntype = 0x000000000000fc01
  u = {
    chunk_nil = 0x000000010298d508
    chunk_cons = 0x000000010298d508
  }
}
```

```
(lldb) cn_print_cn_hdrs HA_post_cn->lseg->after
[
  {alloc_size=0, header_address=5771395208,
     mapped_size=296, va_size=296},
  {alloc_size=400, header_address=5771395504,
    mapped_size=3664, va_size=65104}
]
```

To inspect the concrete allocator state, we implemented a command that traverses and prints the real `hyp_allocator` and `chunk_hdr` structures, producing a readable trace of the chunk list. The following is an example output that shows the allocator metadata and a sequential list of chunks with their information:

```
(lldb) hyp
hyp_allocator(0x100036058) metadata:
  start = 0x130008000
  size  = 65536
Traversing chunk list:
0x0000000130008000: alloc_size=104, mapped_size=136, hash=0
0x0000000130008088: alloc_size=0, mapped_size=296, hash=0
0x00000001300081b0: alloc_size=400, mapped_size=3664, hash=0
```

A.4 Darcy versus Bennet

A.4.1 Original Bennet case studies. We ran the case studies from the Bennet paper [1], on an M3 MacBook Pro with 32GB RAM. Bennet used the Etna [71] platform for automatically injecting manually specified faults (mutants) into multiple pieces of software (workloads). Then for each mutant, we ran Bennet and Darcy on each function that could trigger the bug (tasks). With a timeout of two minutes, we recorded if the bug was discovered or not, how many inputs and how much time it took.

We also ran Darcy with a max array length of 4096 (Darcy-4096), which was used when testing the hyp allocator, for the case studies with arrays.

The original Bennet paper used a one minute timeout, which we doubled for Darcy, to account for the increased overhead. Some trials on the MKM exceeded a minute, one for Darcy and two for Darcy-4096, on the partially solved task. For the AVL tree, one trial was over a minute on the partially solved task. Finally, one went over a minute for Darcy, in the ring queue task that was unsolved by Darcy-4096.

Workload	Tasks	Bennet			Darcy		
		Solved (Partial)	Inputs	Time (s)	Solved (Partial)	Inputs	Time (s)
Ring Queue	4	4 (0)	2.9 (± 0.60)	0.48 (± 0.00)	4 (0)	8.8 (± 2.90)	5.66 (± 1.97)
Runway	6	6 (0)	25.6 (± 7.5)	0.87 (± 0.00)	6 (0)	157.6 (± 72.4)	1.81 (± 0.61)
MKM	2	2 (0)	34.0 (± 9.60)	0.60 (± 0.00)	1 (1)	3.4 (± 0.70)	0.74 (± 0.01)
Sorted List	6	6 (0)	9.2 (± 2.1)	0.45 (± 0.01)	6 (0)	18.2 (± 5.60)	0.76 (± 0.08)
BST	32	32 (0)	22.1 (± 6.2)	0.55 (± 0.00)	32 (0)	38.5 (± 10.0)	1.68 (± 0.31)
AVL	26	26 (0)	384.0 (± 86.2)	0.69 (± 0.05)	25 (1)	57.5 (± 16.4)	1.94 (± 0.47)

Workload	Tasks	Darcy-4096		
		Solved (Partial)	Inputs	Time (s)
Ring Queue	4	3 (0)	6.0 (± 1.5)	11.80 (± 1.31)
MKM	2	2 (0)	769.4 (± 257.4)	15.52 (± 5.01)

For tasks that were completely solved, the mean and standard error of the number of inputs and time taken to find the bugs are provided.

On these case studies, Bennet solves all tasks across all workloads, while Darcy solves nearly all, with two partially solved (one each in MKM and AVL). Bennet is consistently faster and requires fewer inputs, as expected for these simpler data structures whose preconditions involve mostly local constraints that are easy to satisfy by random backtracking search. Darcy-4096 recovers the partially-solved MKM task but at the cost of increased overhead due to the larger search space, and loses one ring queue task. Overall, Bennet's random backtracking is well suited to these simpler, local constraints; the hyp allocator results below show where it breaks down.

A.4.2 Hyp allocator bug-finding study. For each bug from the hyp allocator, we ran at least 10 trials with a timeout of three minutes, on an M3 MacBook Pro with 32GB RAM. For the bug that was found flakily, we ran 50 trials. For the bugs that required a max array length of 4096, we mark them as such, as well as the bugs we did not find in our first effort. For the remaining bugs we ran Darcy without the array length increase.

We also ran Bennet for all the bugs. For the tasks that were solved, we report the mean and standard error for the number of inputs and time taken to find the bugs.

Strategy	Tasks	Solved (Partial)	Inputs	Time (s)
Bennet	24	3 (0)	1.3 (± 0.2)	1.02 (± 0.01)
Darcy	24	18 (4)	2.6 (± 0.5)	5.90 (± 2.16)

Out of the four partially solved tasks, one was our flakily-found bug, which was found in 42 out of 50 trials.

Two of them were each a single failed trial for one of the bugs we found consistently. The number of inputs used for these two was quite large. Since we follow QuickCheck [18]'s approach to sized inputs, the test case size grows over time. However, prior work [71] has shown that large test case sizes are worse for finding bugs related to the interaction of multiple arguments. In these cases, it failed to find a failing input at the lower sizes, resulting in it being less likely for each following test case, as the size grew.

The last of the partially solved tasks was one of the bugs we failed to detect. Due to successive improvements to the runtime, more inputs were able to be run within a three minute window, resulting in a single trial detecting the bug.

Finally, there were the 3 remaining bugs that were not found in any trial.

The difference between the two tools is stark: Bennet finds only 3 of the 24 bugs, all in functions with relatively simple preconditions, and fails to generate a single valid input for the remaining functions. Darcy finds 18 bugs across all trials and 4 across some, demonstrating its ability to handle the complex interleaved pointer arithmetic and ownership constraints that arise in the hyp allocator's specifications. The 2 unsolved bugs are in functions that call `chunk_install`, where the preconditions are deep enough that it is not straightforward to strengthen them to guide generation towards the buggy paths. Both of these were, however, found by Fulminate.

A.4.3 Time to generate inputs. Darcy and Bennet use "sized generation" as introduced by QuickCheck [18]. However, the way they select sizes is different, with Bennet uniformly sampling from $[1, \texttt{max\_size}]$ and Darcy using the QuickCheck algorithm.

There is also the matter that Bennet abandons some generation attempts based on per-input timeouts (as well as other conditions). However, Darcy does not currently use timeouts for the SMT solver, which would be the equivalent.

As such, we control for various features, to have a more meaningful comparison. First, we disable the input timeouts for Bennet, so that we can see when the random search gets stuck. We made the test case sizes constant and generated 100 test inputs at various sizes. We used a timeout of 2 minutes for the Bennet case studies and 3 minutes for the hyp allocator.

We calculated the final numbers by averaging over the timeout or the time taken to generate 100 inputs (if 100 were generated within the time limit), per function. We took the time taken relative to Bennet per function, to account for the varying complexity of different preconditions. We then took the geometric mean of these ratios to get our final results. While less meaningful, we also provide the actual average times as well.

AVL-1						
	Relative Time per Input			Average Time (ms)		
Size	**Bennet**	**Darcy**	**Darcy-4096**	**Bennet**	**Darcy**	**Darcy-4096**
1	1x	1200.02x	1235.07x	0.010	4.783	4.885
2	1x	668.33x	690.60x	0.050	9.364	9.643
4	1x	116.60x	113.59x	1.160	30.238	28.794
8	1x	7.62x	9.80x	36.443	52.894	65.552
16	1x	0.20x	0.40x	406.166	22.134	51.896
32	1x	0.03x	0.06x	486.998	9.843	25.653
Avg	1x	13.05x	16.63x	155.138	21.543	31.070

The Bennet case studies have multiple versions of the data structure invariant for some of them, in order to demonstrate how Bennet's performance varies across them. Looking at the table we can see that Bennet is outperformed by Darcy for this version of the AVL tree. This version is hard for Bennet to optimize, so a lot of backtracking is performed. For larger sizes, the cost of this becomes quite large. At the same time, Darcy intelligently prunes branches that won't satisfy AVL invariant. AVL trees are required to be balanced, which depends on when the recursive branch of the generator is taken. As such, Darcy's branch pruning at test time learns the set of paths that result in balanced trees.

AVL-2									
	Relative Time per Input			Average Time (ms)			No Valid Inputs		
Size	**Bennet**	**Darcy**	**Darcy-4096**	**Bennet**	**Darcy**	**Darcy-4096**	**Bennet**	**Darcy**	**Darcy-4096**
1	1x	221.87x	235.19x	0.117	4.593	4.860	1	1	1
2	1x	274.73x	272.90x	0.121	6.476	6.569			
4	1x	152.51x	145.49x	0.135	11.663	10.366	1		
8	1x	132.14x	181.91x	0.147	11.318	14.219			
16	1x	89.44x	88.03x	0.127	7.587	7.487			
32	1x	80.88x	102.27x	0.132	5.851	7.478			
Avg	1x	139.57x	154.42x	0.130	7.915	8.496			

This AVL tree variant is more amenable to Bennet's optimizations and as such, does not see the same improvement by Darcy.

BST-1						
	Relative Time per Input			Average Time (ms)		
Size	**Bennet**	**Darcy**	**Darcy-4096**	**Bennet**	**Darcy**	**Darcy-4096**
1	1x	1620.22x	1872.34x	0.004	5.926	6.828
2	1x	1754.00x	1944.34x	0.003	5.914	6.561
4	1x	1322.97x	1473.20x	0.006	8.280	9.182
8	1x	277.99x	286.88x	0.038	10.663	11.007
16	1x	41.64x	41.94x	0.682	28.561	28.598
32	1x	6.40x	6.91x	31.372	205.412	216.907
Avg	1x	255.58x	276.42x	5.351	44.126	46.514

The predicate for BST-1 is the same as the predicate for AVL-1, just without the balanced requirement. As such, we can see that Darcy does not improve as much at the larger sizes (though there is still an improvement).

BST-2									
	Relative Time per Input			Average Time (ms)			No Valid Inputs		
Size	**Bennet**	**Darcy**	**Darcy-4096**	**Bennet**	**Darcy**	**Darcy-4096**	**Bennet**	**Darcy**	**Darcy-4096**
1	1x	1438.28x	1398.67x	0.005	6.465	6.289	1		
2	1x	178.15x	202.45x	0.033	6.020	6.851			
4	1x	1351.28x	1219.26x	0.007	8.973	9.084			
8	1x	716.38x	752.56x	0.017	12.033	12.863			
16	1x	336.23x	343.22x	0.058	20.633	20.294			
32	1x	301.66x	316.81x	0.102	30.697	32.870			
Avg	1x	518.79x	530.01x	0.037	14.137	14.708			

Runway						
	Relative Time per Input			Average Time (ms)		
Size	**Bennet**	**Darcy**	**Darcy-4096**	**Bennet**	**Darcy**	**Darcy-4096**
1	1x	1356.59x	1427.28x	0.004	4.718	4.960
2	1x	2633.61x	2772.41x	0.002	4.785	5.039
4	1x	2488.48x	2643.85x	0.002	4.815	5.117
8	1x	2435.11x	2575.47x	0.002	4.825	5.101
16	1x	2738.49x	2870.88x	0.002	4.777	5.007
32	1x	2961.13x	3034.26x	0.002	4.763	4.881
Avg	1x	2366.31x	2483.64x	0.002	4.780	5.018

Ring Queue									
	Relative Time per Input			Average Time (ms)			No Valid Inputs		
Size	**Bennet**	**Darcy**	**Darcy-4096**	**Bennet**	**Darcy**	**Darcy-4096**	**Bennet**	**Darcy**	**Darcy-4096**
1	1x	1.93x	100.17x	1183.229	17.427	904.099	1	1	1
2	1x	1.99x	98.92x	18.474	18.167	904.638		1	1
4	1x	2.03x	100.24x	12.330	18.446	908.704		1	1
8	1x	1.92x	100.16x	10.823	17.645	921.145		1	1
16	1x	1.97x	98.24x	9.655	18.000	899.085		1	1
32	1x	1.91x	101.50x	9.466	17.667	938.689		1	1
Avg	1x	1.96x	99.87x	207.330	17.892	912.727			

The ring queue's "put" function uses a recursive function over an array in its specification, leading to Darcy stalling on the constraint that the number of elements is less than the size of the buffer.

pKVM hyp allocator									
	Relative Time per Input			Average Time (ms)			No Valid Inputs		
Size	**Bennet**	**Darcy**	**Darcy-4096**	**Bennet**	**Darcy**	**Darcy-4096**	**Bennet**	**Darcy**	**Darcy-4096**
1	1x	3649.02x	4141.81x	0.542	173.350	764.520	8	1	1
2	1x	5721.27x	6715.57x	0.697	555.804	1775.551	8		
4	1x	4477.89x	4427.35x	0.598	1289.845	1314.784	8		
8	1x	3637.62x	4181.39x	1.450	1374.905	1625.325	8		
16	1x	1855.28x	2495.73x	1.619	1307.379	1718.104	8		
32	1x	1739.07x	1788.37x	1.645	1105.735	1769.469	8		
Avg	1x	3211.54x	3632.72x	1.092	967.836	1494.626			

For the hyp allocator, Bennet can't generate valid inputs for the interesting preconditions, so the relative times are not very useful. As such, the absolute times for Darcy should be paid attention to.

In summary, Darcy generally has a higher per-input cost than Bennet but far broader applicability, as the bug-finding study confirms.

The remaining Bennet case studies are not all that interesting and are included below, without comment, for completeness.

Remaining case studies.

SortedList-1						
	Relative Time per Input			Average Time (ms)		
Size	**Bennet**	**Darcy**	**Darcy-4096**	**Bennet**	**Darcy**	**Darcy-4096**
1	1x	2005.70x	2097.70x	0.003	6.478	6.776
2	1x	2009.75x	2080.18x	0.004	8.340	8.633
4	1x	1326.33x	1483.54x	0.007	9.855	11.023
8	1x	283.90x	379.76x	0.039	11.138	14.898
16	1x	8.13x	7.78x	1.560	12.675	12.132
32	1x	0.01x	0.02x	862.557	11.371	14.990
Avg	1x	73.88x	83.23x	144.028	9.976	11.409

SortedList-2						
	Relative Time per Input			Average Time (ms)		
Size	**Bennet**	**Darcy**	**Darcy-4096**	**Bennet**	**Darcy**	**Darcy-4096**
1	1x	330.52x	358.51x	0.020	6.491	7.041
2	1x	502.33x	494.81x	0.018	8.942	8.808
4	1x	828.73x	773.32x	0.014	11.710	10.927
8	1x	1024.71x	975.92x	0.012	11.999	11.428
16	1x	892.33x	977.64x	0.014	12.368	13.550
32	1x	710.89x	976.98x	0.018	12.533	17.224
Avg	1x	668.74x	709.79x	0.016	10.674	11.496

SortedList-3						
	Relative Time per Input			Average Time (ms)		
Size	**Bennet**	**Darcy**	**Darcy-4096**	**Bennet**	**Darcy**	**Darcy-4096**
1	1x	291.29x	235.19x	0.023	6.574	5.308
2	1x	455.24x	461.30x	0.018	8.185	8.294
4	1x	374.64x	360.23x	0.027	10.171	9.780
8	1x	592.01x	659.83x	0.020	11.787	13.137
16	1x	846.55x	908.02x	0.015	12.393	13.293
32	1x	318.00x	455.92x	0.032	10.227	14.662
Avg	1x	446.44x	469.24x	0.023	9.889	10.746

MKM						
	Relative Time per Input			Average Time (ms)		
Size	**Bennet**	**Darcy**	**Darcy-4096**	**Bennet**	**Darcy**	**Darcy-4096**
1	1x	440.29x	430.45x	0.034	14.877	14.545
2	1x	429.30x	421.04x	0.035	14.974	14.686
4	1x	456.02x	441.74x	0.034	15.418	14.935
8	1x	402.27x	406.57x	0.037	14.872	15.031
16	1x	437.12x	443.37x	0.036	15.636	15.860
32	1x	410.25x	408.91x	0.038	15.528	15.477
Avg	1x	428.82x	425.10x	0.036	15.218	15.089

References

[1] Zain K Aamer and Benjamin C. Pierce. 2025. Bennet: Randomized Specification Testing for Heap-Manipulating Programs. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 413 (Oct. 2025), 30 pages. doi:10.1145/3764115

[2] Sergio Antoy, Rachid Echahed, and Michael Hanus. 2000. A Needed Narrowing Strategy. *Journal of the ACM (JACM)* 47, 4 (2000), 776–822.

[3] Vytautas Astrauskas, Aurel Bílý, Jonás Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. 2022. The Prusti Project: Formal Verification for Rust. In *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13260)*, Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez (Eds.). Springer, 88–108. doi:10.1007/978-3-031-06773-0_5

[4] Rini Banerjee, Kayvan Memarian, Dhruv Makwana, Christopher Pulte, Neel Krishnaswami, and Peter Sewell. 2025. Fulminate: Testing CN Separation-Logic Specifications in C. *Proc. ACM Program. Lang.* 9, POPL, Article 43 (jan 2025), 33 pages. doi:10.1145/3704879

[5] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. 2005. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 49–69.

[6] Thibaut Benjamin and Julien Signoles. 2023. Abstract Interpretation of Recursive Logic Definitions for Efficient Runtime Assertion Checking. In *Tests and Proofs - 17th International Conference, TAP 2023, Leicester, UK, July 18-19, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 14066)*, Virgile Prevosto and Cristina Seceleanu (Eds.). Springer, 168–186. doi:10.1007/978-3-031-38828-6_10

[7] Sandrine Blazy and Xavier Leroy. 2009. Mechanized Semantics for the Clight Subset of the C Language. *J. Autom. Reason.* 43, 3 (2009), 263–288. doi:10.1007/S10817-009-9148-3

[8] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. 2021. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 836–850. doi:10.1145/3477132.3483540

[9] Marc Brooker and Ankush Desai. 2024. Systems Correctness Practices at AWS: Leveraging Formal and Semi-formal Methods. *ACM Queue* 22, 6 (2024), 60. doi:10.1145/3712057

[10] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems*

Design and Implementation (OSDI'08). USENIX Association, USA, 209–224.

[11] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 459–465. doi:10.1007/978-3-642-20398-5_33

[12] 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 (2018), 367–422. doi:10.1007/s10817-018-9457-5

[13] Mathieu Carlier, Catherine Dubois, and Arnaud Gotlieb. 2010. Constraint Reasoning in FocalTest. In *ICSOFT*. https://inria.hal.science/hal-00699233

[14] Matthieu Carlier, Catherine Dubois, and Arnaud Gotlieb. 2013. FocalTest: A Constraint Programming Approach for Property-Based Testing. In *Software and Data Technologies*, José Cordeiro, Maria Virvou, and Boris Shishkov (Eds.). Springer, Berlin, Heidelberg, 140–155. doi:10.1007/978-3-642-29578-2_9

[15] Arthur Charguéraud. 2024. *Separation Logic Foundations*. Software Foundations, Vol. 6. Electronic textbook. https://softwarefoundations.cis.upenn.edu Version 2.2.

[16] Koen Claessen, Jonas Duregård, and Michał H. Pałka. 2014. Generating Constrained Random Data with Uniform Distribution. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 18–34.

[17] Koen Claessen, Jonas Duregård, and Michał H. Pałka. 2015. Generating constrained random data with uniform distribution. *Journal of Functional Programming* 25 (2015), e8. doi:10.1017/S0956796815000143

[18] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279. doi:10.1145/351240.351266

[19] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 2988)*, Kurt Jensen and Andreas Podelski (Eds.). Springer, 168–176. doi:10.1007/978-3-540-24730-2_15

[20] Joseph W. Cutler, Craig Disselkoen, Aaron Eline, Shaobo He, Kyle Headley, Michael Hicks, Kesha Hietala, Eleftherios Ioannidis, John Kastner, Anwar Mamat, Darin McAdams, Matt McCutchen, Neha Rungta, Emina Torlak, and Andrew M. Wells. 2024. Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 118 (April 2024), 28 pages. doi:10.1145/3649835

[21] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[22] Will Deacon. 2020. Virtualisation for the Masses: Exposing KVM on Android. KVM Forum slides, https://mirrors.edge.kernel.org/pub/linux/kernel/people/will/slides/kvmforum-2020-edited.pdf. Accessed 2022-07-07.

[23] Rafael Dutra, Jonathan Bachrach, and Koushik Sen. 2018. SMTSampler: Efficient Stimulus Generation from Complex SMT Constraints. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. doi:10.1145/3240765.3240848

[24] GNU Foundation. 2025. Statements and Declarations in Expressions. https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html. [Online; accessed 13-November-2025].

[25] S.J. Garland, J.V. Guttag, K.D. Jones, J.J. Horning, A. Modet, and J.M. Wing. 2012. *Larch: Languages and Tools for Formal Specification*. Springer New York. https://books.google.com/books?id=4_LxBwAAQBAJ

[26] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. 2010. Test generation through programming in UDITA. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 225–234. doi:10.1145/1806799.1806835

[27] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 213–223. doi:10.1145/1065010.1065036

[28] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (March 2012), 40–44. doi:10.1145/2093548.2093564

[29] 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* (Mumbai, India) *(POPL '15)*. ACM, New York, NY, USA, 595–608. doi:10.1145/2676726.2676975

[30] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 653–669. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

[31] Klaus Havelund and Grigore Rosu. 2018. Runtime Verification - 17 Years Later. In *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11237)*, Christian Colombo and Martin Leucker (Eds.). Springer, 3–17. doi:10.1007/978-3-030-03769-7_1

[32] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael Lowell Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, Ethan L. Miller and Steven Hand (Eds.). ACM, 1–17. doi:10.1145/2815400.2815428

[33] Robert M. Hierons. 1997. Testing from a Z Specification. *Software Testing, Verification and Reliability* 7, 1 (1997), 19–33.

[34] Daniel Jackson and Jeannette Wing. 1996. *IEEE Computer*. Chapter Formal Methods Light §Lightweight formal methods, 21–22. doi:10.1109/MC.1996.10038

[35] 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 - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 41–55. doi:10.1007/978-3-642-20398-5_4

[36] Bart Jacobs, Jan Smans, and Frank Piessens. 2024. The VeriFast Program Verifier: A Tutorial. doi:10.5281/zenodo.13380705

[37] Cliff B. Jones. 1996. *IEEE Computer*. Chapter Formal Methods Light §A rigorous approach to formal methods, 20–21. doi:10.1109/MC.1996.10038

[38] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. doi:10.1017/S0956796818000151

[39] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2010. seL4: formal verification of an operating-system kernel. *Commun. ACM* 53, 6 (2010), 107–115. doi:10.1145/1743546.1743574

[40] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM Transactions on Computer Systems* 32, 1 (Feb. 2014), 2:1–2:70. doi:10.1145/2560537

[41] Dawid Lachowicz. 2025. Contracts: primitive ownership assertions: owned and block #942. Rust-lang proposal. https://github.com/rust-lang/compiler-team/issues/942. Accessed 2025-11-11.

[42] Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. Beginner's Luck: A Language for Property-Based Generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. Association for Computing Machinery, New York, NY, USA, 114–129. doi:10.1145/3009837.3009868

[43] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2018. Generating Good Generators for Inductive Relations. *Proceedings of the ACM on Programming Languages* 2, POPL (Jan. 2018), 1–30. doi:10.1145/3158133

[44] Butler W. Lampson, James J. Horning, Ralph L. London, James G. Mitchell, and Gerald J. Popek. 1977. Report on the programming language Euclid. *ACM SIGPLAN Notices* 12, 2 (1977), 1–79. doi:10.1145/954666.971189

[45] Quang Loc Le, Makoto Tatsuta, Jun Sun, and Wei-Ngan Chin. 2017. A Decidable Fragment in Separation Logic with Inductive Predicates and Arithmetic. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 495–517. doi:10.1007/978-3-319-63390-9_26

[46] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. 2002. How the Design of JML Accomodates Both Runtime Assertion Checking and Formal Verification. In *Formal Methods for Components and Objects, First International Symposium, FMCO 2002, Leiden, The Netherlands, November 5-8, 2002, Revised Lectures (Lecture Notes in Computer Science, Vol. 2852)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.). Springer, 262–284. doi:10.1007/978-3-540-39656-7_11

[47] Dirk Leinenbach and Thomas Santen. 2009. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*. 806–809. doi:10.1007/978-3-642-05089-3_51

[48] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reason.* 43, 4 (2009), 363–446. doi:10.1007/s10817-009-9155-4

[49] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In *30th USENIX Security Symposium, USENIX Security 2021, August*

11-13, 2021, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 3953–3970. https://www.usenix.org/conference/usenixsecurity21/presentation/li-shih-wei

[50] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. A Secure and Formally Verified Linux KVM Hypervisor. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 839–856. doi:10.1109/SP40001.2021.00049

[51] Fredrik Lindblad. 2007. Property Directed Generation of First-Order Test Data. *Trends in Functional Programming* 8 (2007), 105–123.

[52] Petar Maksimovic, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. 2021. Gillian, Part II: Real-World Verification for JavaScript and C. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 827–850. doi:10.1007/978-3-030-81688-9_38

[53] Gregory Malecha, Gordon Stewart, Frantisek Farka, Jasper Haag, and Yoichi Hirai. 2022. Developing With Formal Methods at BedRock Systems, Inc. *IEEE Secur. Priv.* 20, 3 (2022), 33–42. doi:10.1109/MSEC.2022.3158196

[54] Kayvan Memarian, Ben Simner, David Kaloper-Meršinjak, Thibaut Pérami, and Peter Sewell. 2025. Ghost in the Android Shell: Pragmatic Test-oracle Specification of a Production Hypervisor. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. ACM, Seoul, Republic of Korea. doi:10.1145/3731569.3764817

[55] Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* 25, 10 (1992), 40–51. doi:10.1109/2.161279

[56] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2017. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Dependable Software Systems Engineering*, Alexander Pretschner, Doron Peled, and Thomas Hutzelmann (Eds.). NATO Science for Peace and Security Series - D: Information and Communication Security, Vol. 50. IOS Press, 104–125. doi:10.3233/978-1-61499-810-5-104

[57] Peter Naur and Brian Randell (Eds.). 1969. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO.* http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF. Accessed 2025-11-12.

[58] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 89–100. doi:10.1145/1250734.1250746

[59] Haobin Ni, Antoine Delignat-Lavaud, Cédric Fournet, Tahina Ramananandro, and Nikhil Swamy. 2023. ASN1*: Provably Correct, Non-malleable Parsing for ASN.1 DER. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*, Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve Zdancewic (Eds.). ACM, 275–289. doi:10.1145/3573105.3575684

[60] Long H. Pham, Quang Loc Le, Quoc-Sang Phan, and Jun Sun. 2019. Concolic Testing Heap-Manipulating Programs. In *Formal Methods – The Next 30 Years*, Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira (Eds.). Springer International Publishing, Cham, 442–461. doi:10.1007/978-3-030-30942-8_27

[61] Long H. Pham, Quang Loc Le, Quoc-Sang Phan, Jun Sun, and Shengchao Qin. 2018. Testing Heap-Based Programs with Java StarFinder. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 268–269. doi:10.1145/3183440.3194964

[62] Long H. Pham, Quang Loc Le, Quoc-Sang Phan, Jun Sun, and Shengchao Qin. 2019. Enhancing Symbolic Execution of Heap-Based Programs with Separation Logic for Test Input Generation. In *Automated Technology for Verification and Analysis*, Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza (Eds.). Springer International Publishing, Cham, 209–227. doi:10.1007/978-3-030-31784-3_12

[63] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 181–198. https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[64] Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. 2023. CN: Verifying systems C code with separation-logic refinement types. In *Proceedings of the 50th ACM SIGPLAN Symposium on Principles of Programming Languages*. doi:10.1145/3571194

[65] Christopher Pulte, Benjamin C. Pierce, Cole Schlesinger, and Elizabeth Austell. 2024. CN tutorial. https://rems-project.github.io/cn-tutorial/. [Online; accessed 26-October-2024].

[66] José Fragoso Santos, Petar Maksimovic, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, part i: a multi-language platform for symbolic execution. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 927–942. doi:10.1145/3385412.3386014

[67] Eric L. Seidel, Niki Vazou, and Ranjit Jhala. 2015. Type Targeted Testing. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer, Berlin, Heidelberg, 812–836. doi:10.1007/978-3-662-46669-8_33

[68] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. *SIGSOFT Software Engineering Notes* 30, 5 (Sept. 2005), 263–272. doi:10.1145/1095430.1081750

[69] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Annual Technical Conference, USENIX ATC 2012, Boston, MA, USA, June 13-15, 2012*, Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[70] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized verification of fine-grained concurrent programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 77–87. doi:10.1145/2737924.2737964

[71] Jessica Shi, Alperen Keles, Harrison Goldstein, Benjamin C. Pierce, and Leonidas Lampropoulos. 2023. Etna: An Evaluation Platform for Property-Based Testing (Experience Report). *Proceedings of the ACM on Programming Languages* 7, ICFP (Aug. 2023), 878–894. doi:10.1145/3607860

[72] Xiaomu Shi, Jean-François Monin, Frédéric Tuong, and Frédéric Blanqui. 2011. First Steps towards the Certification of an ARM Simulator Using Compcert. In *Certified Programs and Proofs*, Jean-Pierre Jouannaud and Zhong Shao (Eds.). Lecture Notes in Computer Science, Vol. 7086. Springer Berlin Heidelberg, 346–361. doi:10.1007/978-3-642-25379-9_25

[73] Julien Signoles. 2018. *From Static Analysis to Runtime Verification with Frama-C and E-ACSL*. https://tel.archives-ouvertes.fr/tel-04469397

[74] Julien Signoles. 2021. The e-ACSL perspective on runtime assertion checking. In *VORTEX 2021: Proceedings of the 5th ACM International Workshop on Verification and mOnitoring at Runtime EXecution, Virtual Event, Denmark, 12 July 2021*, Wolfgang Ahrendt, Davide Ancona, and Adrian Francalanza (Eds.). ACM, 8–12. doi:10.1145/3464974.3468451

[75] Julien Signoles, Nikolai Kosmatov, and Kostyantyn Vorobyov. 2017. E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs (tool paper). In *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA (Kalpa Publications in Computing, Vol. 3)*, Giles Reger and Klaus Havelund (Eds.). EasyChair, 164–173. doi:10.29007/FPDH

[76] J.M. Spivey. 1988. *Understanding Z: A Specification Language and Its Formal Semantics*. Cambridge University Press. https://books.google.com/books?id=tcTIiZ3Dnn8C

[77] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 256–270. https://www.fstar-lang.org/papers/mumon/

[78] Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. 2021. Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 866–881. doi:10.1145/3477132.3483560

[79] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2018. Refinement reflection: complete verification with SMT. *Proc. ACM Program. Lang.* 2, POPL (2018), 53:1–53:31. doi:10.1145/3158141

[80] Jeannette M. Wing. 1995. Hints for Writing Specifications. In *ZUM '95: The Z Formal Specification Notation, 9th International Conference of Z Users, Limerick, Ireland, September 7-9, 1995, Proceedings (Lecture Notes in Computer Science, Vol. 967)*, Jonathan P. Bowen and Michael G. Hinchey (Eds.). Springer, 497. doi:10.1007/3-540-60271-2_139

[81] Litao Zhou, Jianxing Qin, Qinshi Wang, Andrew W. Appel, and Qinxiang Cao. 2024. VST-A: A Foundationally Sound Annotation Verifier. *Proc. ACM Program. Lang.* 8, POPL (2024), 2069–2098. doi:10.1145/3632911

Received 2025-11-14; accepted 2026-04-03