Live Verification in C via In-Situ Proof Code

Abstract. A recently proposed Live Verification framework promises real-time feedback during program-proof co-development of C programs in the Rocq theorem prover. However, it supports only a simplified subset of C, limiting its applicability to realistic C programs. We present C*, a verifier that shifts the *prover-centric* approach to a *program-centric* one: users write implementations, specifications, and in-situ proof code in C, while a proof-supporting verifytime separates program-logic reasoning from assertion-logic reasoning. C* supports a substantial subset of C and is extensible via user-defined proof libraries and trusted automation procedures. We evaluate C* on standard benchmarks and conduct a case study towards verifying a sequential version of pKVM's `hyp_allocator`, demonstrating that live verification scales to realistic C code.

Keywords: C program verification · In-situ proof · Theorem proving

1 Introduction

Modern proof assistants such as Rocq, Isabelle/HOL, and Lean have matured into vibrant platforms for engineering formally verified software [3,36]. This is reflected in a growing body of verification frameworks, including Bedrock [11,12], VST [1,8], Iris [26,28], and many others [10,14,16,19–21,23,33,42]. These frameworks are highly expressive: they embed the semantics of target programs into a host proof assistant, and hence inherit its higher-order specification logic, mathematical libraries, and proof-programming facilities. Such high expressiveness for specification and proof has been indispensable for scaling functional-correctness verification to handle both the complexity and the tediousness of reasoning required in real-world systems software [15,22,27].

The recent proposal of Live Verification by Gruetter et al. [20] suggests a promising direction for further improving the usability of these embedded verification frameworks for systems software developers. Specifically, their framework allows users to develop a program in C syntax, along with its proofs, in a single Rocq script. During development, Rocq's goal panel displays a summary of the verification state (also called the symbolic state) at the cursor position; users can inspect this summary and, whenever a non-trivial reasoning step is needed, provide guidance using proof tactics that lead the symbolic state into a desired form before proceeding to the next program statement. Such a *live verification* experience provides real-time feedback on verification progress and keeps proofs *in situ* with the program, promoting co-development and co-evolution of the implementation code and its correctness reasoning.

However, Gruetter et al.'s Live Verification framework supports only a simplified subset of C. For example, it requires all values to have the single `uintptr_t`

type, which rules out structure definitions and field accesses; it does not support jump statements such as `break` and `continue`; and it does not support nested function calls or taking addresses of variables. The key idea of their approach is to leverage Rocq's Ltac meta-programming to treat C snippets as syntactic sugar for program constructors and program-reasoning tactics, but it is unclear how this syntax-embedding approach could be extended systematically to support richer C features. This limited language support hinders its applicability and prevents further investigation of the benefits of program-proof co-development in more practical scenarios.

This work. We aim to develop a verifier for C that: (i) supports live verification, in which verification proceeds via automatic forward symbolic execution of program statements, interleaved with manual proof steps and real-time feedback; (ii) supports more C features for practical development, including the commonly used ones mentioned above; and (iii) achieves the level of expressiveness in proof support as that of verification frameworks embedded in proof assistants.

We propose C*, a new design and a prototype implementation of a verifier that meets all three goals. Our guiding philosophy is a shift from the *prover-centric* view of the Live Verification framework to a *program-centric* view, i.e., rather than embedding more C syntactic constructs into a proof script, we bring the high proof power of proof assistants into the world of C programming. This shift leads to the following language and architectural design:

Proof-integrated programs. C* accepts C programs with verification-related annotations as input. These annotations include behavioral specifications broadly similar to those in assertion-based program verifiers [7,13,25,34], such as pre- and post-conditions, and loop invariants. More distinctively, C* allows *in-situ proof code* to be interleaved with normal program statements. This proof code is written in C, using the same language constructs as the implementation code, but executes only at verification time in a proof-supporting environment, as described next.

Proof-supporting verifytime. Verification of a C* program takes place in a proof-supporting *verifytime*, a term we use to distinguish it from the vanilla *runtime* environment used for deployment. The verifytime decouples the two reasoning concerns underlying live verification: (i) *program-logic reasoning* is handled by a symbolic-execution engine that captures the semantics of program statements; and (ii) *assertion-logic reasoning* is supported by a proof interface for user-guided theorem proving. During verification, program statements are fed to the symbolic-execution engine incrementally; in-situ proof code steers the symbolic execution by using the proof interface to prove an entailment from the current symbolic state to a desired one, which the engine then soundly installs before proceeding to the next statement.

Our implementation consists of a front end that translates each C* file into a verification C program, and a core verifytime in which this program is executed to perform the verification. Our core verifytime builds on existing works supplying the two components as C libraries: a separation-logic symbolic-execution

engine [41] for a substantial subset of C, and an LCF-style proof interface [9] for trustworthy theorem proving in C. It is extensible: additional proof facilities built as C libraries can be linked into the verifytime and used in proof code.

Contributions. In this article, we present the design and implementation of C* and report our experience with it. Our contributions are:

- We design C*, a verifier for C that supports live verification and incorporates both behavioral specifications and proof-programming capabilities. It enables users to write implementation and proof code in the same language, within the same program and development environment.
- We implement a prototype for C*, comprising a front end and a core verifytime built on existing symbolic-execution [41] and theorem-proving [9] libraries. Outside the core, we build additional proof libraries and trusted automation procedures that we use throughout our evaluation.
- We use C* to verify a suite of common examples, including arithmetic functions, linked data structures, and array programs. To showcase C* as a promising verifier for practical program-proof co-development, we incorporate pKVM's `hyp_allocator`, retrospectively redevelop a sequential version of it, and report our ongoing experience towards verifying its memory safety and functional correctness in C*.[1]

Roadmap. Section 2 provides an overview using a concrete C* program example. Section 3 presents the core design of C*. Section 4 describes the verifytime extensions we experimented with. Section 5 summarizes our evaluation benchmarks and presents the `hyp_allocator` case study. Section 6 compares with related C program verifiers and verification frameworks. Section 7 concludes.

2 Overview: A Guided Tour of C*

In this section, we use an example to illustrate the live verification experience and the proof power that C* aims to provide in a unified C programming environment: users write implementation code, specifications, and proof code together in one source file, while verification proceeds incrementally and gives real-time feedback of the current symbolic state or verification errors. Readers need not understand every detail of the example; our goal is to convey key aspects of C*'s syntax, user development workflow, and proof-programming interface.

We use the `reverse` function in Fig. 1 as a running example. It implements the classical in-place linked-list reversal algorithm: given a pointer `pt` to the head of a singly-linked list, it sets up an accumulator `ret` pointing to the head of the reversed prefix (initially empty), and then traverses the list in a loop. At each iteration, it reverses one link by prepending the current node to the reversed

[1] Upon submission time, we still have three lemmas that remain unproven. Section 5.2 will explain that they would not diminish C*'s practicality.

```
struct list { int head; struct list *tail; };

struct list *reverse(struct list *pt)
  PARAM(`l:(int)list`)
  REQUIRE(`sll pt l`)
  ENSURE(`sll __return (REVERSE l)`)
{
  struct list *ret = 0;
  PROOF fold_sll_null(`0i`);
  PROOF assert_by_rewrite(
    `l == REVERSE [] ++ l:(int)list`,
    ThmList(REVERSE_DEF, APPEND_DEF));
  while (pt)
    INV(`exists l1 l2 pt_v ret_v.
      fact(l == REVERSE l1 ++ l2) **
      data_at pt__addr Tptr pt_v ** data_at ret__addr Tptr ret_v **
      sll ret_v l1 ** sll pt_v l2`)
  {
    PROOF unfold_sll_not_null(`pt_v:int`);
    struct list *nxt = pt->tail;
    pt->tail = ret; ret = pt; pt = nxt;
    PROOF fold_sll_not_null(`pt_v:int`);
    PROOF assert_by_rewrite(
      `l == REVERSE (h :: l1) ++ t:(int)list`,
      ThmList(REVERSE_DEF, gsym_rule(APPEND_ASSOC), APPEND_DEF));
  }
  PROOF unfold_sll_null(`pt_v:int`);
  PROOF {
    term eq = `l1 == REVERSE l:(int)list`;
    assert_by_rewrite(eq, ThmList(APPEND_NIL,REVERSE_REVERSE));
    pure_rewrite1_st(assume_rule(eq));
  }
  return ret;
}
```

Fig. 1: Verified `reverse` (prelude omitted). Shaded lines are verification-related.

prefix, updates the accumulator pointer `ret`, and advances the current pointer `pt` to the next node. Upon loop exit, `ret` points to the fully reversed list.

For brevity, Fig. 1 shows only the main body of the original source file, omitting the prelude that imports verification headers, proves lemmas, and defines proof functions. We return to these omitted components later in this section.

2.1 Separation-Logic Specifications

To develop a verified program, one must first specify its behavior formally. For pointer-manipulating code such as `reverse`, *separation logic* [31,35] has proven effective in giving concise specifications and modular proofs based on the notion of ownership. The data structure manipulated by `reverse` is a `struct list` with two fields: `head` (the value of this node) and `tail` (a pointer to the next node). To represent both the ownership and the functional content of a well-formed linked list, we introduce a representation predicate `` `sll pt l` ``. It is defined in the prelude as a recursive function on the representation list `` `l` `` by two inductive clauses (the `GLOBAL` annotation marks this definition as top-level proof code):

```
GLOBAL thm sll_def = new_rec_definition(/* define a recursive function */
  list_recursion_thm,              /* using the list recursion theorem */
  `sll pt [] = fact(pt == 0i) &&
   sll pt (h :: t) = fact(~(pt == 0i)) ** (exists q.
     data_at (field_addr pt Tlist Fhead) Tint h **
     data_at (field_addr pt Tlist Ftail) Tptr q **
     sll q t)`);
```

Here is a quick guide to the specification syntax appearing in this definition: specification terms are written as literals quoted in backticks `` `...` ``; integer literals such as `` `0i` `` carry a suffix to distinguish them from natural numbers; `` `**` `` denotes separating conjunction; `` `fact` `` embeds a pure proposition with empty ownership; `` `data_at addr ctype v` `` is the usual "points-to" predicate that asserts ownership of a value `` `v` `` with C layout `` `ctype` `` at address `` `addr` ``; and `` `field_addr base <struct_name> <field_name>` `` computes the address of a field in a struct by adding an offset to the given base address.

Using the `` `sll` `` predicate, we specify the input-output relationship of `reverse` in the function contract at lines 4–6. The reserved logical variable `` `__return` `` denotes the return value of a function in its postcondition, and `` `REVERSE` `` is the logical reverse function imported from an existing list theory.

2.2 Symbolic Execution with In-Situ Proof

After the specification is given, the development of a C* program is driven by forward symbolic execution of C program statements, steered—when necessary—by in-situ proof code.

Forward Symbolic Execution Since the seminal work of Berdine et al. [5], separation logic has become a sound basis for computing the strongest postcondition of program statements by *forward symbolic execution*. C* adopts an existing symbolic-execution engine QCP [41] that follows the same methodology and computes a symbolic state for every program point. For example, at the beginning of the function body (line 7), users can see from our prototype IDE that the symbolic state is initialized to:

Symbolic State

```
data_at pt__addr Tptr pt__pre ** sll pt__pre l
```

Apart from the precondition, symbolic execution also creates a `` `data_at` `` predicate for `pt` to reflect that storage is allocated for the function parameter. Here, `` `pt__addr` `` is the address of the parameter `pt`, and `` `pt__pre` `` is its value at function entry. Similarly, after symbolic execution of line 8, another `` `data_at ret__addr Tptr 0i` `` predicate is created for the local variable `ret`. In this manner, the symbolic state provides a concise view of the flow-sensitive facts and ownership about in-scope program variables and the sub-heaps that the program works on at the current point.

In-situ Proof Code At times, symbolic execution may fail to step through a program statement, e.g., because the ownership required for safe execution is not immediately visible in the symbolic state. This is where the user steps in and provides guidance through *in-situ proof code* preceded by the `PROOF` annotation.

For example, at line 18, symbolic execution enters the loop body with a symbolic state that contains the loop invariant as well as the path condition saying that the current value of `pt` is not `NULL`:

Symbolic State

```
exists l1 l2 pt_v ret_v.
  fact (~(pt_v == 0i)) ** fact(l == REVERSE l1 ++ l2) **
  data_at pt__addr Tptr pt_v ** data_at ret__addr Tptr ret_v **
  sll ret_v l1 ** sll pt_v l2
```

If we were to symbolically step through the next program statement at line 20 without adding the intervening proof code, C* would report an error message:

Verification Error

```
"cannot derive the precondition of a memory access"
```

This is because the engine cannot find a conjunct in the symbolic state that matches the required ownership predicate to read the `tail` field of the struct pointer `pt`, i.e., `` `data_at (field_addr pt_v Tlist Ftail) Tptr ...` ``.

The required ownership is actually present in the symbolic state, but hidden inside the `` `sll pt_v l2` `` predicate. To reveal it, the user "unfolds" this predicate using in-situ proof code at line 19, which calls a user-defined proof function `unfold_sll_not_null` with the pointer value `` `pt_v` `` as its argument. We defer an explanation of its definition to Section 2.3.

IDE Support Because both implementation and proof code are written in C, they can benefit from the same integrated environment toolchain. For example, when the user types `PROOF unfold` (before typing the rest), a standard C language server (e.g., clangd) and IDE client (e.g., VS Code) offer a pop-up list of functions in scope with that name prefix (here, `unfold_sll_not_null` and `unfold_sll_null`) together with their type signatures, giving type-aware completion for proof-code development without additional efforts.

As hinted earlier in this section, in addition to standard C language-server features (auto-completion, go-to-definition, inline documentation, etc.), our prototype IDE (via a VS Code extension) provides real-time verification feedback by showing the symbolic state at the cursor position and reporting verification error messages in a side panel. The underlying C*-specific verification compilation and execution workflow is described in Section 3.

2.3 LCF-Style Proof Programming

C* obtains its expressive and extensible theorem-proving capability from a library [9] that exports a C interface of the HOL Light prover [24] in a trustworthy

way. It thus follows the *LCF approach* [17,18] to proof programming: logical objects, such as theorems, terms, and types, are abstract values of types `term`, `thm`, and `type`, respectively; users can compute theorems via arbitrary proof code without loss of trustworthiness. The library also implements common facilities for goal-oriented proof, natively in C, using an explicit *goal tree* data structure.

A C* program gains access to this proof library, as well as other core verifytime interfaces, by the special import directive `#cst_include "cstar.h"` in the prelude; we summarize the core interfaces it provides in Section 3.2. Before that, we walk through some representative proof code in the running example.

Proving Lemmas With this library in hand, we can prove lemmas in the prelude. Similar to the definition of `` `sll` `` in Section 2.1, these proofs are marked `GLOBAL` to make them visible to all subsequent proof code. We show an example goal-oriented proof of the unfolding lemma `unfold_sll_not_null_thm` in Fig. 2a. It initializes a goal tree with a single `root` node containing the desired theorem statement (lines 2–6), saying that if a pointer is not `NULL` and owns a linked list, then this ownership can be transformed into ownership of a `struct list` node and ownership of the tail list. Here, `` `==>` `` is the usual implication connective and `` `|--` `` denotes separation-logic entailment.

The proof consists of a series of tactic applications that grow the goal tree top-down (macros `FOCUS` and `APPLY` implicitly set and update the current goal node). It preprocesses the goal by introducing all universal quantifiers and adding the antecedent of the implication to the assumption list via `INTROS_TAC`. It proceeds by case analysis on the representation list `` `l` `` (line 7), producing two subgoals in `subs`. Each case uses the definitional clauses of `` `sll` `` for rewriting (lines 9 and 12). The `` `NIL` `` case immediately leads to contradictory assumptions and gets solved (line 10). The `` `CONS` `` case introduces the existential quantifiers in the antecedent and provides witnesses for the consequent (lines 13–14), discharges the required fact by rewriting (line 15), and cancels the remaining ownership predicates by auto-framing (line 16). After the two subgoals are all solved, we obtain the desired theorem by bottom-up validation using `GOAL_SOLVE(root)` (line 17). It binds the resulting theorem to a global variable `unfold_sll_not_null_thm`.

Defining Proof Functions Users can define custom proof functions to be called inside in-situ `PROOF` blocks. Such functions are typically *state-changing*: they follow a *fetch–derive–submit* pattern, i.e., they fetch the current symbolic state (via `get_symbolic_state`, a built-in symbolic-execution interface in the core verifytime), derive a separation-logic entailment theorem from it to a desired new state, and submit the resulting theorem to update the symbolic state (via `set_symbolic_state`, ditto).

The definition of `unfold_sll_not_null` (shown in Fig. 2b) illustrates this pattern. It first fetches the current symbolic state and stores it in `cur_st`. It then applies a proof rule `auto_spec_hent` to instantiate the universal quantifiers of the previously proved theorem `unfold_sll_not_null_thm` (statement shown in

```
GLOBAL thm unfold_sll_not_null_thm_proof() {
  GN root = GOAL_INIT(`forall pt l. ~(pt == 0i) ==> (
    sll pt l |-- exists h t q. fact(l == h :: t) **
    data_at (field_addr pt Tlist Fhead) Tint h **
    data_at (field_addr pt Tlist Ftail) Tptr q **
    sll q t)`);
  GN *subs = LIST_CASES_TAC(INTROS_TAC(root), `l:(int)list`);
  { FOCUS(subs[0]);
    APPLY(ASM_REWRITE_LIST_TAC, ThmList(sll_def));
    APPLY(HFALSE_ELIM_TAC); }
  { FOCUS(subs[1]);
    APPLY(ASM_REWRITE_LIST_TAC, ThmList(sll_def)); APPLY(HCLEAN_TAC);
    APPLY(HCHOOSE_TACS);
    APPLY(HEXISTS_LIST_TAC, TermList(`h:int`, `t:(int)list`, `q:int`));
    APPLY(ASM_REWRITE_LIST_TAC, ThmList()); APPLY(HCLEAN_TAC);
    APPLY(AUTO_FRAME_TAC); }
  return GOAL_SOLVE(root);
}
GLOBAL thm unfold_sll_not_null_thm = unfold_sll_not_null_thm_proof();
```

(a) A tactic proof of an unfolding lemma.

```
GLOBAL
void unfold_sll_not_null(term pt) {
  term cur_st = get_symbolic_state();
  thm unfold_thm = auto_spec_hent(
    cur_st, unfold_sll_not_null_thm,
    TermPairList(`pt:int`, pt));
  apply_hentail(unfold_thm);
}
```

(b) An `sll` unfolding function.

```
GLOBAL
void assert_by_rewrite(term goal, thm_list thl) {
  GN root = GOAL_NEW(get_facts(), goal);
  ASM_REWRITE_LIST_TAC(root, thl);
  thm th = GOAL_SOLVE(root);
  add_fact(th);
}
```

(c) An automated fact-deriving function.

Fig. 2: A lemma proof and two user-customized proof functions.

Fig. 2a): `pt` is instantiated to the function input argument `pt`, while `l` is instantiated by finding a symbolic conjunct in `cur_st` that matches the antecedent `sll pt ?l` (where `?l` is to be matched). Finally, `apply_hentail`, a convenient wrapper around `set_symbolic_state`, performs a local update on the current symbolic state using the specialized theorem `unfold_thm`: it checks that `fact(~(pt == 0i))` holds, replaces the found `sll pt ?l` predicate with its unfolded form, and introduces fresh existential quantifiers accordingly. Other fold/unfold helpers occurring in the example (e.g., `fold_sll_not_null`) are defined similarly.

As another example, Fig. 2c shows the automated fact-deriving function `assert_by_rewrite` (invoked at lines 10–12, 23–25, and 30 in the running example). It tries to prove a pure proposition by rewriting (via `ASM_REWRITE_LIST_TAC`) using the facts in the symbolic state as assumptions, along with a user-specified list of theorems as additional rewrite rules. These facts are fetched by `get_facts`, which internally calls `get_symbolic_state`. If the tactic successfully proves the goal, it then adds the derived proposition to the symbolic state as a fact via `add_fact`, which internally calls `set_symbolic_state`. For an example usage, consider the symbolic state just before the `PROOF` block at line 28:

Symbolic State

```
exists ret_v pt_v l1 l2.
  fact(l==REVERSE l1 ++ l2) ** fact(l2==[]) ** fact(pt_v==0i) **
  data_at pt__addr Tptr pt_v ** data_at ret__addr Tptr ret_v **
  sll ret_v l1
```

The call to `assert_by_rewrite` at line 30 proves the equation `` `l1==REVERSE l` `` (meaning that reversal is now complete) from assumptions gathered from the facts in the symbolic state, i.e., `` `l==REVERSE l1 ++ l2` ``, `` `l2==[]` ``, and `` `pt_v ==0i` ``, together with additional algebraic laws of `` `APPEND` `` and `` `REVERSE` ``. This derived fact is then used to rewrite the `` `sll ret_v l1` `` predicate into `` `sll ret_v (REVERSE l)` `` in the symbolic state (line 31), matching the postcondition of `reverse`. Note that the workhorse tactic `ASM_REWRITE_LIST_TAC` here can be replaced with other tactics or trusted automation procedures to meet domain-specific proof needs.

Extending Verifytime Proof Support Our proof-supporting verifytime is extensible: users can package their own proof facilities—e.g., extended proof functions or trusted automation procedures—as C libraries and reuse them across C* programs. Like the standard `cstar.h` header, these libraries can be linked into verifytime and expose their functionality through verification headers.

In fact, the lemma proof and the proof functions above already draw on separation-logic-specific tactics such as `AUTO_FRAME_TAC` and entailment conversions such as `auto_spec_hent`. These are built on top of the core proof interface and packaged as separation-logic proof libraries. The example gains access to the conversion library by `#cst_include "common/operational.h"` in the prelude. We discuss the extensions we have built and experimented with in Section 4.

Proof Pattern: Diff-based Invariant Construction Live Verification [20] proposes a *diff-based* alternative to writing loop invariants by hand: the user performs a sequence of proof steps that transform the symbolic state just before the loop head—typically already close to a valid inductive invariant—into the desired invariant. When the symbolic state is large, this approach can be more succinct and more robust to program changes than spelling out the invariant directly. Supporting this requires fine-grained control over the symbolic state, such as targeted rewriting, assumption management, and quantifier manipulation.

C* supports diff-based invariant construction through state-changing conversions in `common/operational.h`, including `intro_exists(var, tm)` (introduce an existential quantifier by abstracting over occurrences of `tm`), `rewrite_hconj_to(pat, tar, thl)` (rewrite a conjunct matching `pat` to `tar` justified by theorems in `thl`), and `add_fact`/`del_fact` (add or remove facts from the symbolic state). Once the symbolic state matches the desired invariant, users can install it with:

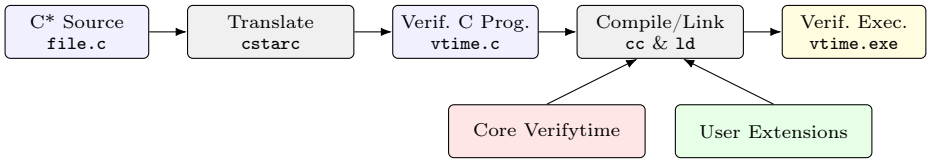

Fig. 3: C*'s compilation pipeline to produce a verification-time executable.

```
... // transform the symbolic state into the desired invariant
PROOF term loop_inv = get_symbolic_state();
while (condition)
  INV(loop_inv) // use the fetched symbolic state as the invariant
```

Note that `loop_inv` is a `term` computed at verification time, so specifications need not be fixed literals. Indeed, C* allows any C expression that constructs a separation-logic assertion to serve as a specification. A diff-based variant of `reverse` appears in our evaluation benchmark (cf. Section 5).

3 The Design of C*

This section presents the core architectural and language design of C*. The verification workflow of C* consists of two stages: (i) a *compilation pipeline* that takes a C* program and produces a verification executable, and (ii) the *verification-time execution* of the executable in C*'s verifytime environment. We detail these two stages in Sections 3.1 and 3.2, respectively, along with the related C*'s language interface for writing specifications and proof code.

3.1 Compilation Pipeline

To verify an annotated C* source file `file.c`, C* first translates it into a verification-time C program, `vtime.c`, and then compiles that generated program into a verifier executable, `vtime.exe`. This pipeline is shown in Fig. 3. The front end `cstarc` of C* is a C23 parser paired with a translator for C*-specific constructs. In this subsection, we first present key C*-specific constructs and then describe the translation process.

Specification Logic and Syntax The specification logic of C* is the higher-order logic of HOL Light [24] extended with a bespoke separation-logic theory for heap-dependent assertions of type `` `:hprop` ``. Our separation-logic theory is currently built on a concrete memory model: memory is treated as a word-addressed array of bytes, and pointers to memory objects are integer values that denote valid addresses, i.e., well-aligned and within the memory bounds.

Symbolic heaps. Assertions appearing in the specifications and the symbolic states submitted to the symbolic-execution engine (via `set_symbolic_state`) are represented in a separation-logic fragment known as *symbolic heaps*, i.e., a list of atomic predicates (built-in or user-defined) and facts, connected by separating conjunction, possibly under existential quantification. More generally, the symbolic state can be a disjunction of symbolic heaps, e.g., for a program point that merges multiple control-flow paths.

As in other symbolic-execution-based separation-logic verifiers [4,25,34], the purpose of this restriction is to ensure that the semantic effects of program statements can be abstracted as syntactic checks and updates on the relevant atomic conjuncts, matching programmers' intuition about how program state evolves locally. On the contrary, separation-logic assertions appearing in proof code need not be restricted to this fragment.

Quotation and anti-quotation. In C* programs, HOL terms and types can be written as literals quoted in backticks, e.g., `` `sll pt l` `` and `` `:int` ``. For convenience, inside a term quotation, anti-quotation of the form `` `...${x:ty}...` `` allows splicing a term of HOL type `` `:ty` `` stored in a program variable `x` of type `term` into the surrounding literal. This allows users to write specification templates that can be instantiated at verification time.

Support for C Features C* supports verification of implementation code in a substantial subset of standard C, inheriting the language coverage of the underlying QCP symbolic-execution engine [41]. For example, it supports integer types of various sizes and signedness, pointer types, structure types,[2] and `typedef` aliases; it covers expressions such as pointer arithmetic and field selection; it supports jump statements such as `break` and `continue` and iteration statements such as `for` and `while`. This language coverage subsumes the commonly used C features mentioned in Section 1. Notable omissions include support for floating-point types, function pointers, bit fields, compound literals, and `goto`.

Verification-related Annotations C* programs use a small set of annotations to distinguish specifications and proof code from ordinary implementation code. Technically, these annotations are implemented as macros that expand to C*-specific attributes in the namespace `cst`, e.g., `[[cst::require(...)]]` so the annotated source remains compatible with ordinary C tooling. They are rendered with a shaded background in the running example in Fig. 1. These verification-only constructs are erased for production builds.

Behavioral specifications. The annotations `PARAM`, `REQUIRE`, `ENSURE`, and `INV` give behavioral specifications to functions and loops. `PARAM(arg, ...)` takes a list of logical variables as ghost arguments that are universally quantified over the rest

[2] C* currently does not support passing structure values as function arguments or return values; passing them indirectly through pointers is supported.

of the specifications for this function. `REQUIRE(asrt)` and `ENSURE(asrt)` take a symbolic-heap assertion as argument, prescribing the pre- and postcondition of the function, respectively. Similarly, `INV(asrt)` attaches a symbolic-heap assertion as an invariant to an iteration statement such as `while` and `for` loops.

All these arguments may be quoted term literals or `term`-valued C expressions computed at verification time. The latter approach is a key enabler of proof patterns such as diff-based invariant generation (cf. Section 2.3) or the structural composition of specifications from simpler ones using term-manipulation functions. For example, we define a macro `ENSURE_EX(exvars, asrt1, asrt2, ...)` to support giving symbolic conjuncts individually: it takes a list of variables (as existential quantifiers) and a list of assertions (as symbolic conjuncts) and calls term construction functions to combine them into a symbolic-heap assertion.

In these specifications, some logical variables are treated specially by the symbolic-execution engine. In the function contract, logical variables with the same names as function parameters refer to their initial values. In the postcondition, the variable `` `__return` `` denotes the return value of the function. Inside the function body, logical variables of the form `` `var__pre` `` are used to refer to the initial value of the function parameter `var` at function entry; the form `` `var__addr` `` are used to refer to the address allocated for the function parameter `var` (the same naming convention is used for other addressable program variables, such as global variables).

Proof code. Proof code is preceded by the `GLOBAL` and `PROOF` annotations. Such code has access to the core verifytime interface (including its proof interface and the symbolic-execution interface) as well as other verification-time extensions (such as user proof libraries) by a special import directive `#cst_include`.

The `GLOBAL` annotation marks top-level proof code that is executed as a prelude before functions are verified. It takes any global declaration form, such as a function declaration or a variable definition. Typical use of these top-level proof code includes (we use the prelude of the `reverse` example to illustrate): importing theories from HOL Light (e.g., fetching the list recursion theorem), defining representation predicates (e.g., defining `` `sll pt l` `` using the list recursion theorem), registering logical symbols with the symbolic-execution engine (e.g., registering `` `sll: addr->(int)list->hprop` `` as a user-defined logical symbol), proving lemmas (e.g., the unfolding lemma `unfold_sll_not_null_thm`), and defining helper proof functions for use in local proof code (e.g., `assert_by_rewrite` and `unfold_sll_null`).

The `PROOF` annotation marks in-situ proof code inserted between ordinary program statements inside a function body. They are executed when symbolic execution reaches the corresponding program point. This in-situ proof code can be any statement, and respects the block scopes created by the surrounding implementation code. They are used to perform a reasoning step that symbolic execution cannot infer on its own. Reusable proof patterns can be abstracted as state-changing proof functions, which typically follow a *fetch–derive–submit* pattern (cf. Section 2.2).

```
void verify_reverse() {
  feed_program_segment(
    "struct list *reverse(struct list *pt)\n"
    "  PARAM(l:(int)list)\n"
    "  REQUIRE(sll pt l)\n"
    "  ENSURE(sll __return (REVERSE l))\n"
    "{\n");
  feed_program_segment(
    "  struct list *ret = 0;\n");
  fold_sll_null(parse_term("0i"));
  assert_by_rewrite(parse_term("l == REVERSE [] ++ l:(int)list"),
    ThmList(REVERSE_DEF, APPEND_DEF));
  feed_program_segment(
    "  while (pt)\n"
    "    INV(...) {\n",);
  { // block scope created for the proof code inside the loop body
    unfold_sll_not_null(parse_term("pt_v:int"));
    feed_program_segment(
    "    struct list *q = pt->tail;\n");
    /* ...remaining proof blocks and program fragments... */
  } /* ... */
}
```

Fig. 4: An excerpt of the verification driver generated for `reverse` (simplified). Shaded lines are program segments fed to the symbolic-execution engine. Unshaded lines are `PROOF` code passed through verbatim.

Front-end Translation The output `vtime.c` of the translation in Fig. 3 contains the verifytime startup code and the generated verification drivers for implementation code. The translation performs three main tasks:

- The translator replaces quoted literals inside backticks with calls to the proof library's term parser `parse_term`, and handles anti-quotation by splicing in the corresponding C `term`-variables through parallel substitution;
- Declarations marked by `GLOBAL` are emitted at top-level, and their initialization code is preserved in source order and executes once after verifytime startup code in the `main` function;
- Each implementation function is translated into its verification driver, which interleaves symbolic-execution calls with the user's local `PROOF` blocks.

Figure 4 shows a simplified excerpt of the driver generated for `reverse`. It feeds specifications and program segments in the function body as input to the incremental symbolic-execution engine via its main interface `feed_program_segment`, and emits intervening `PROOF` blocks as ordinary C code with the same block structure as in the source program.

A standard C compiler then compiles `vtime.c` and links it with the core verifytime libraries: the symbolic-execution engine [41] and the HOL proof library [9]. It may also link in user-specified extensions.

3.2 Verification-Time Execution

Running the resulting `vtime.exe` from the compilation pipeline (shown in Fig. 3) executes the generated drivers and performs verification of `file.c`. The gener-

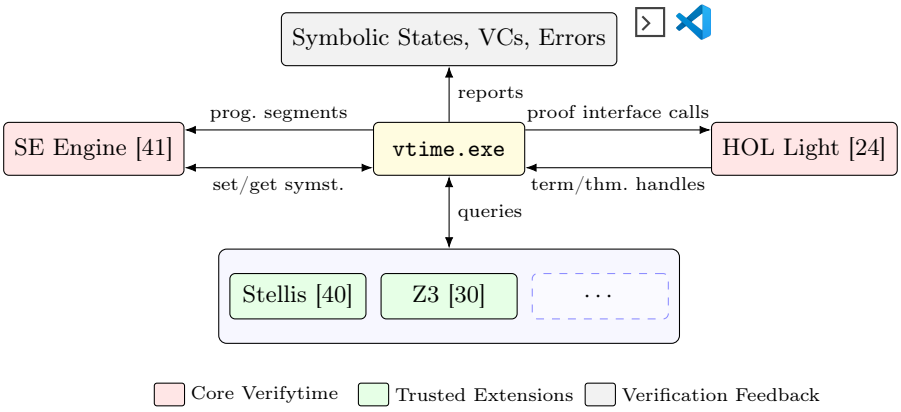

Fig. 5: A schematic illustration of the verification-time execution orchestrated by the generated verifier executable. Extended proof libraries (cf. Section 4.1) that do not extend the trusted base are not shown in this figure. The two trusted extensions are described in Section 4.2.

ated drivers act as the orchestrator between the core verifytime components and any linked extensions, as schematically shown in Fig. 5.

The driver performs several kinds of steps:

- *Program-reasoning steps* are delegated to the symbolic-execution engine by feeding a program segment to it. The engine then updates its internal verification state accordingly, including information about in-scope variables, exit points for jump statements, and the current symbolic state. It emits verification conditions (VCs) if it cannot discharge itself, such as overflow checks or entailments required to establish loop invariants. On an immediate failure (e.g., missing ownership for a memory access), the engine reports an error pinpointed to the corresponding source location.
- *Assertion-reasoning steps* are performed by executing user-provided `PROOF` blocks. The user's proof code queries the symbolic state, calls the proof interface or linked user extensions (such as an SMT oracle) to derive a theorem, and submits it back to the engine to perform a justified state update.
- *Reporting steps* give verification feedback to the user through the terminal output or IDE client, by printing the symbolic state at relevant source locations as well as the undischarged VCs and errors collected from the driver.

In the rest of this subsection, we describe the symbolic-execution and proof interface provided by the core verifytime of C*, and then discuss the trustworthiness of verification results.

Symbolic-execution Interface User proof code steers verification through a narrow interface to the symbolic-execution engine. At the core are two built-in verifytime functions for inspecting and updating the current symbolic state:

- `term get_symbolic_state()`: queries the symbolic-execution engine for the current symbolic state and returns it as a HOL term of type `` `:hprop` ``.
- `void set_symbolic_state(thm th)`: asks the symbolic-execution engine to update the current symbolic state using a theorem `th` that proves a separation-logic entailment `` `cur_st |-- des_st` ``, where `` `cur_st` `` and `` `des_st` `` are symbolic heaps. It checks that `` `cur_st` `` matches the current state (modulo alpha-renaming and the structural rules of separation logic) and then installs `` `des_st` `` as the new state.

The interface also includes registration functions `add_const_to_header` and `add_type_to_header`. They are used to provide basic information to the symbolic-execution engine about user-defined logical symbols—e.g., the type of a function and the arity of a type—so that they can appear in specifications and symbolic states. These registration functions are typically invoked in the prelude.

Proof Interface C* obtains its theorem-proving capability from a proof library [9] that exposes a comprehensive and trustworthy interface to the HOL Light prover [24]. This library extends the *LCF approach* of theorem proving [17,18] to C:

- For trustworthiness of theorem results, it exports logical objects as opaque handles of type `term`, `thm`, `type`, forbidding direct access to their internal representation in HOL Light.
- For forward reasoning and theory development, it provides, as C functions, a core set of HOL Light inference rules (both primitive and derived ones), conversions (e.g., bottom-up iterative rewriting), definitional principles (e.g., defining recursive functions and inductive datatypes), and term syntax utilities (e.g., constructors and destructors, pattern matching and substitution, parser and printer).
- For backward reasoning (i.e., goal-oriented proof), it uses a data structure of *goal trees* to store the intermediate goals and validation chains of an unfinished proof. General-purpose tactics used in HOL-family provers, such as assumption management, rewriting, and the introduction and elimination of logical connectives and quantifiers, are implemented natively in C as *goal-tree transformers*.

We refer readers to the proof library [9] for a comprehensive account of this interface. We describe the extended proof libraries we built for separation-logic reasoning in Section 4.1.

Remark 1 (C as Proof Language). C* uses C as a proof language, departing from both traditional LCF-style provers [18,24,38], which usually rely on functional meta-languages, and Rocq-style provers [6], which provide a dedicated language for tactic scripting. Although C is not an obvious fit for proof programming, this design choice has two practical advantages, as exemplified in Section 2:

- First, users can write implementations and proofs in the same language, simplifying development and lowering the learning barrier;

– Second, even though C lacks proper support for higher-order abstractions and exception handling, our experience with C* shows that lemma proofs and reusable proof libraries can still be developed with reasonable verbosity.

Trustworthiness of Verification Results A successful C* verification establishes its claim by composing program reasoning with assertion reasoning:

– *Program reasoning* is performed by the symbolic-execution engine, which is trusted to faithfully reflect the semantic effects of the C language constructs it processes against an underlying separation logic [41].
– *Assertion reasoning* is performed by user proof code through the proof interface; by the extended LCF discipline of our proof library [9], given that no additional axioms are added, every theorem value it produces is, by construction, derivable from the HOL Light kernel's primitive rules, no matter how complex the proof code is.
– *The two are glued together* by `set_symbolic_state`, which updates the symbolic state only when supplied with a separation-logic entailment theorem witnessing the transformation. Each such call is an in-place application of the program logic's consequence rule, and is the only channel through which user-provided proof code can influence the verification state.

The trusted base is therefore (i) the HOL Light proof kernel; (ii) the symbolic-execution engine [41]; (iii) the embedding that maps the engine's internal symbolic states into the HOL representation of separation-logic assertions; and (iv) any automation procedures treated as oracles, i.e., when their results are trusted directly and posed as axioms (cf. Section 4.2). Conspicuously *not* in the trusted base are user-provided proof code or extended proof libraries (cf. Section 4.1).

4 Verifytime Extensions

This section describes the verifytime extensions we built and used throughout our evaluation. Following the discussion at the end of Section 3.2, we organize them along the axis of trustedness: Section 4.1 describes separation-logic-specific proof libraries built on top of the core proof interface—every theorem they produce is *foundationally* checked by the HOL Light kernel, so they stay outside the trusted base regardless of their complexity; Section 4.2 describes automation procedures whose results are *trusted* and directly asserted as axioms, and therefore extend the trusted base for any verification that invokes them.

4.1 Separation-Logic Proof Libraries

The proof interface introduced in Section 3.2 gives access to general-purpose inference machinery and tactics, but it does not, on its own, provide high-level support for separation-logic reasoning. We close this gap with separation-logic proof libraries built on top of that interface and linked into the verifytime.

Tactics for Separation-logic Entailment The core proof library provides tactics for the introduction and elimination of usual HOL logical connectives and quantifiers, handling goals of form `` `asmps ?- concl` ``, i.e., a sequent where `` `asmps` `` is the list of assumptions and `` `concl` `` is the conclusion. We lift this support to the embedded separation-logic theory for specialized handling of goals of the form `` `asmps ?- (hp1 |-- hp2)` ``, where the conclusion is a separation-logic entailment.

We have introduction and elimination tactics for the separation-logic connectives. For a few examples used in the lemma proof in Fig. 2a: `HCHOOSE_TAC` extracts existentials from the antecedent; `HEXISTS_TAC` instantiates the consequent's existentials with user-supplied witnesses; `HFALSE_ELIM_TAC` closes any goal whose antecedent contains `` `fact(false)` ``.

We also combine these basic tactics into more advanced ones that automatically apply the structural rules of separation logic. To list a few examples: `FACTS_INTRO_TAC` moves `` `fact(p)` `` conjuncts in the antecedent into the goal's assumptions; `FACTS_SPLIT_TAC` dually splits each `` `fact(p)` `` in the consequent into a separate pure subgoal. `HCLEAN_TAC` erases vacuous conjuncts such as `` `fact(true)` ``; `AUTO_FRAME_TAC` cancels syntactically identical conjuncts on both sides of the entailment, leaving the residual frame as the new goal (it solves the goal if there are no remaining conjuncts on both sides).

Overall, these separation-logic tactics are broadly similar to those provided by existing separation-logic tactic libraries such as VST-Floyd [8] and IPM [28], but they are implemented in C and are currently less automated than their counterparts in these more mature frameworks. We include an example proof in Appendix A to illustrate the use of these tactics.

Conversions for Separation-logic Entailment A *conversion*, in the LCF jargon [32], is a function from a term to an equality theorem that rewrites the term to a provably equivalent term, e.g., to perform simplifications. Here, we adopt the name *entailment conversion* for any function that takes separation-logic assertions as input and returns an entailment theorem that has the input assertion as its antecedent. These entailment conversions are the forward-reasoning counterpart to the tactics above and the workhorse behind state-changing proof functions: they allow us to soundly transform an assertion into a new one *operationally*, i.e., without explicitly spelling out the resulting consequent.

The most commonly used entailment conversion is `auto_apply`, which transforms an input assertion "locally" by applying an entailment theorem if each conjunct in the theorem's antecedent has a matching conjunct in the input assertion. In fact, the state-changing `apply_hentail` function used in Fig. 2b is just a thin wrapper around `auto_apply` that calls `set_symbolic_state` to submit the entailment. On top of this primitive and utilities for pattern matching, the library offers state operators that together form the diff-based invariant construction vocabulary mentioned in Section 2.3. Moreover, since separation-logic entailment is a reflexive relation, every equality conversion (such as the rewriting conversion) can be turned into an entailment conversion.

4.2 Trusted Proof Automation Procedures

When foundational reasoning becomes too verbose for a recurring class of goals, the verifytime can be extended by employing more aggressive *trusted automation procedures* that serve as oracles. Unlike the libraries of Section 4.1, these procedures bypass the HOL Light kernel: their verified results are directly asserted as axioms via `new_axiom`. We currently incorporate two such procedures wrapped into tactics, designed to compose: `PURIFY_TAC` performs spatial reasoning to reduce a separation-logic entailment goal to a pure entailment by rule-based automation, and `SMT_TAC` discharges a pure goal via SMT solving.

For concreteness of exposition, we use throughout this subsection a representative VC arising in the verification of a binary heap implementation (available in our benchmark suite as `binary_heap.c`). Its specification follows Tafat et al. [39], with the central functional invariant being the min-heap property (auxiliary integer-bound side conditions omitted):

$$\mathsf{MH}(a, m) \equiv \forall k.\ 0 < k < m \Rightarrow a\left[\left\lfloor \tfrac{k-1}{2} \right\rfloor\right] \leq a[k].$$

Rule-based Automation of Spatial Reasoning We integrate Stellis [40], a recent rule-based automation framework for separation logic, as a trusted verifytime extension. Stellis takes a separation-logic entailment and tries to repeatedly apply user-supplied entailment lemmas in an actionable format (called *strategies*), until the entailment is *purified*, i.e., with no spatial conjuncts. We provide a verifytime interface `add_strategy_to_header` for loading strategy files and a tactic `PURIFY_TAC` that invokes Stellis on the current goal using all loaded strategies. We refer readers to Stellis [40] for the detailed syntax and soundness justification of its strategies.

We illustrate `PURIFY_TAC` on the VC arising in the sift-up loop of `heap_insert`, at the point where the parent element has been copied down to the current index `i` and the loop invariant must be re-established for the next iteration (simplified for brevity):

```
`fact(e <= array_select a i_v) ** fact(e < array_select a ((i_v - 1i) / 2i)) **
 fact(MH a (n + 1i)) ** fact(0 < i_v) ** fact(i_v <= n) **
 data_at e__addr Tint e ** store_array_hole h__addr n i_v a **
 data_at n__addr Tint n ** data_at i__addr Tint ((i_v - 1i) / 2i) **
 data_at (h__addr + i_v * sizeof Tint) Tint (array_select a ((i_v - 1i) / 2i))
 |-- exists a' i_v'.
    fact(MH a' (n + 1i)) **
    data_at n__addr Tint n ** data_at i__addr Tint i_v' **
    data_at e__addr Tint e ** store_array h__addr n a'`
```

Here, the predicate `` `store_array h n a` `` asserts ownership of an `int` array at base address `` `h` `` with length `` `n` `` and contents given by a functional array `` `a` ``; `` `store_array_hole h n i a` `` asserts the same but with the cell at index `` `i` `` removed. The operations `` `array_select` `` and `` `array_store` `` denote array read and update: the former reads the value at a given index, and the latter returns the array updated at that index with a given value. Driven by strategies, `PURIFY_TAC` can eliminate spatial conjuncts in the loop invariant, witness `` `a'` `` with an updated array `` `array_store a i_v (array_select a ((i_v-1i) / 2i))` `` and `` `i_v'` `` with the new index `` `(i_v-1i) / 2i` ``, and leaves the residual pure subgoal:

```
ast term_to_z3(context ctx, term tm) {
  if (is_imp(tm)) {
    res = dest_imp(tm);
    return Z3_mk_implies(ctx,
      term_to_z3(ctx, res.tm1),
      term_to_z3(ctx, res.tm2));
  } else if (is_forall(tm)) {
    res = dest_forall(tm);
    ...
    return Z3_mk_forall_const(...);
  }
  ...
  else return term_to_z3_ex(ctx, tm);
}
```

(a) Built-in translation core.

```
ast term_to_z3_ex(context ctx, term tm) {
#ifdef SMT_ARRAY
  Z3_ast res = array_term_to_z3(ctx, tm);
  if (res != NULL) return res;
#endif
  ...
  return NULL; }
ast array_term_to_z3(context ctx, term tm) {
  res = strip_comb(tm);
  if (!strcmp(name(res), "array_select"))
    return Z3_mk_select(ctx,
      term_to_z3(ctx, res.tms[0]),
      term_to_z3(ctx, res.tms[1]));
... }
```

(b) Theory-specific translation hook.

Fig. 6: Fragments of the SMT translation.

```
`MH a (n + 1i) && 0 < i_v && i_v <= n &&
 e < array_select a ((i_v - 1i) / 2i) && e <= array_select a i_v
 ==> MH (array_store a i_v (array_select a ((i_v - 1i) / 2i))) (n + 1i)`
```

SMT-based Automation of Pure Reasoning To automate the class of pure reasoning exemplified by the residual VC above, we implement `SMT_TAC`, a trusted tactic that translates a pure entailment goal into an SMT query and delegates it to Z3 [30]. The architecture of the underlying translator, shown in Fig. 6, separates a built-in core from theory-specific translations. The core handles the logical connectives, equality, the quantifiers, and integer arithmetic (Fig. 6a); operations outside this fragment are forwarded to registered extension hooks (Fig. 6b). For the binary heap example, the logical array type is mapped to a Z3 array sort, and array operations `` `array_select` ``, `` `array_store` `` are interpreted as Z3's `` `select` ``, `` `store` `` operations on arrays, respectively, in a trusted manner.

Composing the Automation Procedures The two oracles can be *composed* for full automation: `PURIFY_TAC` eliminates the spatial conjuncts by Stellis, and `SMT_TAC` dispatches the resulting pure obligation to Z3. In our binary heap example, we can package this composition into a state-changing proof function `heap_auto(target)` that sets up a goal as the entailment from the current symbolic state to the supplied target, applies `PURIFY_TAC` followed by `SMT_TAC`, and submits the resulting entailment theorem to the symbolic-execution engine via `set_symbolic_state`, which updates the symbolic state to `target`. Powered by this composition, most proof steps in the binary heap verification are performed declaratively by a single `heap_auto` call.

5 Evaluation

We implemented a prototype for our C* verifier, including (i) the compilation pipeline described in Section 3.1, (ii) the core verifytime environment described

Table 1: Benchmark statistics. `#Fun` is the number of implementation functions verified; `#Impl`, `#Spec`, and `#Proof` are lines of implementation code, specifications (function contracts and loop invariants), and in-situ `PROOF` code, respectively; `Extensions` records which trusted-automation procedures from Section 4.2 the benchmark relies on; `Time` reports the running times of verification-time executables in seconds.

Class	File	#Fun	#Impl	#Spec	#Proof	Extensions	Time(s)
basic	feature.c	6	38	48	0	–	0.33
	arith.c	12	101	40	74	SMT	4.29
array	traversal.c	3	32	13	39	Stellis	6.29
	binary_heap.c	3	53	16	89	Stellis+SMT	24.20
	dsu.c	3	31	23	84	Stellis+SMT	20.24
pointer	sll.c	5	61	18	69	–	7.94
	sll_reverse.c	1	10	7	15	–	1.35
	sll_reverse_diff.c	1	10	4	27	–	1.98
	dll.c	2	35	15	0	Stellis	0.22
	bst.c	8	129	32	164	–	76.87
hyp_allocator	psi_capsule.c	13	62	56	55	SMT	14.86
	list.c	18	90	113	82	SMT	33.83
	allocator.c	46	500	497	522	Stellis+SMT	427.39

in Section 3.2, (iii) the verifytime extensions we developed during evaluation described in Section 4, and (iv) a VS Code extension together with a verification-focused language server. The language server invokes the C* verification workflow to provide feedback on verification conditions, symbolic states, and intermediate proof states (e.g., the contents of `term` and `thm` values). During our evaluation, we used VS Code with our extension as the C* IDE for co-developing programs and proofs. The runtime statistics were collected on a machine running Ubuntu 24.04 (WSL2) with a Core i7-12700H CPU and 8 GB of RAM.

5.1 Standard Examples

We assembled a suite of benchmarks to exercise the C* verifier on a range of programming idioms and proof patterns.[3] Table 1 summarizes each benchmark together with its development scale. Note that the `#Proof` column counts only in-situ `PROOF` blocks; `GLOBAL` proof code, such as lemma proofs and per-program proof-function customizations, as well as the reusable proof libraries of Section 4.1, are excluded but reported in the Appendix (cf. Table 2), where shared header files are also listed. While the `GLOBAL` proof code currently requires more effort than the in-situ `PROOF` code in most benchmarks, we emphasize the in-situ `PROOF` code to qualitatively evaluate C*'s capability for live verification, i.e., support of C features and program-proof co-development.

The benchmarks are grouped into four classes.

[3] All the benchmarks can be found in the submission's supplementary material.

Feature tests and arithmetic functions (`basic`). File `feature.c` consists of functions that test C features such as `for`-loops, `break`/`continue` jumps, taking the address of local variables, struct field access, and nested function calls. File `arith.c`[4] contains arithmetic functions such as bitwise shifts, Euclidean division, GCD with Bézout coefficients, and fast exponentiation. Their proofs are carried out either by manually rewriting against the relevant lemmas in nonlinear arithmetic or by delegating the resulting VCs to the SMT-based automation described in Section 4.2.

Array-based data structures (`array`). File `traversal.c` verifies summation, zeroing, and linear-search traversals over `char` buffers. File `binary_heap.c` verifies a fixed-capacity binary heap (`heap_init`, `heap_insert`, `heap_extract_min`); its specification follows Tafat et al. [39]. As explained in Section 4.2, it adopts a declarative proof style in which VCs are discharged by the rule-based and SMT-based automation procedures, packaged behind the user-defined helper `heap_auto`. File `dsu.c` verifies a disjoint-set union data structure (`create`, `find`, `merge`) under a similar automation setting.

Pointer-based data structures (`pointer`). File `sll.c` verifies functional correctness of singly-linked-list operations, including in-place `append` and `stack` operation; File `bst.c` verifies iterative and recursive variants of binary search tree operations such as `lookup`/`insert`, as well as its rotation operations. Verification in this class centers on managing user-defined ownership predicates (e.g., `sll`, `tree_repr`) through unfold/fold helpers derived from lemmas (cf. Section 2.3). File `dll.c` verifies shape-preservation properties of doubly-linked-list operations; here we use a form of strategy application hooked into the frame-inference process of the symbolic-execution engine (not described in this paper), which yields fully automated proofs with no manual `PROOF` code (hence `#Proof = 0`).

The `sll_reverse.c` benchmark is the running example of Section 2. Its variant `sll_reverse_diff.c` constructs the loop invariant via the diff-based workflow in Section 2.3, using the operational entailment conversions in Section 4.1. Reading across the two rows highlights the trade-off between specifying the invariant up front and constructing it from the previous symbolic state.

Case study (`hyp_allocator`). We redevelop the pKVM hypervisor's allocation algorithm as a case study for program-proof co-development. The case study is split across three files: `psi_capsule.c` contains the *Ψ-capsules* (which will be discussed in Section 5.2), `list.c` contains the intrusive-list helpers, and `allocator.c` contains the allocator operations themselves.

5.2 Case Study: pKVM `hyp_allocator`

This case study evaluates C* on the Android pKVM hypervisor allocator,[5] a compact but realistic systems component with low-level idioms such as intru-

[4] The `arith.c` aggregates three files `arith.c`, `arith_overflow.c`, `arith_smt.c` from the benchmark suite for brevity.

[5] The original source is available at `arch/arm64/kvm/hyp/nvhe/alloc.c`.

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

struct hyp_allocator {                struct chunk_hdr {
  struct list_head  chunks;             unsigned int      alloc_size;
  unsigned long     start;              unsigned int      mapped_size;
  unsigned int      size;               struct list_head  node;
} hyp_allocator;                        unsigned int      hash; };
```

Fig. 7: The main allocator structure definitions.

sive lists, pointer arithmetic between headers and payloads, chunk splitting and coalescing, integrity hashes, and on-demand page mapping. Our goal is *not* to present a complete end-to-end proof of the production pKVM allocator. Rather, we retrospectively redevelop a sequential version of the allocator as a test for C*'s intended live-verification style in a practical setting: implementation, specifications, and proof code are developed together; a readable symbolic state is maintained at all program points; and recurring program-proof fragments are packaged into reusable units.

Allocator Model and Verification Scope The main structures of the global allocator state are shown in Fig. 7. The allocator `hyp_allocator` manages a contiguous, page-aligned virtual address range. A block of memory is represented by a *chunk*; each chunk starts with a `struct chunk_hdr` header containing the allocated size, the physically mapped size, list links, and a hash value, followed by the client-visible payload. The allocator keeps all chunks in an intrusive circular doubly linked list, sorted by address. Allocation searches for a best-fitting free chunk, maps additional pages if necessary, splits the selected chunk when the remainder is large enough, and returns a zeroed payload pointer. Freeing marks the chunk as free and then tries to coalesce it with adjacent free chunks.

Our re-development covers the main allocator user interfaces `hyp_alloc` and `hyp_free`, as well as their related internal functions, e.g., chunk list helpers. The interface specifications and internal invariants express both memory-shape invariants, such as list consistency and chunk ownership, and stronger functional properties, such as its best-fitting allocation strategy and the preservation of the abstract allocation state across allocation and freeing.

Our implementation mostly follows the original algorithm but makes several adaptations for verification: (i) it assumes a sequential setting and ignores lock operations; (ii) it replaces macros such as `container_of` with helper functions with explicit specifications; (iii) it duplicates functions with different input scenarios (e.g., when the chunk list is empty or not) into separate versions; and (iv) it encapsulates recurring program-proof patterns into reusable functions with specifications at different abstraction levels.

We make some explicit assumptions about the environment: unmapped virtual memory regions are logically represented by abstract predicates with ax-

```
`list_hd (la:int) (nxt:int) (prv:int) : hprop =
  data_at (field_addr la Tlist_head Fnext) Tptr nxt **
  data_at (field_addr la Tlist_head Fprev) Tptr prv`

`chunk_hdr ca (a, m, nxt, prv) hf =
  data_at (field_addr ca Tchunk_hdr Falloc_size) Tuint a **
  data_at (field_addr ca Tchunk_hdr Fmapped_size) Tuint m **
  list_hd (node ca) nxt prv **
  hash_at (field_addr ca Tchunk_hdr Fhash) (a, m, nxt, prv) hf **
  fact(~(ca == 0i)) ** fact(valid_ca ca) ** fact(valid_size a m)`

`chunk_hdr_with_unmap ca (a, m, nxt, prv) hf ha end =
  chunk_hdr ca (a, m, nxt, prv) hf **
  mapped (ca + hdr_size + a) (m - (hdr_size + a)) **
  unmapped (ca + m) (hdr_or nxt ha end - (ca + m))`

`cst_hyp_allocator ha start end l NXT PRV =
  start_end ha start end **
  list_hd (chk ha) (NXT (chk ha)) (PRV (chk ha)) **
  chunk_hdrs l NXT PRV ha end **
  fact(dll (chk ha) (chk ha) (map_fst l) NXT PRV) **
  fact(valid_dll (chk ha) (map_fst l))`
```

Fig. 8: Examples from the predicate hierarchy used in the allocator development, from low-level to high-level: the primitive field ownership predicate `data_at`, the list-node ownership predicate `list_hd`, the chunk-level predicate `chunk_hdr` (and its variant `chunk_hdr_with_unmap` that also owns the mapped and unmapped regions), and the global allocator state predicate `cst_hyp_allocator`.

iomatized laws; external services, such as hash computation and page mapping/unmapping, have trusted specifications without verified implementations.

Layered Specifications The allocator specifications are organized around a hierarchy of separation-logic predicates, examples of which are shown in Fig. 8. At the bottom, primitive predicates such as `data_at`, `mapped`, `unmapped`, and `zeroed` describe fields and contiguous memory regions. The list layer packages the two links of a `struct list_head` into `list_hd`. The chunk layer combines header fields, the embedded list node, hash state, and optionally the mapped/unmapped layout around a chunk. The allocator layer owns the global allocator record, the sentinel list node, the whole chunk list, and pure facts stating that the list is a well-formed circular doubly linked list over valid chunk addresses.

This hierarchy lets each function (either an internal helper or a user-facing interface) have a specification at the abstraction level that matches programmers' expectations and the client code's calling context. For example, a list-manipulation helper only mentions the list-node ownership predicate `list_hd`; while a user-facing allocator operation such as `hyp_alloc` is specified using the top-most `cst_hyp_allocator` predicate. This improves the readability and maintainability of the specifications compared to exposing a large conjunction of primitive ownership predicates.

```
void set_chunk_hdr_alloc_size(struct chunk_hdr *chunk, unsigned int alloc_size)
  PARAM(`a:int`, `m:int`, `nxt:int`, `prv:int`, `hf:bool`)
  REQUIRE(`chunk_hdr chunk (a, m, nxt, prv) hf`,
          `fact(valid_size(alloc_size, m))`)
  ENSURE(`chunk_hdr chunk (alloc_size, m, nxt, prv) false`)
{
  PROOF term _chunk = `chunk__pre:int`;
  PROOF unfold_chunk_hdr(_chunk);
  chunk->alloc_size = alloc_size;
  PROOF set_hash_at_false(_chunk);
  PROOF fold_chunk_hdr(_chunk);
}
```

Fig. 9: A Ψ-capsule for updating the `alloc_size` field. The implementation is one assignment; the specification stays at the chunk layer; the proof performs the local unfolding, update, hash invalidation, and refolding steps.

Proof-Spec-Impl Capsules A recurring pattern in the allocator is that a small C fragment exhibits fixed symbolic-execution behavior but can be used across different symbolic states (i.e., abstract views of the heap). For example, assigning to `chunk->alloc_size` requires an explicit `data_at` predicate of the `alloc_size` field at the instant of the write. However, call sites often hold a higher-level predicate such as `chunk_hdr` or `chunk_hdr_with_unmap`. The proof must unfold the current view, expose the field, perform the write, update associated invariants such as hash validity, and fold the desired view back. Repeating these proof steps at every call site would make the source hard to read and duplicate proof effort.

We therefore use an idiom we call *Proof-Spec-Impl capsules*, or *Ψ-capsules*. A Ψ-capsule binds three artifacts that should evolve together during program-proof co-development into a function: a reusable C implementation fragment, the specification at which clients use that fragment, and the proof code that justifies the implementation against that specification. The same underlying programming action can be exposed through capsules at different layers: one capsule may update a field under `chunk_hdr`, while another may perform the same update but under `chunk_hdr_with_unmap`. The caller chooses the capsule whose contract matches the current symbolic state, rather than manually replaying the unfold-modify-fold sequence.

Figure 9 shows an example of a Ψ-capsule for updating the `alloc_size` field. The caller sees a concise chunk-level transformation: a valid `chunk_hdr` is transformed into another `chunk_hdr` with updated `alloc_size`, with the hash-validity flag set to `false` because the stored hash no longer matches the header fields. The proof code contains the steps needed to make the assignment legal for symbolic execution. This improves specification and symbolic-state readability because call sites refer to chunks rather than primitive field resources. It also reduces the effort required for repeated proofs and may improve verification performance. If the representation of `chunk_hdr` changes, the capsule proof is the local place where the new unfolding and refolding steps are maintained.

```
void hyp_free(void *addr)
  PARAM(`start:int`, `end:int`, `l:(int#int#int)list`,
        `NXT:int->int`, `PRV:int->int`, `sz:int`)
  REQUIRE(
    `cst_hyp_allocator hyp_allocator__addr start end l NXT PRV`,
    `fact(malloc_token addr sz l)`, `mapped addr sz`)
  ENSURE(.../* allocator restored with the chunk marked free */)
{
  PROOF term _addr = `addr__pre:int`;
  PROOF term _ca = `addr__pre - hdr_size:int`;
  PROOF unfold_cst_hyp_allocator(`hyp_allocator__addr:int`);
  PROOF intro_split_at_malloc_ptr(_addr, `l1`, `l2`, `m`);
  PROOF split_chunk_hdrs_at(`l`, node(_ca));
  PROOF unfold_chunk_hdr_with_unmap_full(_ca);

  char *chunk_data = (char *)addr;
  struct chunk_hdr *chunk =
      chunk_get__nonzero(chunk_hdr_of_data(chunk_data));
  set_chunk_hdr_alloc_size(chunk, 0);
  chunk_hash_update__nonzero(chunk);
  ... /* merge with adjacent free chunks */
}
```

Fig. 10: Abridged excerpt of the verified `hyp_free`. The proof focuses global allocator ownership down to the chunk being freed, after which capsules handle the concrete field updates.

Example: Freeing a Chunk The beginning of `hyp_free` illustrates how layered predicates and Ψ-capsules work together, as shown in Fig. 10. At the allocator interface, the specification owns the entire allocator via `cst_hyp_allocator`, owns the payload region being freed via `mapped addr sz`, and provides a pure `malloc_token` indicating that the argument is the payload address of an allocated chunk. The specification says more than memory safety; it also justifies the *functional correctness* of `hyp_allocator`, in that the chunk to be freed must have been allocated previously. The implementation immediately needs the corresponding header so that it can mark the chunk free, update the hash, and later coalesce it with adjacent free chunks. The proof, therefore, descends from allocator ownership to the target chunk before executing the ordinary C updates.

The first proof command opens the top-level allocator predicate and exposes the recursive ownership of the chunk list. The next commands use the allocation token to split the list at the chunk being freed, then unfold the selected chunk to the layer needed by the update capsules. After this focusing step, the implementation fragment invokes capsules for the concrete field updates, which perform the same steps as the original allocator but also encapsulate the proof steps to maintain a chunk-level view of the symbolic state: compute the header pointer, set `alloc_size` to zero (to mark it is free), and recompute the hash. It then continues with coalescing (not shown in the excerpt).

The case study demonstrates a specific style of C* development: layered predicates give stable, readable names to the allocator states that programmers already reason about informally, and Ψ-capsules attach in-situ proof code to reusable implementation fragments useful across different specification layers.

Remark 2. As mentioned in Section 1, three lemmas remain unproven for the `hyp_allocator` case study at the time of submission. Appendix B details the three lemmas: `FIND_BEST_SNOC`, `DLL_INSERT_INV`, and `DLL_DELETE_INV`. These are all *pure* entailments, i.e., logical implications that are free of separation-logic connectives. We have manually checked their validity; it would require only some extra effort to prove them in HOL. To this end, we believe our case study demonstrates the practicality of C* for verifying realistic C programs.

6 Related Work

We draw inspiration from two long-standing lines of work on verifying C programs: (i) verification frameworks embedded in interactive proof assistants, which offer expressive proof support; and (ii) assertion-based verifiers, which offer a program-centered workflow. We compare with representative systems below.

Live Verification. The Live Verification framework of Gruetter et al. [20] is the most direct inspiration for C*. Its design and implementation rely on several Rocq mechanisms: custom notations that interpret C snippets as Bedrock2 [14] AST constructors and special comment tokens as symbolic-execution tactics; meta-existentials that represent a program under construction; and Rocq's goal panel, which displays the current symbolic state at the cursor position. C* adopts the live-verification idea but shifts the development environment from a proof assistant to plain C, and lifts the language-coverage restriction by being realized as a standalone tool that parses C23-syntax programs and delegates program-logic reasoning to a separately developed symbolic-execution engine QCP [41].

VST and VST-A. VST [1,8] is a foundational verification framework built on top of the CompCert C semantics, and has been applied to verify substantial real-world systems software [2,29,43,44]. The original VST workflow is to translate a C program into its Clight deep embedding, then specify its state and prove its correctness in Rocq. Its recent derivative VST-A [45] brings the development environment closer to C: users write specifications and intermediate assertions as annotations in the C source, and the tool compiles each annotated function into a proof template for the user to fill in. Compared to VST-A, C* takes a further step along this trajectory: in addition to specifications and intermediate assertions, C* also lets users write proof code in situ.

RefinedC. RefinedC [37] is a foundational (on top of the Caesium C semantics) and highly automated verifier for C. Users write specifications as refined-ownership types, which are semantically interpreted as Iris separation-logic predicates. Symbolic-execution-based type checking is then carried out automatically by Lithium, a syntax-directed proof-search engine for a fragment of separation logic; users can register additional domain-specific typing rules (proved sound as entailment lemmas) so that automation continues to apply to bespoke predicates and primitives. Compared to RefinedC, C* gives the user full control over

the proof process and allows the injection of in-situ proof code at precise program points. To handle recurring proof patterns, we use the idiom of Ψ-capsules (Section 5.2), which bind a code fragment to the proof steps needed to use it at a given specification layer. This is reminiscent of RefinedC's domain-specific typing rules, but unlike RefinedC, which uses syntax- and type-directed dispatch by the verifier, C* currently requires users to manually select the capsule whose contract matches the current symbolic state. We leave the automatic selection of capsules based on the symbolic state at the call site—for example, via IDE-level suggestion—as future work.

CN and VeriFast. CN [34] and VeriFast [25] are two state-of-the-art separation-logic-based, SMT-automated verifiers for real-world C programs. To make automation predictable, both restrict the assertion logic: predicate definitions should follow an input/output mode discipline that guarantees deterministic frame inference (CN enforces this via a syntactic-scope-based approach), and the use of quantifiers is constrained so that the resulting verification conditions remain within decidable SMT fragments. For proof support, both expose a fixed set of proof hints as ghost code—predicate unfold/fold, case analysis, and defining and invoking lemma functions—that users can insert when the built-in automation is insufficient. CN additionally allows stating lemmas that lie outside its SMT theory (e.g., non-linear arithmetic) to be exported to Rocq and proven there. Compared to CN and VeriFast, C* offers a more extensible and trustworthy proof interface. For example, local lemma application (as well as its pattern-matching-based quantifier instantiation strategy) baked into the implementation of these tools are realized in C* by ordinary user-level proof functions (e.g., `auto_spec_hent` and `apply_hentail`, cf. Fig. 2b), whose outputs are checked by the HOL Light kernel and therefore do not enlarge the trusted base. On the other hand, our current SMT integration and verification automation (e.g., for array reasoning) are much less mature than theirs.

7 Conclusion

We presented C*, a verifier that brings live program-proof co-development into the C language. Moving beyond the prover-centric syntax embedding of prior work, C* adopts a program-centric design: developers write implementation, separation-logic specifications, and in-situ proof code together in a single source file. C*'s verifytime environment integrates a symbolic-execution engine that handles forward program reasoning, as well as an LCF-style proof interface that provides trustworthy assertion-logic reasoning. Our evaluation on standard benchmarks and a re-developed version of pKVM's `hyp_allocator` demonstrates that live verification scales to realistic C code; the case study highlights reusable Ψ-capsules that bundle implementation, specification, and proof.

Future work will focus on programmable automation to reduce proof effort without enlarging the trusted base, as well as support for concurrency and richer C features. An IDE that suggests appropriate Ψ-capsules and interface functions based on the current verification state would further improve the experience.

References

1. Appel, A.W.: Program Logics - for Certified Compilers. Cambridge University Press (2014). https://doi.org/10.1017/CBO9781107256552
2. Appel, A.W.: Verification of a cryptographic primitive: SHA-256. ACM Trans. Program. Lang. Syst. **37**(2), 7:1–7:31 (2015). https://doi.org/10.1145/2701415
3. Appel, A.W.: Coq's vibrant ecosystem for verification engineering (invited talk). In: Popescu, A., Zdancewic, S. (eds.) CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022. pp. 2–11. ACM (2022). https://doi.org/10.1145/3497775.3503951
4. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures. Lecture Notes in Computer Science, vol. 4111, pp. 115–137. Springer (2005). https://doi.org/10.1007/11804192_6
5. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Yi, K. (ed.) Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3780, pp. 52–68. Springer (2005). https://doi.org/10.1007/11575467_5
6. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004). https://doi.org/10.1007/978-3-662-07964-5
7. Blanchard, A., Loulergue, F., Kosmatov, N.: Towards full proof automation in Frama-C using auto-active verification. In: Badger, J.M., Rozier, K.Y. (eds.) NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11460, pp. 88–105. Springer (2019). https://doi.org/10.1007/978-3-030-20652-9_6
8. Cao, Q., Beringer, L., Gruetter, S., Dodds, J., Appel, A.W.: VST-Floyd: A separation logic tool to verify correctness of C programs. J. Autom. Reason. **61**(1-4), 367–422 (2018). https://doi.org/10.1007/S10817-018-9457-5
9. Cao, Y., Zhuang, J., Fan, J., Wang, D., Hu, Z.: A HOL theorem proving interface in C. In: Proceedings of the 20th International Symposium on Theoretical Aspects of Software Engineering, TASE 2026, Shanghai, China, July 4-6, 2026 (2026)
10. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) Proceedings of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011. pp. 418–430. ACM (2011). https://doi.org/10.1145/2034773.2034828
11. Chlipala, A.: Mostly-automated verification of low-level programs in computational separation logic. In: Hall, M.W., Padua, D.A. (eds.) Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011. pp. 234–245. ACM (2011). https://doi.org/10.1145/1993498.1993526
12. Chlipala, A.: The Bedrock structured programming system: combining generative metaprogramming and Hoare logic in an extensible program verifier. In: Morrisett, G., Uustalu, T. (eds.) ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013. pp. 391–402. ACM (2013). https://doi.org/10.1145/2500365.2500592

13. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5674, pp. 23–42. Springer (2009). https://doi.org/10.1007/978-3-642-03359-9_2
14. Erbsen, A., Gruetter, S., Choi, J., Wood, C., Chlipala, A.: Integration verification across software and hardware for a simple embedded system. In: Freund, S.N., Yahav, E. (eds.) PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021. pp. 604–619. ACM (2021). https://doi.org/10.1145/3453483.3454065
15. Erbsen, A., Philipoom, J., Jamner, D., Lin, A., Gruetter, S., Pit-Claudel, C., Chlipala, A.: Foundational integration verification of a cryptographic server. Proc. ACM Program. Lang. **8**(PLDI), 1704–1729 (2024). https://doi.org/10.1145/3656446
16. Gladshtein, V., Pîrlea, G., Zhao, Q., Kurin, V., Sergey, I.: Foundational multimodal program verifiers. Proc. ACM Program. Lang. **10**(POPL), 2233–2264 (2026). https://doi.org/10.1145/3776719
17. Gordon, M.J.C., Milner, R., Morris, F.L., Newey, M.C., Wadsworth, C.P.: A metalanguage for interactive proof in LCF. In: Aho, A.V., Zilles, S.N., Szymanski, T.G. (eds.) Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978. pp. 119–130. ACM Press (1978). https://doi.org/10.1145/512760.512773
18. Gordon, M.J.C., Milner, R., Wadsworth, C.P.: Edinburgh LCF, Lecture Notes in Computer Science, vol. 78. Springer (1979). https://doi.org/10.1007/3-540-09724-4
19. Greenaway, D., Andronick, J., Klein, G.: Bridging the gap: Automatic verified abstraction of C. In: Beringer, L., Felty, A.P. (eds.) Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings. pp. 99–115. Lecture Notes in Computer Science, Springer (2012). https://doi.org/10.1007/978-3-642-32347-8_8
20. Gruetter, S., Fukala, V., Chlipala, A.: Live verification in an interactive proof assistant. Proc. ACM Program. Lang. **8**(PLDI), 1535–1558 (2024). https://doi.org/10.1145/3656439
21. Gu, R., Koenig, J., Ramananandro, T., Shao, Z., Wu, X.N., Weng, S., Zhang, H., Guo, Y.: Deep specifications and certified abstraction layers. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. pp. 595–608. ACM (2015). https://doi.org/10.1145/2676726.2676975
22. Gu, R., Shao, Z., Chen, H., Wu, X.N., Kim, J., Sjöberg, V., Costanzo, D.: CertiKOS: An extensible architecture for building certified concurrent OS kernels. In: Keeton, K., Roscoe, T. (eds.) 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016. pp. 653–669. USENIX Association (2016), https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu
23. Guéneau, A., Myreen, M.O., Kumar, R., Norrish, M.: Verified characteristic formulae for CakeML. In: Yang, H. (ed.) Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala,

Sweden, April 22-29, 2017, Proceedings. pp. 584–610. Lecture Notes in Computer Science, Springer (2017). https://doi.org/10.1007/978-3-662-54434-1_22
24. Harrison, J.: HOL light: An overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5674, pp. 60–66. Springer (2009). https://doi.org/10.1007/978-3-642-03359-9_4
25. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6617, pp. 41–55. Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_4
26. Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28**, e20 (2018). https://doi.org/10.1017/S0956796818000151
27. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D.A., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS kernel. In: Matthews, J.N., Anderson, T.E. (eds.) Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009. pp. 207–220. ACM (2009). https://doi.org/10.1145/1629575.1629596
28. Krebbers, R., Timany, A., Birkedal, L.: Interactive proofs in higher-order concurrent separation logic. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 205–217. POPL '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009855
29. Mansky, W., Appel, A.W., Nogin, A.: A verified messaging system. Proc. ACM Program. Lang. **1**(OOPSLA), 87:1–87:28 (2017). https://doi.org/10.1145/3133911
30. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
31. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings. pp. 1–19. Lecture Notes in Computer Science, Springer (2001). https://doi.org/10.1007/3-540-44802-0_1
32. Paulson, L.C.: Logic and computation - interactive proof with Cambridge LCF, Cambridge tracts in theoretical computer science, vol. 2. Cambridge University Press (1987)
33. Protzenko, J., Zinzindohoué, J.K., Rastogi, A., Ramananandro, T., Wang, P., Zanella-Béguelin, S., Delignat-Lavaud, A., Hritcu, C., Bhargavan, K., Fournet, C., Swamy, N.: Verified low-level programming embedded in F*. Proc. ACM Program. Lang. **1**(ICFP), 17:1–17:29 (2017). https://doi.org/10.1145/3110261
34. Pulte, C., Makwana, D.C., Sewell, T., Memarian, K., Sewell, P., Krishnaswami, N.: CN: verifying systems C code with separation-logic refinement types. Proc. ACM Program. Lang. **7**(POPL), 1–32 (2023). https://doi.org/10.1145/3571194

35. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings. pp. 55–74. IEEE Computer Society (2002). https://doi.org/10.1109/LICS.2002.1029817
36. Ringer, T., Palmskog, K., Sergey, I., Gligoric, M., Tatlock, Z.: QED at large: A survey of engineering of formally verified software. Found. Trends Program. Lang. **5**(2-3), 102–281 (2019). https://doi.org/10.1561/2500000045
37. Sammler, M., Lepigre, R., Krebbers, R., Memarian, K., Dreyer, D., Garg, D.: RefinedC: automating the foundational verification of C code with refined ownership types. In: Freund, S.N., Yahav, E. (eds.) PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021. pp. 158–174. ACM (2021). https://doi.org/10.1145/3453483.3454036
38. Slind, K., Norrish, M.: A brief overview of HOL4. In: Mohamed, O.A., Muñoz, C.A., Tahar, S. (eds.) Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5170, pp. 28–32. Springer (2008). https://doi.org/10.1007/978-3-540-71067-7_6
39. Tafat, A., Marché, C.: Binary Heaps Formally Verified in Why3. Research Report RR-7780, INRIA (Oct 2011), https://inria.hal.science/inria-00636083
40. Wang, Z., Wu, X., Fang, Y., Li, C., Zhong, H., Xie, L., Cao, Q., Hu, Z.: Stellis: A strategy language for purifying separation logic entailments (2025), https://arxiv.org/abs/2512.05159
41. Wu, X., Feng, Y., Lu, X., Lin, T., Liu, K., Wang, Z., Wu, S., Xie, L., Yang, C., Zhong, H., Zhang, Z., Li, J., Zhan, N., Hu, Z., Cao, Q.: QCP: A practical separation logic-based C program verification tool. In: Proceedings of the 20th International Symposium on Theoretical Aspects of Software Engineering, TASE 2026, Shanghai, China, July 4-6, 2026 (2026)
42. Xu, Q., Sanán, D., Hou, Z., Luan, X., Watt, C., Liu, Y.: Generically automating separation logic by functors, homomorphisms, and modules. Proc. ACM Program. Lang. **9**(POPL), 1992–2024 (2025). https://doi.org/10.1145/3704903
43. Ye, K.Q., Green, M., Sanguansin, N., Beringer, L., Petcher, A., Appel, A.W.: Verified correctness and security of mbedtls HMAC-DRBG. In: Thuraisingham, B., Evans, D., Malkin, T., Xu, D. (eds.) Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 2007–2020. ACM (2017). https://doi.org/10.1145/3133956.3133974
44. Zhang, H., Honoré, W., Koh, N., Li, Y., Li, Y., Xia, L., Beringer, L., Mansky, W., Pierce, B.C., Zdancewic, S.: Verifying an HTTP key-value server with interaction trees and VST. In: Cohen, L., Kaliszyk, C. (eds.) 12th International Conference on Interactive Theorem Proving, ITP 2021, Rome, Italy (Virtual Conference), June 29 - July 1, 2021. pp. 32:1–32:19. LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPICS.ITP.2021.32
45. Zhou, L., Qin, J., Wang, Q., Appel, A.W., Cao, Q.: VST-A: A foundationally sound annotation verifier. Proc. ACM Program. Lang. **8**(POPL), 2069–2098 (2024). https://doi.org/10.1145/3632911

Table 2: GLOBAL code statistics. #Global are lines of GLOBAL code in the files. Reusable proof libraries are put under class common. Here, we also include header files because benchmarks in the same class may share some code in these headers.

Class	File	#Global
common	operational.h	371
	smt_ctx.h	279
	smt_extensions.h	20
	smt.h	258
	veriftime.h	312
basic	feature.h	3
	feature.c	0
	arith.c	276
array	char_array.h	52
	traversal.c	306
	smt_array.h	282
	binary_heap.h	73
	binary_heap.c	0
	dsu.h	135
	dsu.c	0
pointer	sll.h	327
	sll.c	0
	sll_reverse.c	104
	sll_reverse_diff.c	84
	dll.h	45
	dll.c	1
	bst.h	368
	bst.c	191
hyp_allocator	allocator_def.h	299
	allocator_lem_decl.h	541
	allocator_lem.h	1428
	allocator_verif.h	889
	page.h	0
	psi_capsule.h	0
	psi_capsule.c	1
	list.h	0
	list.c	1
	allocator.h	60
	allocator.c	2

A An Example Separation-Logic Entailment Proof

To illustrate how the separation-logic tactics of Section 4.1 drive the proof of separation-logic entailments, Fig. 11 shows a small but representative entailment proof. Recall from Section 4.1 that a separation-logic goal has the form `` `asmps ?- (hp1 |-- hp2)` ``, where `` `asmps` `` is the assumption list and the conclusion is an entailment between the antecedent `` `hp1` `` and the consequent `` `hp2` ``. We walk through the proof step by step, showing the goal after each tactic application.

Initialization (lines 2–4). The proof initializes a goal tree whose `root` node holds the closed entailment to be proved and focuses on it. The goal is initially `` `?- (exists n. P n ** fact(n > 0i)) ** R |-- exists n. fact(n > 1i) ** R ** P (n-1i)` ``, with no assumptions and the two outer universals `` `P` `` and `` `R` `` still in place.

```
1  GLOBAL thm example_proof() {
2    GN root = GOAL_INIT(`forall (P:int->hprop) (R:hprop).
3      (exists n. P n ** fact(n > 0i)) ** R
4      |-- (exists n. fact(n > 1i) ** R ** P (n-1i))`);
5    FOCUS(root);
6    APPLY(INTROS_TAC);
7    APPLY(HEXISTS_PULL_TAC);
8    APPLY(HCHOOSE_TACS);
9    APPLY(FACTS_INTRO_TAC);
10   APPLY(HEXISTS_TAC, `n+1i`);
11   GNS gs = SPLIT(FACTS_SPLIT_TAC);
12   { FOCUS(gs[0]);
13     APPLY(REWRITE_TAC, int_arith_rule(`(n+1i)-1i == n`));
14     APPLY(AUTO_FRAME_TAC); }
15   { FOCUS(gs[1]);
16     APPLY(ASM_INT_ARITH_TAC); }
17   return GOAL_SOLVE(root);
18 }
```

Fig. 11: An example tactic proof of a separation-logic entailment.

INTROS_TAC *(line 6).* Introduces the universally quantifiers `` `P` `` and `` `R` ``. The entailment is unchanged, but `` `P` `` and `` `R` `` are now free in the goal. The goal is now `` `?- (exists n. P n ** fact(n > 0i)) ** R |-- exists n. fact(n > 1i) ** R ** P (n-1i)` ``.

HEXISTS_PULL_TAC *(line 7).* Pulls all existentials out of the antecedent's separating conjunction. The antecedent becomes `` `exists n. (P n ** fact(n > 0i)) ** R` ``.

HCHOOSE_TACS *(line 8).* Eliminates the antecedent's existential by choosing a fresh witness (defaults to the same name as the existential quantifier), so the antecedent becomes `` `(P n ** fact(n > 0i)) ** R` ``.

FACTS_INTRO_TAC *(line 9).* Moves the pure conjunct `` `fact(n > 0i)` `` from the antecedent into the assumption list. The goal is now `` `n > 0i ?- (P n ** R |-- exists n. fact(n > 1i) ** R ** P (n-1i))` ``.

HEXISTS_TAC *with witness* `` `n+1i` `` *(line 10).* Instantiates the consequent's existential with the user-supplied witness `` `n+1i` ``, turning the consequent into `` `fact (n+1i > 1i) ** R ** P (n+1i-1i)` ``.

SPLIT(FACTS_SPLIT_TAC) *(line 11).* Splits the `` `fact` `` conjunct in the consequent off into a separate pure subgoal, returning the two resulting goal nodes in `gs`:

- the spatial residual `gs[0]`: `` `n > 0i ?- (P n ** R |-- R ** P (n+1i-1i))` ``;
- the pure obligation `gs[1]`: `` `n > 0i ?- (n+1i) > 1i` ``.

REWRITE_TAC *and* ***AUTO_FRAME_TAC*** *(line 12-14).* Rewrites the consequent with the arithmetic equality `` `(n+1i)-1i == n` `` and cancels the syntactically identical conjuncts `` `P n` `` and `` `R` `` on both sides, leaving nothing and thus solving the goal.

ASM_INT_ARITH_TAC *(line 15).* Proves the pure obligation `` `n+1i > 1i` `` from the assumption `` `n > 0i` `` by integer arithmetic.

With both subgoals solved, `GOAL_SOLVE(root)` (line 16) validates the completed goal tree bottom-up and returns the proven theorem.

B Unproven Lemmas in the Case Study

This section informally summarizes three lemmas that remain unproven in the verification of `hyp_allocator` and are therefore currently assumed as axioms. The descriptions below are intended to convey each lemma's role in the verification.

FIND_BEST_SNOC `` `find_best` `` is the logical representation of finding the best fit for an allocation request in the allocator's free list. This lemma establishes the equivalence between the standard recursive definition of `` `find_best` `` (which recurses on the head and tail of a list) and an alternative snoc-recursive formulation (which recurses on the head segment and the last element), ensuring both approaches produce the same result.

DLL_INSERT_INV This lemma formalizes the insertion of a node into a doubly-linked circular list. The list structure is represented abstractly as a mapping from addresses to addresses, capturing forward and backward pointer information. The lemma establishes that if the original mapping satisfies the doubly linked circular list property (i.e., the successor of the predecessor of any node is that node) and the list's strict ordering property, then inserting a new node at an address between two adjacent nodes preserves these properties in the updated mapping. Moreover, the mapping remains unchanged at all addresses except those directly affected by the insertion.

DLL_DELETE_INV This lemma formalizes the removal of a node from the doubly-linked circular list. It states that deleting a node located between two adjacent addresses updates only the successor and predecessor entries of those two neighbors so that they point to each other, leaves every other mapping entry unchanged, and therefore yields a mapping that still satisfies the circularity and strict-ordering invariants of the list with the specified node absent.