# Verifying Verified Code*

Siddharth Priya[1], Xiang Zhou[1], Yusen Su[1], Yakir Vizel[2], Yuyan Bao[1], and Arie Gurfinkel[1]

University of Waterloo[1] and The Technion[2]

**Abstract.** A recent case study from AWS by Chong et al. proposes an effective methodology for Bounded Model Checking in industry. In this paper, we report on a followup case study that explores the methodology from the perspective of three research questions: (a) can proof artifacts be used across verification tools; (b) are there bugs in verified code; and (c) can specifications be improved. To study these questions, we port the verification tasks for `aws-c-common` library to SeaHorn and KLEE. We show the benefits of using compiler semantics and cross-checking specifications with different verification techniques, and call for standardizing proof library extensions to increase specification reuse. The verification tasks discussed are publicly available online.

## 1 Introduction

Bounded Model Checking (BMC) is an effective static analysis technique that reduces program analysis to propositional satisfiability (SAT) or Satisfiability Modulo Theories (SMT). It works directly on the source code. It is very precise, e.g., accounting for semantics of the programming language, memory models, and machine arithmetic. There is a vibrant ecosystem of tools from academia (e.g., SMACK [24], CPAChecker [4], ESBMC [12]), industrial research labs (e.g., Corral [19], F-SOFT [15]), and industry (e.g., CBMC [9], Crux [13], QPR [5]). There is an annual software verification competition, SV-COMP [3], with many participants. However, with a few exceptions, BMC is not actively used in software industry. Especially, when compared to dynamic analysis techniques such as fuzzing [25], or light-weight formal methods such as static analysis [2].

Transitioning research tools into practice requires case-studies, methodology, and best-practices to show how the tools are best applied. Until recently, there was no publicly available industrial case study on successful application of BMC for continuous verification[1] of C code. This has changed with [7] – a case study from the Automated Reasoning Group (ARG) at Amazon Web Services (AWS) on the use of CBMC for proving memory safety (and other properties) of several AWS C libraries. This case study proposes a verification methodology with two core principles: (a) verification tasks structured around units of

* This research was supported by grants from WHJIL and NSERC CRDPJ 543583-19.

[1] By *continuous verification*, we mean verification that is integrated with continuous integration (CI) and is checked during every commit.

functionality (i.e., around a single function, as in a unit test), and (b) the use of code to express specifications (i.e., pre-, post-conditions, and other contextual assumptions). We refer to these as *unit proofs*, and *Code as Specification (CaS)*, respectively. The methodology is efficient because small verification tasks help alleviate scalability issues inherent in BMC. More significantly, developers adopt, own, extend and even use specifications (as code) in other contexts, e.g., unit tests. Admirably, AWS has released all of the verification artifacts (code, specifications and verification libraries)[2]. Moreover, these are maintained and integrated into Continuous Integration (CI). This gave us a unique opportunity to study, validate, and refine the methodology of [7]. In this paper, we report on our experience on adapting the verification tasks of [7] to two new verification tools: a Bounded Model Checking engine of SeaHorn, and the symbolic execution tool KLEE. We present our experience as a case study that is organized around three Research Questions (RQ):

*RQ1: Does CaS empower multiple tools for a common verification task?* Code is the *lingua franca* among developers, compilers, and verification tools. Thus, CaS makes specifications understandable by multiple verification tools. To validate effectiveness of this hypothesis, we adapted the unit proofs from AWS to different tools, and report on the experience in Sec. 3.1. While giving a positive answer to RQ1, we highlight the importance of the semantics used to interpret CaS, and that effectiveness of each tool depends on specification styles.

*RQ2: Are there bugs in verified code?* Specifications written by humans may have errors. Do such errors hide bugs in verified implementations? What sanity checks are helpful to find bugs in implementations *and* specifications? The public availability of [7] is a unique opportunity to study this question. In contrast to [7], we found no new bugs in the library being verified (`aws-c-common`). However, we have found multiple errors in specifications! Reporting them to AWS triggered a massive review of existing unit proofs with many similar issues found and fixed. We report the bugs, and techniques that helped us discover them, in Sec. 3.2.

*RQ3: Can specifications be improved while maintaining CaS philosophy?* Some mistakes in specifications can be prevented by improvements to the specification language. We propose a series of improvements that significantly reduce specification burden. They are mostly in the form of built-in functions, thus, familiar to developers. In particular, we show how to make the verification of the `linked_list` data structure in `aws-c-common` significantly more efficient, while making the proofs unbounded (i.e., correct for linked list of any size).

In our case study, we used the BMC engine of SeaHorn [14] and symbolic execution tool KLEE [6]. We have chosen SeaHorn because it is conceptually similar to CBMC that was used in [7]. Thus, it was reasonable to assume that all verification tasks can be ported to it. We are also intimately familiar with

[2] `https://github.com/awslabs/aws-c-common/tree/main/verification/cbmc`

SeaHorn. Thus, we did not only port verification tasks, but proposed improvements to SeaHorn to facilitate the process. We have chosen KLEE because it is a well-known representative of symbolic execution – an approach that is the closest alternative to Bounded Model Checking.

Overall, we have ported all of the 169 unit proofs of `aws-c-common` to SeaHorn, and 153 to KLEE. The case study represents a year of effort. The time was divided between porting verification tasks, improving SeaHorn to allow for a better comparison, and, many manual and semi-automated sanity checks to increase confidence in specifications. Additionally, we have experimented with using unit proofs as fuzz targets using LLVM fuzzing library `libFuzzer` [25] and adapted 146 of the unit proofs to `libFuzzer`.

We make all results of our work publicly available and reproducible at `https://github.com/seahorn/verify-c-common`. In addition to what is reported in this paper, we have developed an extensive CMake build system that simplifies integration of additional tools. The case study is *live* in the sense that it is integrated in CI and is automatically re-run nightly. Thus, it is synchronized both with the tools we use and the AWS library we verify.

We hope that our study inspires researchers to adapt their tools to industrial code, and inspires industry to release verification efforts to study.

*Caveats and non-goals.* We focus on the issues of methodology and sharing verification tasks between different tools. The tools that we use have different strengths and weaknesses. While they all validate user-supplied assertions, they check for different built-in properties (e.g., numeric overflow, undefined behaviours, memory safety). The goal is not to compare the tools head-to-head, or to find the best tool for a given task. We have not attempted to account for the differences between the tools. Nor have we tried to completely cover all verification tasks by all tools. Our goal was to preserve the unit proofs of [7] as much as possible to allow for a better comparison. For that reason, while we do report on performance results for the different tools, we do not describe them in detail. An interested reader is encouraged to look at the detailed data we make available on GitHub. Furthermore, while we have applied fuzzing to the unit proofs, we do not focus on effectiveness and applicability of static vs dynamic verification but only on the issues of methodology.

To summarize, we make the following contributions: (a) we validate that CaS can be used to share specifications between multiple tools, especially tools that share the same techniques (i.e., BMC), or tools with related techniques (i.e., BMC and Symbolic Execution); (b) we describe in details bugs that are found in verified code (more specifically, in specifications), some are quite surprising; (c) we suggest a direction to improve CaS with additional built-in functions that simplify common specification; and (d) we make our system publicly available allowing other researches to integrate their tools, use it as a benchmark, and to validate new verification approaches on industrial code.

The rest of the paper is structured as follows. Sec. 2 recalls the methodology of unit proof and CaS. And Sec. 3 presents the architecture of the case study

```
1  void aws_array_list_get_at_ptr_harness() {
2    struct aws_array_list list;
3    /* memhavoc(&list, sizeof(struct aws_array_list))); */
4    __CPROVER_assume(aws_array_list_is_bounded(&list));
5    ensure_array_list_has_allocated_data_member(&list);
6    void **val = can_fail_malloc(sizeof(void *));
7    size_t index /* = nd_size_t() */;
8    __CPROVER_assume(aws_array_list_is_valid(&list) && val != NULL);
9    if (aws_array_list_get_at_ptr(&list, val, index) == AWS_OP_SUCCESS)
10     assert(list.data != NULL && index < list.length);
11   assert(aws_array_list_is_valid(&list)); }
```

Fig. 1: The unit proof of `aws_array_list_get_at_ptr` from [7].

and answers the three research questions. We discuss related work in Sec. 4 and offer concluding remarks in Sec. 5.

## 2 Unit Proofs with Code-as-Specification

In [7], a methodology for program verification is proposed that allows developers to write specifications and proofs using the C programming language. The core of the methodology are *unit proof*[3] and *Code as Specification (CaS)*. A *unit proof* is similar to a *unit test* in that it is a piece of a code (usually a method) that invokes another piece of code (under test) and checks its correctness [23]. Fig. 1 shows an example of a unit proof for the method `aws_array_list_get_at_ptr`, from `aws-c-common` library. It has three parts: (1) the specification of `aws_array_list_get_at_ptr`, i.e., pre- (line 8) and post-conditions (lines 10–11); (2) a call to the function under verification (line 9); and (3) the specification of the program context that the method is called from (lines 2–7). Note that all specifications are written directly in C. We call this specification style – CaS. Assumptions (or pre-conditions) correspond to `__CPROVER_assume`, and assertions (or post-conditions) correspond to **`assert`**. Specifications are factored into functions. For example, `aws_array_list_is_valid` specifies a representation invariant of the array list. In this unit proof, the context is restricted to a list of bounded size but with unconstrained elements and an `index` with (intentionally) unspecified value of type **`size_t`**. Even without expanding the code further, its meaning is clear to any C developer familiar with the library.

The unit proof is verified with CBMC [9]. CBMC uses a custom SMT solver to check that there are no executions that satisfy the pre-conditions and violate at least one of the assertions (i.e., a counterexample). Together with the explicit assertions, CBMC checks built-in properties: memory safety and integer overflow.

According to [7], CaS and unit proofs are a practical and productive verification methodology. It has been used successfully to verify memory safety (and other properties) of multiple AWS projects, including the `aws-c-common` library that we use in our case study. The library provides cross platform configuration, data structures, and error handling support to a range of other AWS

[3] In [7], these are called *proof harnesses*.

| | | LOC | | | CBMC (s) | | SeaHorn (s) | | KLEE (s) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| category | num | avg | min | max | avg | std | avg | std | count | avg | std |
| arithmetic | 6 | 33 | 11 | 40 | 3.8 | 0.8 | 0.6 | 0.1 | 6 | 0.9 | 0.3 |
| array | 4 | 97 | 78 | 112 | 5.6 | 0.0 | 1.7 | 0.7 | 4 | 32.3 | 6.0 |
| array_list | 23 | 126 | 77 | 181 | 35.8 | 60.8 | 2.5 | 3.3 | 23 | 55.4 | 49.3 |
| byte_buf | 29 | 97 | 50 | 188 | 17.6 | 47.3 | 1.0 | 0.8 | 27 | 75.3 | 124.1 |
| byte_cursor | 24 | 98 | 47 | 179 | 6.9 | 3.8 | 1.0 | 0.5 | 17 | 12.8 | 14.4 |
| hash_callback | 3 | 115 | 49 | 198 | 9.7 | 5.5 | 4.9 | 3.6 | 3 | 64.0 | 45.5 |
| hash_iter | 4 | 177 | 169 | 185 | 12.8 | 9.2 | 9.2 | 15.0 | 3 | 20.8 | 9.7 |
| hash_table | 19 | 172 | 36 | 328 | 23.5 | 33.3 | 5.3 | 7.5 | 15 | 104.6 | 333.4 |
| linked_list | 18 | 115 | 17 | 219 | 58.9 | 209.4 | 2.0 | 2.1 | 18 | 0.7 | 0.1 |
| others | 2 | 15 | 10 | 21 | 3.5 | 0.0 | 0.5 | 0.0 | 1 | 0.7 | – |
| priority_queue | 15 | 187 | 136 | 258 | 208.1 | 303.4 | 10.6 | 16.9 | 15 | 46.4 | 11.6 |
| ring_buffer | 6 | 155 | 56 | 227 | 20.0 | 19.5 | 29.5 | 34.2 | 6 | 48.1 | 26.4 |
| string | 15 | 87 | 11 | 209 | 6.3 | 1.3 | 2.9 | 1.8 | 15 | 139.7 | 159.7 |
| **Total** | 168 | Loc | 20,190 | | Time | 6,475 | Time | 691 | Time | 8,577 | |

Fig. 2: Verification results for CBMC, SeaHorn and KLEE.

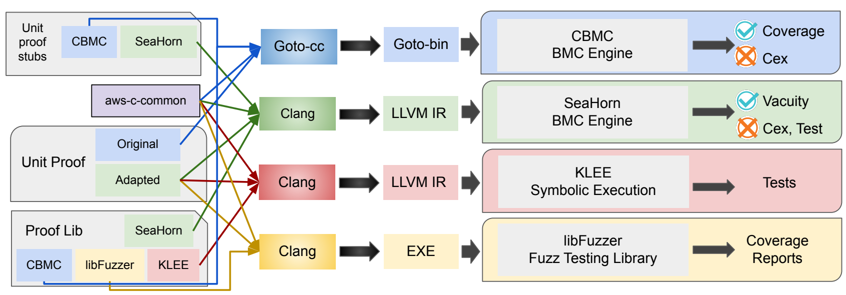



Fig. 3: Architecture of the case study.

C libraries and SDKs. It is the foundation of many security related libraries, such as the AWS Encryption SDK for C [7]. It contains 13 data structures, 169 unit proofs that verify over 20K lines of code (LOC). Fig. 2 shows the LOC and running time for each data structure.

## 3 Case Study

The architecture of our case study is shown in Fig. 3. To compare with CBMC, we use two tools based on the LLVM framework [20]: SeaHorn and KLEE. SeaHorn [14] is a verification framework. We used the bit- and memory-precise BMC developed during the case study. Its techniques are closest to CBMC. KLEE [6] is a well-known symbolic execution tool. It is an alternative to BMC for bounded exhaustive verification. In addition, we have experimented with `libFuzzer` – a coverage-guided random testing framework. It does no symbolic reasoning, and, together with address sanitizer, is known to be effective at discovering memory errors. Fuzzing results are available online. [4]

The rest of the section describes the research questions and our findings.

### 3.1 RQ1: Does CaS Empower Multiple Tools?

Hypothetically, CaS methodology enables sharing the same formal specification among multiple, potentially distinct, tools and techniques. For example, semantic analyses of IDEs and compilers can catch simple semantics bugs and inconsistencies in specifications. Fuzzers can validate specifications through testing. Symbolic execution can supplement BMC by capitalizing on a different balance in performance versus precision. Static analysis tools can be used to compute inductive invariants. However, is the hypothesis true in practice?

To validate the hypothesis, we adapted the unit proofs from `aws-c-common` to two distinct verification techniques: BMC with SeaHorn and symbolic execution with KLEE. We have also attempted to use unit proofs as fuzz targets for `libFuzzer`. While our experience supports the hypothesis, we encountered two major challenges: *semantics* and *effectiveness of specifications.*

*Semantics.* Code without semantics is meaningless. Developers understand code without being versed in formal semantics, however, many technical details and "corner cases" are often debated. This is especially true for C – *"the semantics of C has been a vexed question for much of the last 40 years"* [21]. Clear semantics are crucial when code (and CaS) are used with multiple tools.

The unit proofs in [7] do not follow the C semantics. For example, consider the proof in Fig. 1. According to C, it has no meaning as both `list` (line 2) and `index` (line 7) are used uninitialized. CBMC treats uninitialized variables as non-deterministic. So it is well-defined for CBMC, but not for other tools.

What is a good choice of semantics for CaS? In [21], two semantics are described – the ISO C Standard and the *de facto* semantics of compilers. Developers understand (and use) the de facto semantics. For example, comparison of arbitrary pointers is undefined according to ISO C, but defined consistently in mainstream compilers (and used in `aws-c-common`!). Therefore, we argue that CaS must use the de facto semantics. Furthermore, unit proofs must be compilable and, therefore, executable, so developers can execute them not *just* in their heads (like with [7]). Note that de facto semantics is not complete with regards to C semantics, but is a commonly agreed upon subset. What de facto semantics does not cover is compiler dependent semantics.

In our experience, using CaS with the de facto semantics is not hard. For example, to adapt Fig. 1, we introduced `memhavoc` and `nd_size_t`, shown as comments, that fills a memory region at a given address with non-deterministic bytes, and returns a non-deterministic value of type **`size_t`**, respectively.

[4] `https://seahorn.github.io/verify-c-common/fuzzing_coverage/index.html`.

```c
size_t len = nd_size_t();
size_t cap = nd_size_t();
assume(len <= cap);
assume(cap <= MAX_BUFFER);

buf->len = len;
buf->capacity = cap;
buf->buffer = can_fail_malloc(
  cap * sizeof(*(buf->buffer)));
buf->allocator = sea_allocator()
```

(a) for SeaHorn

```c
size_t cap = nd_size_t();
assume(cap <= MAX_BUFFER);
buf->buffer = can_fail_malloc(
  cap * sizeof(*(buf->buffer)));
if (buf->buffer) {
    size_t len = nd_size_t();
    assume(len <= cap);
    buf->len = len;
    buf->capacity = cap;
}
else {
    buf->len = 0;
    buf->capacity = 0;
}
buf->allocator = sea_allocator();
```

(b) for KLEE

```c
size_t len = nd_size_t();
size_t cap = nd_size_t();
cap len = (cap == 0) ? 0 : len

buf->len = len;
buf->capacity = cap;
buf->buffer = can_fail_malloc(
  cap * sizeof(*(buf->buffer)));
buf->allocator = sea_allocator();
```

(c) for libFuzzer

Fig. 4: Tool-specific implementations for initialize_byte_buf.

*Effectiveness of specifications.* We used three different tools on the same unit proof. Each tool requires slightly different styles of specifications to be effective. We believe that these stylistic differences between specifications can be captured by traditional code refactoring techniques (i.e., functions, macros, etc.). However, this is not easy whenever the specifications have not been written with multiple tools (and with their strengths and weaknesses) in mind. A significant part of our work has been in refactoring unit proofs from [7] to be more modular.

We illustrate this with the pre-condition for the byte_buf data-structure. In [7], data structures are assumed to be initially non-deterministic, and various assumptions throughout the unit proof are used to restrict it (e.g., lines 2–5 in Fig. 1). This impedes specification re-use since different tools work well with different styles of pre-conditions. For example, symbolic execution and fuzzing require memory to be explicitly allocated, and all tools that use de-facto semantics require all memory be initialized before use.

For byte_buf, we factored out its pre-conditions into a function init_byte_buf.[5] Its implementations for SeaHorn, KLEE, and libFuzzer are shown in Fig. 4. It takes buf structure as input, and initializes its fields to be consistent with the representation invariant of byte_buf.

**SeaHorn** initialization is closest to the original of [7]. Fields are initialized via calls to external functions (nd_<type>) that are assumed to return arbitrary values. Representation invariants (i.e., length is less or equal to capacity), as well as any upper bounds on buffer size are specified with *assumptions*. Note that can_fail_malloc internally initializes allocated memory via a call to memhavoc, ensuring that reading buf->buffer is well-defined.

**KLEE** initialization is similar to SeaHorn, but special care must be taken about the placement of assumptions, and implementation of can_fail_malloc. In particular, KLEE prefers that memory allocation functions are given explicit size, otherwise, it picks a concrete size non-deterministically. Special cases, like buf->buffer being NULL, are split in the initialization to aid KLEE during symbolic execution. Similar changes can be done for SeaHorn, but are not as

[5] Similarly, we introduced init_array_list to replace lines 2–5 in Fig. 1.

effective. For that reason, we chose to keep SeaHorn initialization as close to [7] as possible, but adjusted the one for KLEE to be most effective.

**`libFuzzer`** initialization is the most different since non-determinisim must be replaced by randomness. In this case, `nd_<type>` functions are implemented using the random inputs generated by `libFuzzer`. *Assumptions* are implemented by aborting the current fuzzing run if the condition evaluates to `false`. Of course, this limits fuzzing effectiveness since the fuzzer must randomly guess inputs to pass all of the assumptions. For that reason, as many assumptions as possible are modeled by an explicit initialization. For example, in line 3 of Fig. 4c, `cap` is re-assigned to the modulo of `MAX_BUFFER` if `libFuzzer` generated a value exceeding `MAX_BUFFER`. This way, code after line 3 always executes regardless of the return value of `nd_size_t()` in line 2.

Overall, our results indicate that CaS empowers multiple verification tools to share specifications among them. Common refactoring techniques make specifications sharing effective. Specifications are easiest to share among tools that use similar techniques.

*Discussion.* We conclude this section with a discussion of our experience in using de facto semantics. First, the code of `aws-c-common` is written with de facto semantics in mind. We found that in [7] it had to be extended with many conditional compilation flags to provide alternative implementations that are compatible with CBMC or that instruct CBMC to ignore some seemingly undefined behavior. However, we have not changed any lines of `aws-c-common`. We analyze the code exactly how it is given to the compiler – improving coverage. Second, a compiler may generate different target code for different architectures. By using the compiler as front-end, we check that the code is correct as compiled on different platforms. This is another advantage of CaS. Third, compilers may provide additional safety checks. For example, `aws-c-common` uses GCC/Clang built-in functions for overflow-aware arithmetic. By using de facto semantics, all the tools used in the case study were able to deal with this in both CaS and code seamlessly. Fourth, `aws-c-common` uses inline assembly to deal with speculative execution-based vulnerabilities [17]. While inline assembly is not part of the ISO C standard, it is supported by compilers. Thus, it is not a problem for `libFuzzer`. We developed techniques to handle inline asm in SeaHorn. For KLEE, we had to ignore such unit proofs.

### 3.2 Are there bugs in verified code?

Specifications may have errors as they are just programs: "Writing specifications can be as error-prone as writing programs".[22] Although [7] suggests to use code coverage and code review to increase the confidence in specifications, we still found non-trivial bugs. We summarize three most interesting ones below. A complete list of all bugs we found is available in **??**.

*Bug 1.* Fig. 5a shows the definition of `byte buffer` that is a length delimited byte string. Its data representation should be either the buffer (`buf`) is `NULL`

```
typedef
struct byte_buf {
  char* buf;
  int len, cap;
} BB;
bool BB_is_ok(BB *b)
{ return (b->len == 0
        || b->buf);  }
```

(a) bug 1

```
1 assume(0 <= b && b <= 10);
2 if (a < (b - 5) &&
3     a >= (b + 5))
4 {
5   assert(c > 0);
6 }
```

(b) bug 2

```
1 void ht_del_over(HASH_TB *t) {
2   /* remove entry */
3   /* t->entry_count--; */
4 }
```

(c) bug 3

Fig. 5: Simplified code for specification bugs.

or its capacity (`cap`) is 0 (not the `len` as defined in `BB_is_ok`). We found this bug by a combination of sanity checks in SeaHorn and our model of the memory allocator (i.e., `malloc`). More details are explained in **??**.The bug did not manifest in [7] because other pre-conditions ensured that `buf` is always allocated. Our report of this bug to AWS triggered a massive code auditing effort in `aws-c-common` and related libraries where many related issues were found.[6]

*Bug 2.* Fig. 5b shows a verification pattern where a property (line 5) is checked on the program path (from lines 1 to 5). As the condition at lines 2 and 3 can never be true, the property cannot be checked either. Our vacuity detection (discussed later) found the bugs occurring in this pattern. More details on the bug can be found in **??**. Note that the bug was missed by the code coverage detection used by CBMC, thus, may have been present for several years.

*Bug 3.* To make verification scalable, the verification of method A that calls another method B may use a *specification stub* that approximates the functionality of B. AWS adopts this methodology when verifying the iterator of a hash table. The iterator calls a function `ht_del` to remove an element in a hash table. During verification `ht_del` is approximated by a specification stub shown in Fig. 5c. However, the approximation does not decrement `entry_count`, i.e., line 3 should be added to the spec for correct behavior. In [7], the use of the buggy stub hides an error in the specification. See more details in **??**.

*Discussion.* Code coverage of a unit proof is, at best, a sanity check for CaS. It reports which source lines of the specification and code under verification are covered under execution. However, because source lines can remain uncovered for legitimate reasons e.g., dead code, interpreting a coverage report is not straightforward. There is no obvious *pass/fail* criterion. Thus, we found that code coverage may be insufficient to detect bugs in CaS reliably. In fact, bugs exist for multiple years even after code coverage failures. To help find bugs in CaS with SeaHorn, we adapted *vacuity detection* [18] to detect unreachable post-conditions. Vacuity detection checks that every assert statement is reachable. We

[6] An example is `https://github.com/awslabs/aws-c-common/pull/686/commits`.

```
linked_list l;
Node *p = malloc(sizeof(Node));
l.head.next = p;
for (int idx=0; idx < MAX; idx++) {
  Node *n = malloc(sizeof(Node));
  p->next = n;
  p = n; }
p->next = &l.tail;
l.tail.prev = p;
list_front(l);
Node *nnode = l.head.next;
for (int idx=0; idx < MAX; idx++) {
  nnode = nnode->next;  }
assert(nnode == l.tail);
```

(a) Spec in the style of [7]

```
linked_list l;
Node *n = malloc(sizeof(Node));
n->next = nd_voidp();
l.head.next = n;
l.tail.prev = nd_voidp();
list_front(l);
assert(l.head.next == n);
```

(b) New specification

Fig. 6: Simplified code for differing CaS specifications.

```
  char buf[SZ];
  init_buf(buf, SZ);
  int idx = nd_int();
  assume(0 <= idx && idx < SZ);
  char saved = buf[idx];
  read_only_op(buf);
  assert(saved == buf[idx]);
```

(a) Spec in style of [7]

```
  char buf[SZ];
  init_buf(buf, SZ);
  tracking_on();
  read_only_op(buf);
  assert(!is_mod(buf));
```

(b) Spec using a built-in `is_mod`

Fig. 7: Two styles of specifications for a read only buffer operation.

encountered engineering challenges when developing vacuity detection. For example, we received spurious warnings due to code duplication. We silenced such warnings by only reporting a warning if all duplicate asserts reported a vacuity failure. In addition, due to CaS, an unreachable assertions may be removed by compiler's dead code elimination. This is not desirable for vacuity detection. To mitigate this issue, we report when dead code is eliminated. However, since many eliminations are unrelated to specs, there is noise in the report which makes it un-actionable. Interaction between dead code removal by the compiler and vacuity detection remains an open challenge for us.

We have found bugs in specifications, but we do not know what bugs remain. As shown in this section, the bugs were found with a combination of manual auditing and tools. However, these techniques are far from complete.

### 3.3 Can specifications be improved while maintaining the CaS philosophy?

There are many alternative ways to express a specification in CaS. In this section, we illustrate how to make proofs more efficient and make specs more readable. For example, a unit proof can fully instantiate a data structure (as in a unit test), or minimally constrain it (as in [7]). In this section, we illustrate this by describing our experience in making `linked_list` unit proofs unbounded (and more efficient). Furthermore, we believe that extending the specification language with additional verifier-supported built-in functions simplifies specs while making them easier to verify. We illustrate this with the built-ins developed for SeaHorn to specify absence of side-effects.

*Linked List.* A common pattern in *unit proofs* is to assume the representation invariant of a data structure, and to assert it after invocation of the function under verification along with other properties that must be maintained by the function. For example, a simplified version of its unit proof from [7] is shown in Fig. 6a. The pre-conditions are specified by (explicitly) creating a list in lines 4–7 using a loop. The post-condition is checked by completely traversing the list in lines 12–14. This specification is simple since it closely follows the style of unit tests. However, it is inefficient for BMC: (a) unrolling the loops in the pre- and post-conditions blows up the symbolic search space; (b) it makes verification of the loop-free function `list_front` bounded, i.e., verification appears to depend on the size of the list in the pre-conditions.

Our alternative formulation is to construct a partially defined linked list stub as shown in Fig. 8a. This stub can be used to verify `list_front` since it is expected that only the first node after head is accessed. The resulting CaS is shown in Fig. 6b. The `next` field of `n` points to a potentially invalid address (returned by `nd_voidp`). Either `list_front` never touches `n->next` or has a memory fault. Finally, the assert on line 7 in Fig. 6b checks that `list_front` did not modify the head of the list either. If there is no memory fault, then `list_front` did not modify the linked list after the node `n`. Our specification is not inductive. It uses the insight that the given linked list API only ever accesses a single element. It, therefore, avoids loops in both the pre- and post-conditions and verifies `list_front` for linked lists of any size.

Unfortunately, our new spec in Fig. 6b is difficult to understand by non-experts because it relies on the interplay between `nd_voidp` and memory safety checking. To make the spec accessible, we hide the details behind a helper API. Fig. 9 shows the unit proof for `aws_linked_list_front` with this API. The function `sea_nd_init_aws_linked_list_from_head` constructs partial `aws_linked_list` instances with non-deterministic length (as shown in Fig. 8a). The function `aws_linked_list_save_to_tail` saves concrete linked list nodes from the partial `aws_linked_list`. Finally, the function `is_aws_list_unchanged_to_tail` is used in post-conditions to check that linked list nodes are not modified. The unit proof for `aws_linked_list_front` is not only more efficient than the original CBMC proof, but it is also a *stronger* specification. For example, if `aws_linked_list_front` removes or modifies a linked list node, our unit proof catches this as a violation, while the original proof only checks whether the returned value is valid and whether the linked list is well formed. The API we devised is generalized to work with all linked list operations in `aws-c-common`. For operations which access the node before the tail we construct a partially defined stub as shown in Fig. 8b while Fig. 8c is constructed for operations which access the list from both ends. We provide corresponding versions of the above API to save and check immutability of linked list nodes for each kind of stub.

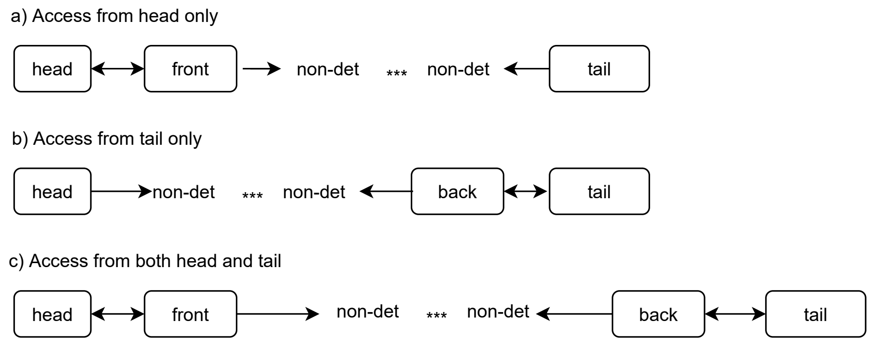



Fig. 8: Linked list stubs for proofs.

```
void aws_linked_list_front_harness() {
  /* data structure */
  struct aws_linked_list list;
  struct saved_aws_linked_list to_save = {0};
  size_t size;

  sea_nd_init_aws_linked_list_from_head(&list, &size);
  struct aws_linked_list_node *start = &list.head;
  aws_linked_list_save_to_tail(&list, size, start, &to_save);

  // precondition in function does not accept empty linked list
  assume(!aws_linked_list_empty(&list));

  /* perform operation under verification */
  struct aws_linked_list_node *front = aws_linked_list_front(&list);

  /* assertions */
  sassert(list.head.next == front);
  sassert(aws_linked_list_node_prev_is_valid(front));
  sassert(aws_linked_list_node_next_is_valid(front));
  sassert(is_aws_list_unchanged_to_tail(&list, &to_save));

  return 0;
}
```

Fig. 9: SeaHorn unit proof for `aws_linked_list_front`.

*Increasing CaS expressiveness.* Verification tools should provide built-ins to aid in concise specifications. Moreover, such built-ins enable specifications that are not otherwise expressible in CaS. For example, Fig. 7b uses a SeaHorn built-in, `is_mod`, to specify that `read_only_op` does not change the buffer. This built-in returns true if memory pointed by its argument is modified since the last call to `tracking_on`. In contrast, the original specification for CBMC in Fig. 7a is tricky. It saves a byte from some position in the buffer (lines 3–5), and checks that it is not changed (line 7). This example illustrates that built-ins make specifications simpler and more direct. They ease specification writing for users and might be exploited efficiently by verification tools. As another example, SeaHorn provides a built-in `is_deref` to check that a memory access is within bounds, which is not (easily) expressible in C.

*Discussion.* CaS enables concise specifications and efficient proofs. As advanced verification techniques may not generalize, a standard extension is needed, such as verification-specific built-in functions. The semantics of these can be provided by a run-time library, validated by fuzzing and supported by multiple verification tools. Additional case studies are needed to identify a good set of built-ins. A standard extension can increase specification reuse and make verification more productive and effective.

## 4 Related work

To our knowledge, [7] is the first significant, publicly available, example of an application of BMC on industrial code that is actively maintained with the code. Thus, our work is the first exploration of potential issues with software verified in this way. The closest verification case studies are `coreutils` with KLEE [6] and `busybox ls` with CBMC [16]. However, those focus on the scalability of a specific verification technology, while we focus on methodology, reuse, and what bugs might be hidden in the verification effort.

As we mentioned in the introduction, the Software Verification Competition (SV-COMP) [3] provides a large collection of benchmarks, and, an annual evaluation of many verification tools. However, it is focused on performance and soundness of the tools. The benchmarks are pre-processed to fit the competition format. At that stage, it is impossible to identify and evaluate the specifications, or to modify the benchmarks to increase efficiency of any particular tool. We hope that our case study can serve as an alternative benchmark to evaluate suitability of verification tools for industrial transition.

In addition to [7], there has been number of other recent applications of BMC at AWS, including [11,8,10]. However, they are either not publicly available, too specialized, or not as extensive as the case study in [7].

Using code as specification has a long history in verification tools, one prominent example is Code Contracts introduced in Spec$^\sharp$ [1]. One important difference is that in our case CaS is used to share specifications between completely different tools that only share the semantics of the underlying programming language, and the language itself is used to adapt specifications to the tools.

## 5 Conclusion

This case study would not have been possible without artifacts released by AWS in [7]. To our knowledge, it is the first publicly available application of BMC (to software) in industry. Related case studies on verification are those on `coreutils` with KLEE [6] and on `busybox ls` with CBMC [16]. SV-COMP is a large repository of benchmarks, but its goals are different from an actively maintained industrial project. The availability of both methodology and artifacts has given us a unique opportunity to study how verification is applied in industry and to improve verification methodology. We encourage industry to release more benchmarks to enable further studies by the research community.

In addition to answering the research questions, we are contributing a complete working system that might be of interest to other researchers. We have implemented a custom build system using CMake that simplifies integrating new tools. We provide Docker containers to reproduce all of the results. We created continuous integration (CI) on GitHub that nightly re-runs all the tools on the current version of `aws-c-common`. Since we use standard tools, the project integrates seamlessly into IDEs and refactoring tools. The CI runs are done in parallel by CTest. Running SeaHorn takes under 8 minutes!

While comparing different tools on performance is not our primary concern, in Fig. 2, we show the running time for all of the verification tools, collected on the same machine. Runtime for individual unit proofs are shown in **??** (and online). For `libFuzzer`, we make the detailed coverage report available online. We stress that while the tools check the same explicit assertions, they check different built-in properties. Thus, running time comparison must be taken with a grain of salt.

Our main conclusion is in agreement with [7], and we strengthen the evidence for it. CaS is a practical and scalable approach for specifications that is easy to understand and empowers many tools. We argue that using de facto compiler semantics in CaS is key for enabling many verification tools, each with its own characteristic, to be used on the same verification problem. We find that specifications can be written in different ways and specification writer must account for both verification efficiency and developer readability. We suggest that a set of common built-ins be shared by different verification tools. Such built-ins improve the expressive power of CaS while retaining portability across verification tools. With built-ins defined in a specification library, software developers will be able to write unit proofs in a way no difference than programming with libraries provided by some framework.

Today, formal verification is not the primary means of building confidence in software quality. Our hope is that case studies like this one are useful to both software engineering researchers and practitioners who want to make formal methods an integral part of software development. To further this agenda, we plan to continue applying the CaS methodology to larger and more complex code bases (and languages) in the future.

## References

1. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. Commun. ACM **54**(6), 81–91 (2011)
2. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Gros, C., Kamsky, A., McPeak, S., Engler, D.R.: A few billion lines of code later: using static analysis to find bugs in the real world. Commun. ACM **53**(2), 66–75 (2010), `https://OPTdoi.org/10.1145/1646353.1646374`
3. Beyer, D.: Advances in automatic software verification: SV-COMP 2020. In: Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on

Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12079, pp. 347–367. Springer (2020)
4. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 184–190. Springer (2011)
5. Büning, M.K., Sinz, C., Faragó, D.: QPR verify: A static analysis tool for embedded software based on bounded model checking. In: Software Verification - 12th International Conference, VSTTE 2020, and 13th International Workshop, NSV 2020, Los Angeles, CA, USA, July 20-21, 2020, Revised Selected Papers. Lecture Notes in Computer Science, vol. 12549, pp. 21–32. Springer (2020)
6. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings. pp. 209–224. USENIX Association (2008)
7. Chong, N., Cook, B., Kallas, K., Khazem, K., Monteiro, F.R., Schwartz-Narbonne, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: Code-level model checking in the software development workflow. In: ICSE-SEIP 2020: 42nd International Conference on Software Engineering, Software Engineering in Practice, Seoul, South Korea, 27 June - 19 July, 2020. pp. 11–20. ACM (2020)
8. Chudnov, A., Collins, N., Cook, B., Dodds, J., Huffman, B., MacCárthaigh, C., Magill, S., Mertens, E., Mullen, E., Tasiran, S., Tomb, A., Westbrook, E.: Continuous formal verification of amazon s2n. In: Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10982, pp. 430–446. Springer (2018)
9. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings. Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer (2004)
10. Cook, B., Döbel, B., Kroening, D., Manthey, N., Pohlack, M., Polgreen, E., Tautschnig, M., Wieczorkiewicz, P.: Using model checking tools to triage the severity of security bugs in the Xen hypervisor. In: 2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020. pp. 185–193. IEEE (2020), `https://OPTdoi.org/10.34727/2020/isbn.978-3-85448-042-6_26`
11. Cook, B., Khazem, K., Kroening, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: Model checking boot code from AWS data centers. In: Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10982, pp. 467–486. Springer (2018), `https://OPTdoi.org/10.1007/978-3-319-96142-2_28`
12. Gadelha, M.Y.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B., Nicole, D.A.: ESBMC 5.0: an industrial-strength C model checker. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018. pp. 888–891. ACM (2018)

13. Galois: Crux: A Tool for Improving the Assurance of Software Using Symbolic Testing, `https://crux.galois.com/`
14. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn Verification Framework. In: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 343–361. Springer (2015)
15. Ivancic, F., Yang, Z., Ganai, M.K., Gupta, A., Shlyakhter, I., Ashar, P.: F-soft: Software verification platform. In: Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3576, pp. 301–306. Springer (2005)
16. Kim, Y., Kim, M.: SAT-Based Bounded Software Model Checking for Embedded Software: A Case Study. In: 21st Asia-Pacific Software Engineering Conference, APSEC 2014, Jeju, South Korea, December 1-4, 2014. Volume 1: Research Papers. pp. 55–62. IEEE Computer Society (2014)
17. Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. meltdownattack.com (2018)
18. Kupferman, O.: Sanity checks in formal verification. In: CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4137, pp. 37–51. Springer (2006)
19. Lal, A., Qadeer, S.: Powering the static driver verifier using Corral. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014. pp. 202–212. ACM (2014)
20. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA. pp. 75–88. IEEE Computer Society (2004)
21. Memarian, K., Matthiesen, J., Lingard, J., Nienhuis, K., Chisnall, D., Watson, R.N.M., Sewell, P.: Into the depths of C: elaborating the de facto standards. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016. pp. 1–15. ACM (2016)
22. Moy, Y., Wallenburg, A.: Tokeneer: Beyond formal program verification. Embedded Real Time Software and Systems **24** (2010)
23. Osherove, R.: The Art of Unit Testing: With Examples in .Net. Manning Publications Co. (2009)
24. Rakamaric, Z., Emmi, M.: SMACK: Decoupling Source Language Details from Verifier Implementations. In: Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 106–113. Springer (2014)
25. Serebryany, K.: libFuzzer: A library for coverage-guided fuzz testing, `https://llvm.org/docs/LibFuzzer.html`

```c
bool aws_byte_buf_is_valid(const struct aws_byte_buf *const buf) {
  return buf != NULL &&
  ((buf->capacity == 0 && buf->len == 0 && buf->buffer == NULL) ||
  (buf->capacity > 0 && buf->len <= buf->capacity &&
  AWS_MEM_IS_WRITABLE(buf->buffer, buf->len /* buf->capacity */)));
}
```

Fig. 10: Representation invariant of `aws_byte_buf`.

## A Description of Specification Bugs Found

In this section, we describe the bugs that we have found in the unit proofs of `aws-c-common`.

### A.1 Representation Invariant of `byte_buf`

*File.* `source/byte_buf.c`

*Description.* The bug is in the representation invariant of the `byte_buf` data structure. The code is shown in **??**, and the revision is shown in the comment. The `byte_buf` data structure contains a field `buffer` that points to a writable memory region of size at least `capacity`. In the representation invariant, `len` was used instead of `capacity`. This allows for `buffer` to be `NULL` when `len` is 0 and `capacity` is not zero. The buggy specification is *not* an invariant. We showed that it is not maintained by the API of `byte_buf`.

The representation invariant is used both as pre- and post-condition in unit proofs, and also in unit tests. The buggy version is weaker than the correct one. Thus, it is true of any correct instance of `byte_buf`. For this reason, it is the bug that has not been detected by unit tests.

*Story.* According to GitHub logs, the bug was in the code for at least two years. We found it indirectly using vacuity in SeaHorn. We have initially modelled memory allocation (i.e., calls to `malloc`) as never failing. That is, never returning a `NULL` pointer. Under this assumption, some post-conditions became vacuous. We have then re-defined `malloc` to non-deterministically fail. This caused one of the unit proofs that was using this representation invariant to fail with a counterexample. Examining the counterexample manually lead to the discovery of the bug.

The bug was reported to AWS. It has pointed to a deeper issue of how `malloc` should be modelled. This resulted in review of all CBMC-based proofs at AWS with numerous issues uncovered and fixed. The bug is fixed in `aws-c-common` in commit `ec70687`[7].

### A.2 Bug in verification helper `assert_bytes_match`

*File.* `verification/cbmc/sources/utils.c`

[7] `https://github.com/awslabs/aws-c-common/pull/686`.

```c
void assert_bytes_match(const uint8_t *const a,
                        const uint8_t *const b, const size_t len) {
  assert(/* len == 0  || */ !a == !b);
  if (len > 0 && a != NULL && b != NULL) {
    size_t i;
    /* prevent spurious pointer overflows */
    __CPROVER_assume(i < len && len < MAX_MALLOC);
    assert(a[i] == b[i]);
  }
}
```

Fig. 11: Verification library function `assert_bytes_match`.

*Description.* The bug in verification library function that checks that two byte regions are equivalent. The bug and the revision (in comments) are shown in **??**. This function is used to compare content of zero-terminated strings with non-zero terminated byte buffers. A zero sized string is not `NULL` (it needs one character for the zero-terminator). A zero sized byte buffer is `NULL` because `aws-c-common` ensures that allocating 0 bytes returns `NULL`. Thus, the buggy version of the function returns false when comparing an empty string with an empty byte buffer. This contradicts the name and the use of the function in unit proofs.

*Story.* According to GitHub logs, the bug was in the code for at least two years. We found the bug by allowing `malloc` to fail and return `NULL` non-deterministically. This enabled code paths in existing unit proofs are infeasible under previous assumptions. This caused some unit proofs to fail. Examining the counterexamples manually lead to discovery of this bug.

### A.3 Post-condition bug in unit proof for `aws_mul_size_checked_harness.c`

*File.* `verification/cbmc/proofs/aws_mul_size_checked/aws_mul_size_checked_harness.c`

*Description.* The bug is in the post-condition of the unit proof. The bug and the revision (in comments) are shown in **??**. The function `aws_mul_size_checked` is a checked overflow multiplication. If the multiplication does not overflow, the result of the multiplication is stored in `r`, otherwise, the function returns an error code. The assertion at line 16 is intended to check for overflow, but instead, it checks for overflow of addition. The verification environment is restricted at line 9 to two special values. The post-condition happens to be true for these two inputs.

*Story.* The comments in the unit proof restrict the environment because of scalability issues. This was not a problem for SeaHorn. When migrating, we have removed the assumption at line 9 and found a counterexample.

The bug was fixed by AWS team as shown in the commented assertion at line 16. The revision, arguably, creates a different bug. When the code if

```
#include <aws/common/math.h>
extern int nondet_int(void);
extern uint64_t nondet_uint64_t(void);
void aws_mul_size_checked_harness() {
  /*
  * In this particular case, full range of nondet inputs leads
  * to excessively long runtimes, so use 0 or UINT64_MAX instead.
  */
  uint64_t a = (nondet_int()) ? 0 : UINT64_MAX;
  uint64_t b = nondet_uint64_t();
  uint64_t r = nondet_uint64_t();
  if (!aws_mul_u64_checked(a, b, &r)) {
    assert(r == a * b);
  } else {
    assert((b > 0) && (a > (UINT64_MAX - b)));
    /*  assert(__CPROVER_overflow_mult(a, b));
  }
}
```

Fig. 12: Unit proof for `aws_mul_size_checked`.

`aws-c-common` is compiled for CBMC, it redefines `aws_mul_u64_checked` with a call to `__CPROVER_overflow_mult`. Thus, this unit proof does not check any actual executable code in the library.

The library provides several implementations of checked arithmetic. The most common one uses checked arithmetic builtins provided by both Clang and GCC compilers, and others rely on inline assembly. In SeaHorn, we verify the version that is compiled using compiler builtins. Thus, with a similar unit proof, SeaHorn is able to check correctness of the actual implemented code.

### A.4 Post-condition bug in unit proof for `s_swap` of priority queue

*File.* `verification/cbmc/proofs/aws_priority_queue_s_swap/aws_priority_queue_s_swap_harness.c`

*Description.* The bug is in one of the post-conditions of the function `s_swap` that is used in the implementation of the priority queue data structure. The complete unit proof is too long to reproduce here, so we present a relevant snippet in **??** instead. Since the function `s_swap` is private to the priority queue translation unit, a special syntax is used to call it directly from the unit proof (line 4). The bug is in the condition of the if-statement at line 8. Conjunction is used instead of disjunction. The revision is shown in comments and corresponds to the comment at line 6.

*Story.* The bug was found by vacuity detection. The assertions inside the function `assert_array_list_equivalence` are unreachable. The bug has been reported, but is not yet fixed. GitHub logs show that the bug has been active for at least two years. The coverage results from CBMC show that the call to `assert_array_list_equvalence` is not covered (i.e., not executed by the unit proof). However, this seems to have been missed.

```c
void aws_priority_queue_s_swap_harness() {
  ...
  /* Perform operation under verification */
  __CPROVER_file_local_priority_queue_c_s_swap(&queue, a, b);
  ...
  /* All the elements in the container except for a and b should stay
     unchanged */
  size_t ob_i = old_byte.index;
  if ((ob_i < a * item_sz && /* || */ ob_i >= (a + 1) * item_sz) &&
      (ob_i < b * item_sz && /* || */ ob_i >= (b + 1) * item_sz)) {
    assert_array_list_equivalence(&queue.container, &old, &old_byte);
  }
  ...
}
```

Fig. 13: Snippet of the unit proof for `s_swap` of priority queue.

```c
void aws_hash_callback_string_eq_harness() {
  const struct aws_string *str1 = ...;
  const struct aws_string *str2 = ...;

  __CPROVER_assume(aws_c_string_is_valid(str1));
  __CPROVER_assume(aws_c_string_is_valid(str2));
  /* __CPROVER_assume(aws_string_is_valid(str1)); */
  /* __CPROVER_assume(aws_string_is_valid(str2)); */



  bool rval = aws_hash_callback_string_eq(str1, str2);
  if (rval) {
    assert(str1->len == str2->len);
    assert_bytes_match(str1->bytes, str2->bytes, str1->len);
  }
}
```

Fig. 14: Snippet of the unit proof for `aws_hash_callback_string_eq`.

### A.5 Pre-condition bug mixing different kinds of strings

*File.* `verification/cbmc/proofs/aws_hash_callback_string_eq/hash_callback_string_eq_harness.c`

*Description.* The `aws-c-common` library supports two types of strings. The zero-terminated strings from LibC that are common in C, and its own data structures `aws_string`. The harness shown in **??** assumes that the `aws_string` object satisfies the C-string representation invariant. The representation invariant for C-strings happens to be very weak (essentially that the pointer is not null). According to the C semantics, conversion from a data structure to a C-string is always possible.

*Story.* The bug was found during compilation in an IDE. Compilation warnings indicated that an explicit cast is desirable for the calls to `aws_c_string_is_valid`. It is clear that the intended specification is to assume representation invariant for `aws_string` instead.

### A.6 Specification stub of `hash_iter_delete`

*File.* `verification/cbmc/stubs/aws_hash_iter_overrides.c`

```
1  void aws_hash_iter_delete(struct aws_hash_iter *iter,
2                            bool destroy_contents) {
3      /* Build a nondet iterator, set the required fields,
4         and copy it over */
5      struct aws_hash_iter rval;
6      rval.map = iter->map;
7      rval.slot = iter->slot;
8      rval.limit = iter->limit - 1;
9      rval.status = AWS_HASH_ITER_STATUS_DELETE_CALLED;
10     // rval.map->p_impl->entry_count--;
11     *iter = rval;
12 }
```

Fig. 15: Specification stub of `hash_iter_delete`.

*Description.* A bug in *specification stub* that we have found in this case study is in the stubbed version of `hash_iter_delete`. The code is shown in **??**, and the revision is shown in the comment at line 8. After deleting an item from the iterator, the specification stub did not update the `hash_table_state` of the hash table properly since the stub did not decrease the `entry_count` field of it.

*Story.* We found the bug in an attempt to use original functions over specification stubs in order to build a more precise proof of `hash_table_foreach`. The function `hash_table_foreach` leverages `hash_iter_begin`, `hash_iter_next` and `hash_iter_done` to iterate over all entries in an `aws_hash_table`. In each iteration, `hash_iter_delete` is conditionally invoked to delete an entry from the table.

When we used the original `hash_iter*` functions in the SeaHorn unit proof of `hash_iter_foreach`, we found that the representation invariant of `aws_hash_table` could become false in the post-condition of `hash_iter_foreach`. A deeper look at the counterexample trace revealed that the `entry_count` field became a negative number after a particular execution of `hash_iter_foreach`.

The reason is two-fold. Firstly, there are two ways in `aws-c-common` hash table implementation to determine the number of entries in `aws_hash_table`. The first one is to explicitly look up the field `entry_count` and the second one is to implicitly infer it from the number of allocated entries that has a non-zero hash code. The original implementation of `hash_iter*` function often use the latter way. Secondly, the CBMC proof non-deterministically initializes the memory that contains allocated hash table entry slots, but it does not set any constraint on the number of allocated entries with non-zero hash codes.

When we tried to use original `aws-c-common` implementations of `hash_iter*` functions in combination of the initialization process from CBMC proofs, it is possible to create an `aws_hash_table` with more entries with non-zero hash codes than `entry_count`, in which case `hash_iter_delete` could be invoked more times than `entry_count` and decrease the field to a negative number.

The aforementioned behavior constitutes a bug in the pre-condition part of *CaS*, which stems from a bug in a specification stub of key functions invoked by the function under verification.

```c
1  /**
2   * Returns whether each byte is zero.
3   */
4  AWS_STATIC_IMPL
5  bool aws_is_mem_zeroed(const void *buf, size_t bufsize) {
6    const uint64_t *buf_u64 = (const uint64_t *)buf;
7    const size_t num_u64_checks = bufsize / 8;
8    size_t i;
9    /* uint64_t val; */
10   for (i = 0; i < num_u64_checks; ++i) {
11     /* memcpy(&val, &buf_u64[i], sizeof(val)); */
12     if (buf_u64[i] /* val */) {
13       return false;
14     }
15   }
16   ...
17 }
```

Fig. 16: A snippet of aws_is_mem_zeroed.

### A.7 Undefined behaviour in aws_is_mem_zeroed

*File.* include/aws/common/zero.inl

*Description.* A snippet of a buggy implementation of aws_is_mem_zeroed is shown in **??**. The function can return false even when the memory pointed by buf is filled with zeroes. The bug in the memory access in line 12. The buffer buf is accessed using a pointer to uint64_t. This access is only defined if buf was previously written with the same type. The code violates the strict aliasing rule of C. A suggested fix is given in comments. We do not know whether the bug manifests in the production code. The TBAA alias analysis of LLVM is strong enough to commute read at line 12 with prior writes to buf.

*Story.* In aws-c-common, the function is called through a macro expansion of AWS_IS_ZEROED. The macro is used in unit proofs and in the production code. In CBMC mode, the macro expands to CBMC specific implementation. Thus, the code of aws_is_mem_zeroed is never seen by CBMC. In SeaHorn, we analyze the code as it is compiled and this function is used. The bug in the function causes a post-condition to be violated and counterexample is generated. Manually examining the counterexample identified the bug.

## B Empirical Results

*Background.* In the experiments, for CBMC we use the original unit proof available at aws-c-common GitHub repository[8]. SeaHorn and KLEE use our adapted unit proofs and our implementation of verification library, including implementation of nd_<type> functions and initialization helpers. Obviously, CBMC uses its internal compiler and pre-processing, while other tools rely on Clang compiler of LLVM. For CBMC, we measure its total time. For SeaHorn and KLEE the compilation time is excluded because files are pre-compiled. The

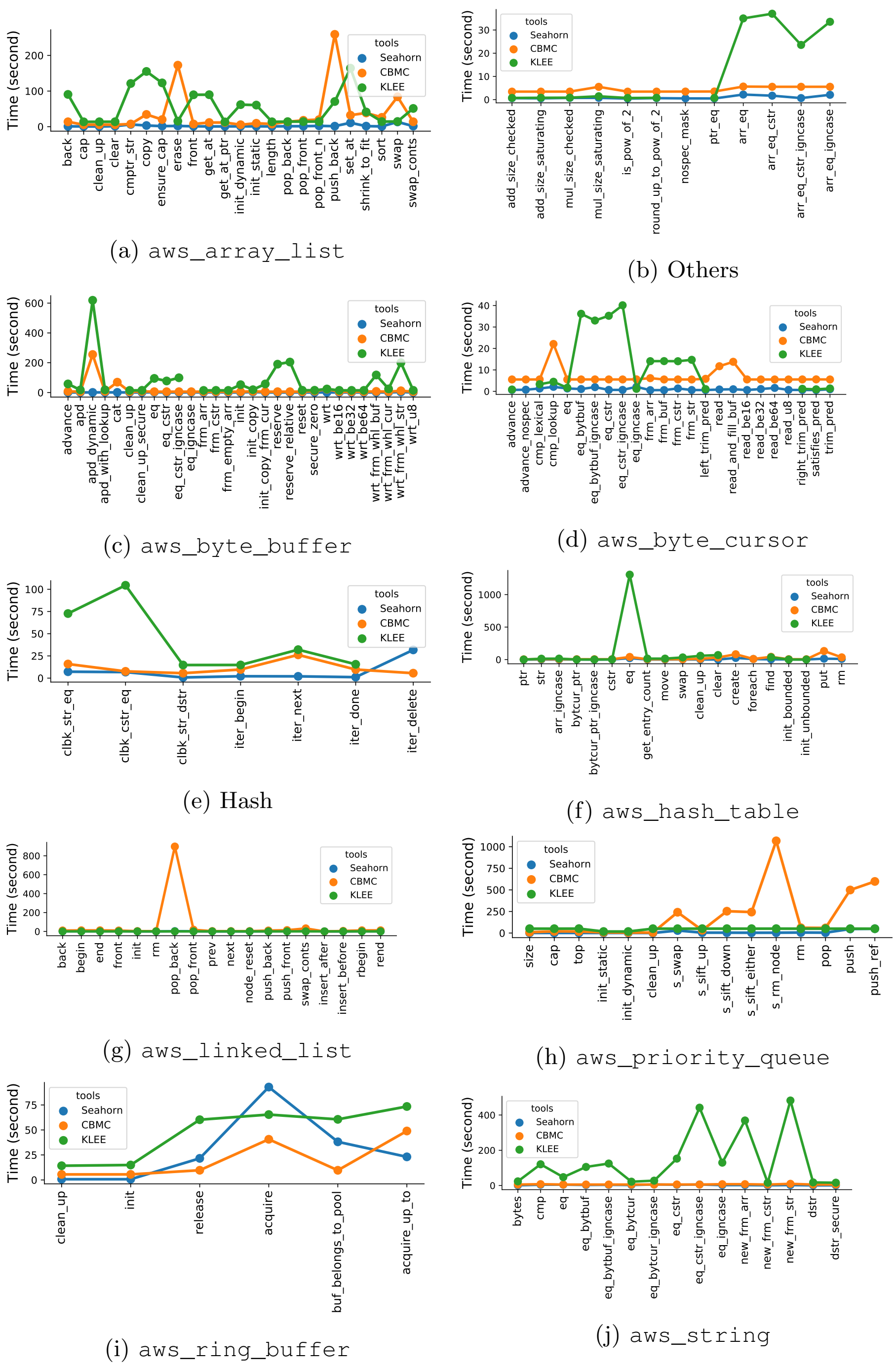
[8] https://github.com/awslabs/aws-c-common/tree/main/verification/cbmc/proofs.

compilation time is insignificant. Finally, for KLEE we had to severely restrict the sizes of allocated memory.

For `libFuzzer`, we do not show the running times because the most important result is coverage report. The details of workflow is shown in Fig. 3.

*Description.* The results are shown in **??**. They are divided into categories. Each graph shows unit proofs on the $x$-axis and verification time on the $y$-axis. The category *Hash* combines `hash_callback` and `hash_inter`, and *Others* combines `arithmetic` and `array`.

Note that our performance comparison must be taken with a grain of salt. We have focused on developing a methodology to adapt the unit proofs to all the different tools. In some cases, this required significant change to what is being verified, both in terms of sizes of data structures considered, and in the properties checked (especially for built-in checks, such as memory safety). In our experience, all tools produced useful results. In particular, all the violations in the specifications have been found using SeaHorn. However, these differences might have a significant effect on the running time. We leave a more in-depth study of performance of different techniques on the *same* verification problems for future work.




(a) `aws_array_list` (b) Others

(c) `aws_byte_buffer` (d) `aws_byte_cursor`

(e) Hash (f) `aws_hash_table`

(g) `aws_linked_list` (h) `aws_priority_queue`

(i) `aws_ring_buffer` (j) `aws_string`

Fig. 17: Running times for CBMC, SeaHorn, and KLEE on `aws-c-common`.