Received: 27 April 2020 | Revised: 19 October 2020 | Accepted: 20 October 2020

DOI: 10.1002/spe.2949

SPECIAL ISSUE PAPER

WILEY

Code-level model checking in the software development workflow at Amazon Web Services

Nathan Chong[1] | Byron Cook[1,2] | Jonathan Eidelman[1] | Konstantinos Kallas[3] | Kareem Khazem[1] | Felipe R. Monteiro[1] | Daniel Schwartz-Narbonne[1] | Serdar Tasiran[1] | Michael Tautschnig[1,4] | Mark R. Tuttle[1]

[1]Amazon, New York City, New York, USA

[2]University College London, London, UK

[3]University of Pennsylvania, Philadelphia, Pennsylvania, USA

[4]Queen Mary University of London, London, UK

Correspondence
Daniel Schwartz-Narbonne, Amazon, New York City, NY, USA.
Email: dsn@amazon.com

Abstract

This article describes a style of applying symbolic model checking developed over the course of four years at Amazon Web Services (AWS). Lessons learned are drawn from proving properties of numerous C-based systems, for example, custom hypervisors, encryption code, boot loaders, and an IoT operating system. Using our methodology, we find that we can prove the correctness of industrial low-level C-based systems with reasonable effort and predictability. Furthermore, AWS developers are increasingly writing their own formal specifications. As part of this effort, we have developed a CI system that allows integration of the proofs into standard development workflows and extended the proof tools to provide better feedback to users. All proofs discussed in this article are publicly available on GitHub.

KEYWORDS

continuous integration, model checking, memory safety

1 | INTRODUCTION

This is a report on making code-level proof via model checking a routine part of the software development workflow in a large industrial organization. Formal verification of source code can have a significant positive impact on the quality of industrial code. In particular, formal specification of code provides precise, machine-checked documentation for developers and consumers of a code base. It improves code quality by ensuring that the program's *implementation* reflects the developer's *intent*. Unlike testing, which can only validate code against a set of concrete inputs, formal proof can assure that the code is both secure and correct for all possible inputs.

Unfortunately, rapid proof development is difficult in cases where proofs are written by a separate specialized team and not the software developers themselves. This organization frequently occurs in practice, and is in fact recommended for safety critical software such as avionics.[1] The developer writing a piece of code has an internal mental model of their code that explains why, and under what conditions, it is correct. However, this model typically remains known only to the developer. At best, it may be partially captured through informal code comments and design documents. As a result, the proof team must spend significant effort to reconstruct the formal specification of the code they are verifying. This slows the process of developing proofs.

Over the course of 4 years developing code-level proofs in Amazon Web Services (AWS),[2-5] we have developed a proof methodology that allows us to produce proofs with reasonable and predictable effort. For example, using these techniques, one full-time verification engineer and two interns were able to specify and verify 171 entry points over nine key modules in the AWS C Common[1] library over a period of 24 weeks (see Section 3.3 for a more detailed description of this library). All specifications, proofs, and related artifacts (such as continuous integration reports) described in this article have been integrated into the main AWS C Common repository on GitHub, and are publicly available at https://github.com/awslabs/aws-c-common/. This article is a substantially revised and extended version of our work presented at ICSE 2020.[6] The major difference is the detailed description of our continuous integration architecture and its leverage to integrate proofs into the software development workflow. All tools, scripts, benchmarks, and results of our evaluation are also available on a replication package.[7]

1.1 | Methodology

Our methodology has four key elements, all of which focus on communicating with the development team using artifacts that fit their existing development practices. We find that of the many different ways we have approached verification engagements, this combination of techniques has most deeply involved software developers in the proof creation and maintenance process. In particular, developers have begun to write formal functional specifications for code as they develop it. Initially, this involved the development team asking the verification team to assist them in writing specifications for new features. Increasingly, the development team has been writing formal specifications themselves and adding them to their code base. In this article, we describe the principal reasons our relationships with software development teams have been so positive. The lessons we have learned (re-)emphasize that the human factors and social process of software development are as important as the technical aspects of formal verification when it comes to the adoption and integration of code-level proofs in industry. These lessons are:

Make specifications explicit in source code. Tying the specification directly to the source code helps the developers understand what has been proven. As much as possible, formal specifications should follow the idioms and style that the development team is familiar with. Although there are properties which can be difficult to specify using standard coding idioms (such as properties involving temporal logic or separation logic), we have found that in practice the benefit of using idioms developers understand outweighs the potential loss of expressive power. Formal specifications act as documentation for library users. And they provide a systematic way for the development team to reason about the conditions we are proving, and to clearly see whether they are consistent across the code base.

In our experience, the best way to include specifications is by adding them as precondition and postcondition assertions directly in the code base. Making specifications explicit in the code helps to ensure that they remain accurate as the code is updated, and validates the use of specification assumptions in proofs. In particular, adding specifications as runtime-assertions in the code allows developers to interact with the specifications using the same tests they are already familiar with (Section 4.3).

Write proof harnesses in declarative style. Unit-test-like proof-harnesses provide the development team with a familiar conceptual model when entering the world of proofs. They also help the verification team: having a recipe for how to write proofs means that new team members with little familiarity are able to write high-quality proofs within weeks. Our methodology has improved to the point where one new member was able to prove 1500 lines of embedded operating system code in a month, in contrast to 2 years ago when an expert took 2 months to verify 700 lines of firmware. The declarative style made it easy for us to audit the proofs produced before submitting them to the development team (Section 4.1).

Integrate proof artifacts into the development workflow. Making proof artifacts part of the regular workflow decreases developers' cognitive burden and allows them to treat our proofs as "just another test suite," albeit a vastly more thorough one. In particular:

- we merge the proofs into the target code base, such that they become part of the source distribution;
- as part of this merge, we ask the developers to review the proofs like any other code contribution;
- once merged, the proofs are routinely checked along with all other checks in continuous integration; and

[1]https://github.com/awslabs/aws-c-common

- our continuous checking system has low latency and is highly reliable.

This provides value to customers and the developer teams by guaranteeing that the code remains correct as the code changes (Section 5).

Fix bugs instead of just reporting them. Providing bug-fixes instead of bug-reports saves the development team effort, and gets fixes to customers faster. We discovered that it also saves time for the verification team, first, by reducing the communication overhead involved in the bug report, and, second, by enabling immediate proof for the fixed code while it is still fresh in the finder's mind. Most important, delivering bug fixes engenders trust from the developers, who become more responsive and receptive to the proof writers' feedback (Section 4.2).

1.2 | Results

Our experience using the proof methodology has been:

Increased proof speed. Our data shows that our proof development has indeed accelerated as a result of our method. Using these techniques, one full-time verification engineer and two interns, were able to specify and verify 171 entry points over 9 modules in the AWS C Common library over a period of 24 weeks (see Section 3.3 for a more detailed description of this library). As we refined our methodology, our proof productivity increased, as shown by the number of lines of code proven over time in Figure 1. The flattening that occurs at the end of August represents us having reached our verification target. Afterward, further verification only occurs on an as-needed basis.

Increased rate of bugs found and fixed. As part of this effort, we found and fixed 83 issues (see Section 4.2, including Table 1, for details). Our rate of finding bugs increased as we refined our methodology (see Figure 2). As above, the flattening that occurs at the end of August represents us having reached our verification target. And because we provided patches, not just bug reports, bugs were fixed quickly—the median time from bug report to fix being merged was 5 days.
Active developer engagement with proofs. Developers took an active role in reviewing both proofs and specifications, as witnessed by their comments on GitHub pull requests that introduced new proofs (see Figure 5 in Section 5).

Increase in lines of specifications written by developers. Formal specifications written by developers is a key metric of success. By the time we reached our verification target in August, developers had added 89 contracts to their code (see Figure 4 in Section 4.3). Section 4.3 explains why our style of writing code contracts has been easy for developers to adopt.

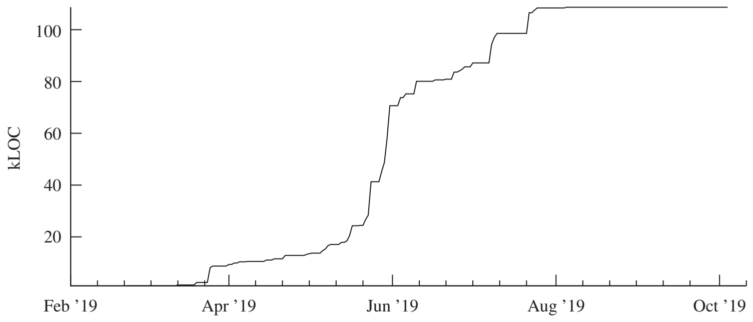
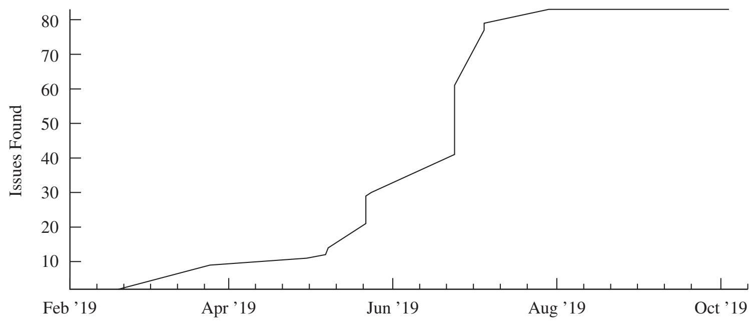


FIGURE 1 Cumulative number of LOC proven

TABLE 1 Severity and root cause of issues found

Root cause	# issues	Severity: High	Medium	Low
Integer overflow	10 (12%)	2	8	0
Null-pointer deref.	57 (69%)	0	14	43
Functional	11 (13%)	0	4	7
Memory safety	5 (6%)	0	5	0
Total	83	2	31	50
		(3%)	(37%)	(60%)

FIGURE 2 Cumulative number of issues found



2 | RELATED WORK

There are a number of powerful static analysis tools such as Infer[8] and Coverity.[9] These are highly effective bug-hunting tools, but they do not prove that a program satisfies a specification. This article is about proof, and drawing developers into the proof process. Similarly, there is a body of work about how to organize the process of writing proofs by formal verification experts, such as Aagaard et al.,[10] but this article focuses on improving the interaction between formal verification and development teams.

There has been significant work on proving conformance to specifications. For example, Chudnov et al.[11] demonstrate the conformance of the s2n HMAC to a formal HMAC specification. In our world, the formal specification does not exist: this article is about a methodology to efficiently extract a formal specification for an existing implementation.

Over the last 16 years, automated reasoning techniques (e.g., model checking) have evolved significantly.[12,13] As a consequence, several software verification frameworks have emerged in the literature;[14] with many applications in the industrial setting.[15-20] Some papers[21-23] describe how human factors impact the adoption and integration of formal verification techniques into well-established software engineering process. Human factors played a significant role in our experience as well.

Ball et al.[24] described many pitfalls faced by Microsoft as they introduced the static driver verifier (SDV) in the development process of device drivers. Similar to our experience, the report recognizes social factors (such as identifying champions and management buy-in) in addition to technical decisions as important to the successful adoption of SDV by Windows device driver developers. Sadowski et al.[20] present how Google uses static analysis tools and highlight two important aspects: static analysis authors should focus on feedback from developers as well as carefully consider workflow integration as it is key for adoption. Both points are confirmed by our case study as discussed in Section 4. O'Hearn et al.[25] describe continuous reasoning as a concept of performing static analyses at every change during the development process at Facebook. He characterizes the use of other, more heavyweight, formal techniques (e.g., bounded model checking) in similar continuous settings as an open scientific challenge. O'Hearn also highlights that reporting static analysis results during code review is crucial to achieving a higher fix rate,[19] which is reaffirmed by our work (cf. Section 5).

The idea that unit test harnesses are well-understood by developers and are therefore a base from which to build-upon is used by parameterized unit tests.[26] In this approach, unit tests are parameterized so that they become algebraic specifications amenable to analysis using randomized testing or dynamic symbolic execution. Unlike our methodology aimed at code-level proof, the aim of parameterized unit tests is to amplify unit tests with better coverage through automatic test generation.

Annotation languages for static analyzers such as Microsoft's Simple Annotation Language (SAL)[27] and ESC/Java[28] are designed for lightweight static checking and to make annotated code more understandable, both for humans and code analysis tools. Similar to our choice regarding annotations, the annotation language for ESC/Java was designed to be as close to the source-language (Java) as possible for ease-of-learning and readability. Both tools operate at compile-time of the code, which is advantageous for being more tightly integrated into the workflow of a developer than in continuous integration. The use of model checking in our methodology means that we trade off this tight feedback for more heavyweight analysis.

3 | BACKGROUND

The focus of our work is the foundational security of software running in AWS data centers, SDKs, and devices. The breadth of our work includes boot code,[29] network communication protocols,[30] real-time operating system code,[31,32] an SDK implementing data structures,[6] and an SDK facilitating principled use of cryptographic primitives.

3.1 | CBMC and program verification

We verify code using CBMC,[33] a bit-precise bounded model checker for C. CBMC is a bounded model checker, but we always run CBMC with the `–unwinding-assertion` flag which ensures that we have fully unwound loops and fully checked the model. Given a C program and loop unwinding instructions, CBMC constructs a Boolean satisfiability (SAT) formula that is satisfiable if and only if an assertion violation is reachable from the entry point of the program.

CBMC then uses a SAT solver such as MiniSat[34] to compute the satisfiability of the formula. If the SAT solver returns "satisfiable," then the assertion can be violated, and CBMC generates an error trace from the model returned by the SAT solver. If the SAT solver returns "unsatisfiable," this provides mathematical proof that no assertion can be violated in the given program up to the assumptions used for the analysis. The third possibility is that CBMC (or the SAT solver) runs out of time or memory, and no result is returned.

If the SAT formula is found to be unsatisfiable, the absence of assertion violations is guaranteed to hold for all possible executions starting at the program entry point. This contrasts with traditional testing, where the result of a test may only apply for the concrete inputs used: the success of a unit test on a set of inputs provides no guarantee about whether there are other inputs on which the test would fail.

3.1.1 | Input to the model checker

CBMC input can be divided into two parts: the code being verified, and a proof harness. The code being verified is simply compiled using the goto-cc compiler from the CBMC toolset. This compiler is a drop-in replacement for gcc, so compiling the code being verified is a simple matter of adding the right ".c" files to the proof Makefile. In addition, as we discuss in Section 4, each proof requires a proof harness. The main challenges to writing a proof harness are: (i) stating the assumptions we make about the inputs to the entry point being verified, (ii) computing the number of loop unwindings required for inputs satisfying our assumptions, and (iii) using techniques, such as modular verification and bounding inputs sizes, to address cases where the solver runs out of time or memory.

3.1.2 | Scalability of model checking

The runtime and memory usage of CBMC can vary by orders of magnitude between different proofs. Proof runtime is subject to multiple non-linear influences, and predicting the time needed for model-checking is an open research question. We have observed that proof runtime tends to scale with:

1. The complexity of the code being verified—more complex code takes CBMC longer to verify. Complexity is often more a function of the features of the code (e.g., non-linearity, complexity of data structures, etc.) than the raw number of lines of code (LOC). For example, a function with multiple nested loops, or a complex branching structure, may have high complexity with few LOC, while straightline code doing simple arithmetic operations may have higher LOC, but lower complexity. We can mitigate the effect of code complexity on proof runtime using modular verification.
2. The size of the data-structures being used. Larger data-structures tend to lead to larger, more complex SAT formulas. And they more loop unrollings to process, which increases the amount of work the solver must do. We can mitigate this by bounding data-structure sizes.

Using these techniques, we have been able to complete proofs that reliably complete in a reasonable amount of time. For the AWS C Common repository discussed in this article, the median proof takes 18 seconds, with an average of 216 seconds, and a standard deviation of 622.

3.1.3 | Why we chose CBMC

Code-level verification involves an inherent trade-off between the human effort required to perform the verification, and the quality of the results. At one end are static analyzers, such as Clang Static Analyzer.[2]These tools require very little human intervention (although they can benefit from user annotations), and typically produce results quickly, but can

[2]https://clang-analyzer.llvm.org/

also provide imprecise results (i.e., both false positives, and false negatives). They are useful for finding bugs, but do not provide the guarantees of formal proof.

At the other extreme, deductive tools such as Frama-C[35] and VeriFast[36] provide strong guarantees about program correctness, but require the user to provide specifications for every function and loop in the program. These specifications are often not included inline in the code as standard C, but annotated using specialized languages, such as separation logic or linear-temporal logic (LTL). In practice, these specifications can be larger and more complex than the code that is actually verified (see, e.g., the experiences of sel4[37] and IronFleet[38]).

In our experience, model-checkers such as CBMC occupy a sweet spot in the middle that fits well with our methodology. CBMC has a bit-precise formal analysis, which allows us to *prove* the absence of certain classes of bugs. As we discuss in Section 4.1, CBMC harnesses and annotations are both written in C, which makes them easier for developers to write, and allows them to be checked as runtime assertions. Failing proofs provide a concrete error trace that developers can debug.

There are a number of tools that provide similar facilities to CBMC, such as SeaHorn,[39] ESBMC,[40] and SMACK.[41] We focused on CBMC due to our team's preexisting familiarity with CBMC, but we believe these other tools would also fit well into our verification methodology.

3.2 | Verification targets

We have applied this methodology to a number of AWS open-source projects. For each verification engagement, we have worked with the development teams to identify a priority order of modules for verification. In this article, we focus on the *AWS C Common* library, but our experiences are similar across these projects.

AWS C Common. AWS C Common[3]is an open-source C99 package (licensed under Apache 2.0) that provides cross-platform configuration, data structures, and error handling support for C code. It is the foundation of other AWS libraries, including the AWS Encryption SDK for C, and verifying its security is a critical first step toward ensuring the security of libraries that depend on it. In this work, we focus on the AWS C Common modules used by the AWS Encryption SDK for C. We prove the memory safety of the AWS C Common functions that account for 98% of function calls into AWS C Common by the AWS Encryption SDK for C (the remaining 2% use concurrent features that are outside the scope of our current verification tools).

FreeRTOS. FreeRTOS[4]is an open source, real-time operating system for microcontrollers that makes small, low-power edge devices easy to program, deploy, secure, connect, and manage. Distributed freely under the MIT open source license, FreeRTOS includes a kernel and a growing set of software libraries suitable for use across industry sectors and applications. This includes securely connecting small, low-power devices to AWS cloud services like AWS IoT Core or to more powerful edge devices running AWS IoT Greengrass.

s2n. s2n[5]is a C99 implementation of the TLS/SSL protocols Amazon, including S3, and AWS services.[30] It is that is designed to be simple, small, fast, and with security as a priority. s2n is released and licensed under the Apache License 2.0.

AWS Encryption SDK for C. The AWS Encryption SDK for C[6]is a client-side encryption library designed to make it easy for everyone to encrypt and decrypt data using industry standards and best practices. It uses a data format compatible with the AWS Encryption SDKs in other languages. AWS Encryption SDK for C is released and licensed under the Apache License 2.0.

3.3 | Properties verified

We check the validity of all user-defined properties (i.e., `assert` statements). These properties can reflect function contracts (i.e., pre/post-conditions), internal sanity checks, and additional functional checks added in the verification harness. More detailed descriptions of how we use write and use assertions can be found in Section 4. In addition, all verified modules have been checked for a standard set of C undefined behaviors. The most important of these

[3]https://github.com/awslabs/aws-c-common
[4]https://github.com/FreeRTOS/FreeRTOS
[5]https://github.com/awslabs/s2n
[6]https://github.com/aws/aws-encryption-sdk-c

are checks for memory-safety violations: Memory safety errors are routinely listed among the most critical security concerns by industry groups monitoring CVEs.[42-45] In addition to memory safety, our proofs guarantee the absence of a subset of undefined C behavior[46] like division-by-zero and arithmetic overflow. We use the following set of CBMC flags, which automatically add checks which assert the absence of undefined C behavior: array bounds checks (`–bounds-check`), values correctly represented after type cast checks (`–conversion-check`), division by zero checks (`–div-by-zero-check`), floating-point for $+/-\infty$ checks (`–float-overflow-check`), floating-point NaN checks (`–nan-check`), search for NULL-pointer dereferences or dereferences of other invalid pointers (`–pointer-check`), pointer arithmetic over- and underflow checks (`–pointer-overflow-check`), all pointers in CPROVER pointer primitive functions are valid or null check (`–pointer-primitive-check`), signed arithmetic over- and underflow checks (`–signed-overflow-check`), unsigned arithmetic over- and underflow checks (`–unsigned-overflow-check`), and range checks for shift distances (`–undefined-shift-check`). Finally, we automatically add checks that ensure that all loops are fully unwound—up to the limited size of the data-structures, discussed below (`–unwinding-assertions`).

4 | METHODOLOGY

This section describes the four pillars of our methodology, and how they support successful interaction with developers. We believe that proofs, invariants, and code contracts need not be arcane and inscrutable; we actively strive to ensure that both our proofs and the development process that surrounds them blend in seamlessly with developers' own code and processes. While the idea that program proofs are "nice to have" is uncontroversial, we believe that developers' enthusiastic and reciprocal involvement with our work is due in great part to our adoption of this methodology.

Running example

In AWS C Common, an array list is a polymorphic variable-length array, which dynamically grows as elements are added to it.

```c
/* Resizable array implementation. */
struct aws_array_list {
    struct aws_allocator *alloc;
    size_t current_size;
    size_t length;
    size_t item_size;
    void *data;
};
```

Here, `alloc` represents the allocator used by the list (to allow consumers of the list to override `malloc` if desired), `current_size` represents the bytes of memory that the array has allocated, `length` is the number of items that it contains, `data_size` represents the size of the objects stored in the list (in bytes), and `data` points to a byte array in memory that contains the data of the array list.

Users of this data structure are expected to access its fields using getter and setter methods, although C does not offer language support to ensure that they do so. Similarly, since the C type system does not have support for polymorphism, authors of the getters and setters are responsible for ensuring that the list is accessed safely. For example, here is the getter for `array_list` (notice how it ensures memory safety):

```c
/**
 * Copies the memory address of the element at index to *val.
 * If element does not exist, AWS_ERROR_INVALID_INDEX will be raised.
 */
int aws_array_list_get_at_ptr(
      const struct aws_array_list *list,
      void **val,
      size_t index)
{
  if (aws_array_list_length(list) > index) {
      *val = (void *)((uint8_t *)list->data + (list->item_size * index));
      return AWS_OP_SUCCESS;
  }
  return aws_raise_error(AWS_ERROR_INVALID_INDEX);
}
```

4.1 | Proof style

Our proofs have the following features:

- They are structured as *harnesses* that call into the function being verified, similar to unit tests. This makes it easy to see how they work, as developers can "execute" the proof in their heads. This style also yields more useful error traces.
- They state their assumptions declaratively. Rather than creating a fully-initialized data structure in imperative style (as in Reference 47), we create unconstrained data structures and then constrain them just enough to prove the property of interest. This means the only assumptions on the data structure's values are the ones we state in the harness.
- They follow a predictable pattern: setting up data structures, assuming preconditions on them, calling into the code being verified, and asserting postconditions.

The following code is an example of a proof harness:

```c
/* CBMC-proof harness for aws_array_list_get_at_ptr function. */
void aws_array_list_get_at_ptr_harness()
{
    /* initialization */
    struct aws_array_list list;
    __CPROVER_assume(aws_array_list_is_bounded(&list));
    ensure_array_list_has_allocated_data_member(&list);

    /* generate unconstrained inputs */
    void **val = can_fail_malloc(sizeof(void *));
    size_t index;

    /* preconditions */
    __CPROVER_assume(aws_array_list_is_valid(&list));
    __CPROVER_assume(val != NULL);

    /* call function under verification */
    if(!aws_array_list_get_at_ptr(&list, val, index)) {
      /* If aws_array_list_get_at_ptr is successful,
       * i.e. ret==0, we ensure the list isn't
       * empty and index is within bounds */
        assert(list.data != NULL);
        assert(list.length > index);
    }

    /* postconditions */
    assert(aws_array_list_is_valid(&list));
    assert(val != NULL);
}
```

The harness shown above consists of five parts:

1. Initialize the data structure to unconstrained values. We write a function that allocates memory for the data structure filled with unconstrained values. We name the function
   `ensure_data_structure_has_allocated_data_member()`.
2. Generate unconstrained inputs to the function.
3. Constrain all inputs to meet the function specification and assume all preconditions using `assume`statements. If necessary, bound the data structures so that the proof terminates.
4. Call the function under verification with these inputs.
5. Check any function postconditions using `assert` statements.

This style of writing a proof harness is motivated by our desire to make assumptions explicit to developers. This style consists of two steps. The first step does the minimal work required to imperatively allocate structures with unconstrained fields, as described in Items (1) and (2) in the above list. The second step uses `assume` statements to enforce the specification about the values that go in those fields (Item (3)). This makes the specification used in the proof harness clear and allows them to be further reused as assertions in the mainline code (cf. Section 4.3).

Syntactically, a proof harness looks quite similar to a unit test. The main difference is that a proof harness calls the target function with a partially-constrained input rather than a concrete value; when symbolically executed by CBMC, this has the effect of exploring the function under *all* possible inputs that satisfy the constraints.

In fact, historically, we started from unit tests, and tried to make them symbolic by replacing concrete values with unconstrained values. We found this difficult, since there are relations that constrain fields in a data structure and must be enforced (e.g., $\mathtt{length} < \mathtt{capacity}$ and $\mathtt{capacity} \neq 0 \Rightarrow \mathtt{buffer} \neq \mathtt{NULL}$). Even worse, these imperative proof-harnesses turned out to be difficult to reason about and to explain to the development team.

The preconditions used as assumptions in (Item (3)) are developed using an iterative process. For each module, we start by specifying the simplest predicates that we can think of for the data structure—usually, that the data of the data structure is correctly allocated. Then we gradually refine these predicates, until the development team accepts them as reasonable invariants for the data structure, aided by having all the unit and regression tests pass.

Using this process, we defined a set of predicates for each data structure in the C source file so that they can be easily accessed and modified by the library developers, and so that they serve as documentation for the library's users. For instance, in the case of the `array_list`, we started with the invariant that `data` points to `current_size` allocated bytes. After several iterations, the validity invariant for `array_list` ended up looking like this:

```c
/* Invariants for aws_array_list data structure. */
bool aws_array_list_is_valid(const struct aws_array_list *list)
{
  /* Object must be valid. */
  if (list != NULL) return false;

  /* Length and item size must not lead to overflows. */
  size_t required_size = 0;
  bool required_size_is_valid =
           (aws_mul_size_checked(list->length,
                                 list->item_size,
                                 &required_size)
            == AWS_OP_SUCCESS);

  /* Current size must be enough to store all elements. */
  bool current_size_is_valid =
           (list->current_size >= required_size);

  /* Underlying pointer must have current_size allocated positions. */
  bool data_is_valid =
           ((list->current_size == 0 && list->data == NULL)
           || AWS_MEM_IS_WRITABLE(list->data, list->current_size));

  /* Item size must not be zero. */
  bool item_size_is_valid = (list->item_size != 0);

  return required_size_is_valid
         && current_size_is_valid
         && data_is_valid && item_size_is_valid;
}
```

The invariant above describes four conditions satisfied by a valid `array_list`:

1. the sum of the sizes of the items of the list must fit in an unsigned integer of type `size_t`, which is checked using the function `aws_mul_size_checked` (see Section 4.2.2 for a discussion of the integer overflow issue this addresses);
2. the size of the `array_list` in bytes (`current_size`) has to be larger than or equal to the sum of the sizes of its items;
3. the `data` pointer must point to a valid memory location, or must be `NULL` if the size of the `array_list` is zero;
4. the `item_size` must be positive.

Item 3 stemmed from a protracted discussion with the developers. Some members of the team felt that in the case of a zero-length array, the value of the pointer was irrelevant; others felt equally strongly that a un-allocated buffer must be `NULL`. Having a single `is_valid` function helped in quickly converging on a consistent specification.

A significant contribution of this work is a library of allocators and validators for each data type. The availability of this library accelerated proof construction and reduced proof size. In total, there are 1.4 kLOC of helper code (with 1.1 kLOC

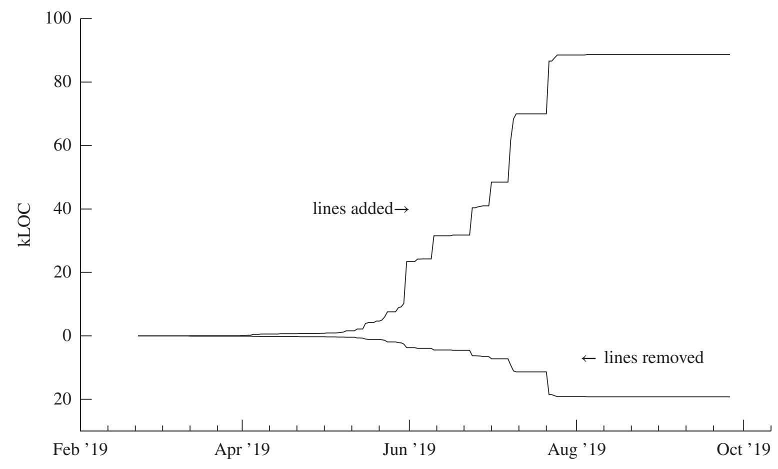


FIGURE 3 Cumulative number of non-proof LOC added to code base by the verification team

comments), supporting 3.5 kLOC of proof harness (with 3.1 kLOC comments). The average proof harness consisted of 20 LOC (standard deviation 8.4), with a similar number of lines of comments.

4.2 | Finding and fixing bugs

In our experience, high-quality bug reports are one of the most effective techniques for getting the attention of developers, and demonstrating to them the immediate value of formal code specification and proof. Formal verification has the ability to find classes of subtle bugs that can escape traditional testing. Once bugs have been found, the trace demonstrating the proof failure, together with the specification, can be instrumental in root-causing the issue. Once the bug is root-caused and the code is modified, formal verification can prove that the modification fixes the bug. The success of formal specification in finding subtle bugs is therefore a strong incentive for developers to both accept such specifications into their existing code bases, and to write new specifications as they write new features. However, the ultimate measure of formal verification is the value it provides to the customers of the library. Hence, we argue that the success of a formal verification engagement should not be evaluated based on the number of bugs *found*, but based on the correctness of the final target code, that is, measuring the number of bugs that were *fixed*.

Finding bugs is a technical process: write the harness, write the assertions, run CBMC, get the error trace, debug the error trace, confirm the existence of a bug. But fixing bugs is a much more complicated social process that can require extensive coordination between the development and verification teams, followed by significant time and effort from the development team. It is well known that bug hunting tools like Fortify[7] and Coverity[8] can generate bug reports faster than developers can fix them.[25] Instead, we reduce the overhead on the development team by *reporting bugs in the form of pull requests with manually written patches*, along with *a CBMC proof that the patch fixed the issue*. This led to the fixes being promptly applied: 100% of the issues discovered during this engagement have been fixed.

Figure 3 shows the amount of code unrelated to proof that *proof writers* contributed to the code base. This code includes bug fixes, code refactoring, and other improvements. In contrast, Figure 4 shows the amount of code that *developers* contributed for the sake of proof. This code represents function contracts the developers have added to their own code. These results show the proof writers and the software developers exchanging roles with ease. Developers improve the program's specification by adding code contracts, leveraging their deep understanding of the program's purpose and design. Proof writers improve the code by refactoring it, leveraging their insight into the code obtained in the course of writing the proof.

A formal proof integrated into continuous integration (CI) provides confidence that the fix is complete, and acts as a super-charged regression test (in fact, the development team agreed that we did not need regression tests for patches that came associated with proofs). And in the cases where there was discussion about the severity and scope of the bug, that discussion could occur guided by a concrete fix.

This focus on delivering solutions was a key factor in building trust with the development team. From the point of view of the verification team, the act of writing bug-fixes helped us understand both the code and the development

[7] https://www.microfocus.com/en-us/solutions/application-security
[8] https://scan.coverity.com

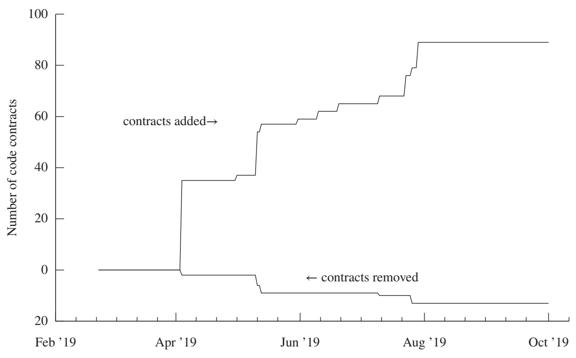


FIGURE 4 Cumulative number of function contracts the development team added to their own code base

methodology. From the point of our relationship with the development team, showing that we have taken the time to understand their code and proposing a fix, helped the developers feel at ease trusting our judgment, to the extent that several members of the verification team were granted commit privileges to the AWS C Common repository. Also notable is the fact that the third and fifth most prolific contributors to the AWS C Common code base are verification team members.

4.2.1 | Issues found

We verified 171 entry points over nine modules in the AWS C Common library. In the course of developing these proofs, we reported 83 issues to the development team. For every bug we found, we wrote a patch with a formal proof of correctness for that patch. In total, we filed 24 pull requests (some of which fixed more than one issue), 100% of which were accepted and merged by the development team. The median time from issue reporting to fix being merged was 5 days; the mean was 9 ± 10 days.

Table 1 gives a breakdown of these issues by severity and root cause. Note that although each bug was classified according to its root cause, many bugs had cross-cutting impacts. For instance, we found cases where both integer overflow and null-dereference bugs could potentially lead to memory-safety issues.

4.2.2 | Example—Integer overflows

In C, signed integer overflow represents undefined behavior; unsigned overflow, while defined, often leads to unexpected results, including the bypassing of safety checks. We discovered 10 integer overflow issues in AWS C Common. Many of these issues were subtle, and had evaded the extensive unit and integration testing used by the AWS C Common development team. For example, when an `aws_array_list` is initialized, the number of bytes required for the array is calculated by multiplying the required length of the array by the item size. If this multiplication overflows, an insufficient number of bytes may be allocated. Concretely, consider a 32-bit machine, where the user attempts to create an `aws_array_list` with length 2^{30} and item_size 2^6. After multiplication, the seemingly valid array will have an allocated size of 2^4 bytes, too small to hold even a single element!

Since C does not have a standard representing overflow-safe arithmetic, preventing integer overflows in C code can be difficult. We added a set of safe arithmetic functions, and used them throughout the code wherever CBMC reported a potential integer overflow—for example, in the `aws_array_list_is_valid()` function described in Section 4.1. These arithmetic functions were performant and safe, and we used them widely. Instead of arguing over whether a particular trace was possible, or likely, we simply fixed the problem anywhere it could occur. Once the fixes were in place, the CI system ensures that any change introducing a new integer overflow issue will trigger a proof failure and raise an alarm.

On the other hand, having proofs made it possible to identify locations where integer overflows could *never* occur, making the use of the safe functions unnecessary. For example, the specification of `aws_array_list` guarantees that integer overflow can never occur, so methods like `aws_array_list_get_at_ptr()` can safely use standard multiplication.

4.2.3 | General code improvements

The act of writing both proofs and patches for the AWS C Common code base helped turn the verification team members into AWS C Common developers. One area this surfaced was in the repeated discovery of potential code improvements during code development. For example, while verifying the `hash_table`implementation, we realized that the code would be both clearer and easier to verify if it were refactored into a set of utility functions; we did so, provided proofs of correctness of the new functions, and had the changes merged. In another case, we noticed code that performed a `malloc`followed by a `memset(0)`, which we replaced with a clearer (and potentially more performant) `calloc`.

An unexpected win from code-refactoring came from the treatment of `static inline` functions, which are used extensively in AWS C Common, but caused issues when we were building our proofs. We refactored the functions into `.inl` files, and added pre-processor directives which controlled whether the functions within these files would be treated as `static inline`, or have normal C linkage. Although this change was merely intended to regularize the module structure of AWS C Common, and simplify our build process, we were recently informed by the development team that it turned out to be critical to a workaround for a gcc 4.8 bug that was preventing them from compiling the code on older versions of Linux.

4.3 | Function contracts

Function contracts are distinct from program proofs: they are embedded in the code base itself, and express the developers' expectations about the function's pre- and postconditions. Our function contracts are written in C, ensuring that developers can easily understand them. Figure 4 demonstrates that developers find these contracts valuable: following our lead, developers started adding contracts to their own code as part of the normal development process.

Our proof methodology helps to clearly state the specifications about the environment of a function. As Section 4.1 describes, these specifications mostly refer to the well-formedness of input arguments and expectations for the value of the global state. Running a proof harness checks that a function satisfies the harness assertions, given that the specification holds. This means that if the specification in the proof harness is too strong, that is, there are cases where the caller of the function does not satisfy them, the proof does not hold. This is a common source of bugs, even in previously verified systems.[48]

Specifications have to be scrutinized to ensure that they are realistic. Both the developers and the verification team check the specifications in the proof harnesses using careful code reviews, which do increase confidence in them, but do not eliminate all doubt, since there is still a window of human error. To tackle this issue, we embed the specification for each function in the code, in the form of assertions written in C. Explicitly annotating these specifications as `AWS_PRECONDITION` and `AWS_POSTCONDITION` rather than simple `assert` statements helps distinguish function contracts from internal error-checking assertions.

```c
int aws_array_list_get_at_ptr(
        const struct aws_array_list* list,
        void **val,
        size_t index)
{
    /* Contracts. */
    AWS_PRECONDITION(aws_array_list_is_valid(list));
    AWS_PRECONDITION(val != NULL);

    if (aws_array_list_length(list) > index) {
        *val = (void *)((uint8_t *)list->data +
                        (list->item_size * index));
        AWS_POSTCONDITION(aws_array_list_is_valid(list));
        return AWS_OP_SUCCESS;
    }

    /* Contracts. */
    AWS_POSTCONDITION(aws_array_list_is_valid(list));
    return aws_raise_error(AWS_ERROR_INVALID_INDEX);
}
```

The first specification requires that the input list satisfies the validity invariant and the second specification requires that the `val` points to an allocated object. Each specification is turned into an assertion when the tests are run and when the code is executed in debugging mode. The function also contains assertions about its postconditions, that is, it preserves the validity invariant of the input list. This increases our confidence about the validity of the specifications, as they are checked by all tests of the library, as well as in the tests of other downstream projects that depend on the target library. For instance, we were able to detect inconsistent test cases at the AWS C IO project[9](later confirmed by the developers) through the insertion of pre- and postconditions in AWS C Common.

Furthermore, predicates are also implemented as Boolean functions in the source code, which are all checked using standard assertions in plain C. Writing the specifications and predicates using the same language adopted in the project enabled developers to get more involved in the proof process, as they do not have to learn a new specification language. Although properties written in C are more verbose, and potentially less expressive than custom contract languages, this is a trade-off that leads to more developer engagement.

Note that the preconditions generated by this methodology are not necessarily mathematically minimal, the postconditions are not necessarily mathematically maximal, and we should expect them to be neither minimal nor maximal. For instance, consider a function `clone_foo(foo* dst, foo* src)`. Local correctness of this function may simply require that the objects pointed to by `src` and `dst`are allocated; global correctness of the program may require that `src` has been correctly initialized, and that the current ref count of `dst` be zero. The goal of the methodology is to determine the set of consistent *global* validity constraints that represent the *intent* of the development team, which often differs from the weakest precondition necessary to make a particular function *correct* in the mathematical sense. Overall, integrating specifications in the code increases our confidence about their validity and whether they are realistic, in the following ways:

1. specifications are checked in all test runs;
2. it is easier for the developers to get involved in the proof-review process, as approving a specification means that they add an assertion in their code;
3. specifications stay in sync with the code more easily, as they are co-located with the function implementation, and not in some detached proof harness; and
4. specifications in the code can also act as documentation for library users, as they explicitly specify how a client should call the external functions of the library.

5 | INTEGRATING WITH DEVELOPERS' WORKFLOW

The previous sections described how we write our proofs in a style that developers find familiar, making it more likely that the developers will scrutinize and engage with our proofs. Emulating developers' working style does not stop with the proofs themselves, however. Our entire proof development process and infrastructure also closely mimics the processes that are more familiar to developers. Specifically, we believe that four aspects of our development process contribute to our success with developers:

- we merge the proofs into the target code base, such that they become part of the source distribution;
- as part of this merge, we ask the developers to review the proofs like any other code contribution;
- once merged, the proofs are continuously checked, and the results are presented beside other test results;
- our continuous checking system has remarkably low latency and is highly reliable.

We elaborate on these points in the following subsections.

5.1 | Proofs in the source distribution

Our proofs do not live in a separate repository. Rather, we add the proofs to the code base that they apply to, giving the developers a sense of ownership. Developers (including external contributors) who move or rename files or change public APIs are thus responsible for also ensuring that proofs that link to those files continue to apply. This encourages developers to think of the proofs as part of the source code, rather than a separate artifact that may safely be ignored.

[9]The fix was merged via https://github.com/awslabs/aws-c-io/pull/132.

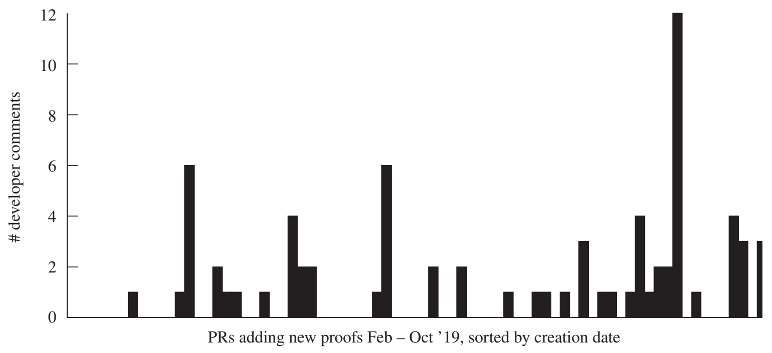


FIGURE 5 Number of developer comments on each pull request

Contributing the proofs to the repository also means that *users* of the software receive the proofs, and we provide instructions for users to run the proofs on their own machines. In some cases, our proofs are explicitly presented as a reliability assurance mechanism that gives users greater trust in the software.[49] Having the proofs as publicly-consumable artifacts provides another motivation for the developers to understand and engage with the proofs.

5.2 | Proof review

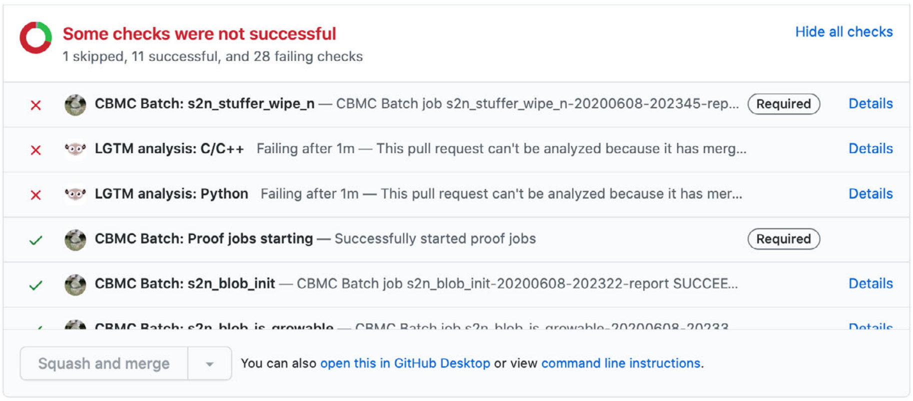
Our proofs are part of the software that AWS offers to users, rather than an internal effort that users cannot audit. This makes it imperative that the proofs we add to the repository are of exemplary quality and have the developers’ stamp of approval. We ensure this through the same mechanism as any other code contribution: by asking developers for a public code review. The fact that our proofs are written in the developers’ working language decreases their cognitive burden during review, and they review our proofs using the same web interface as for other contributions. Figure 5 shows the number of review comments that developers have made on each of our pull requests.

During review, developers feel free to (i) question the assumptions that we made in our proofs, (ii) suggest additional properties that we could check, and (iii) address our questions about the code. Item (iii) is particularly important when our proofs find “benign” bugs—that is, cases where developers intentionally use some potentially-unsafe language feature like arithmetic overflow. In such cases, we ask the developers to confirm that the feature was intentional before suppressing the error; 13 benign integer overflows have been annotated this way. This ensures that we find real bugs when they do exist, involving developers in the bug-finding process; and gives developers an opportunity to consider whether their use of the language feature really is safe under all circumstances. Items (i) and (ii) help developers to “become” members of the proof team, allowing them to contribute their thoughts on what we should be proving. The fact that the developers themselves are in the best position to offer these suggestions means that proof review is a mutually-insightful process, with the proof team and development team learning about each other’s work.

6 | CONTINUOUS FORMAL VERIFICATION

Rather than “proving code correct” and moving on, a core aspect of our activity is ensuring that code *remains* correct as the code changes, and therefore we have created a continuous integration system for proofs. Since this process is automated, it can be triggered every time a developer proposes a code change by posting a “pull request” on GitHub. This mechanism allows developers to ensure that their changes will not cause a previously-proved property to become invalid.

In keeping with the theme of using familiar tools and processes, the results of our CI are displayed beside other test results on the repository’s web interface, shown in Figure 6. Each property that we wrote a proof for has its result displayed using a tick or cross to indicate whether the property continues to hold after the code change, providing developers a peripheral awareness of our proof activity even when all the proofs go through. Conversely, when a proof fails to hold, developers are empowered to find and fix the problem by browsing to the proof report using the “Details” hyperlink. This report gives developers a concrete trace that led to the violated property, as well as an annotated source code listing that shows what part of the code our proof covered. Developers can then fix their code changes themselves or ask for our assistance. By presenting a concrete failed trace, developers can think of the error as a “failed test” rather than the more vague notion of a property failing to hold on some execution.


Some checks were not successful
1 skipped, 11 successful, and 28 failing checks
Hide all checks

×	CBMC Batch: s2n_stuffer_wipe_n — CBMC Batch job s2n_stuffer_wipe_n-20200608-202345-rep...	Required	Details
×	LGTM analysis: C/C++ Failing after 1m — This pull request can't be analyzed because it has merg...		Details
×	LGTM analysis: Python Failing after 1m — This pull request can't be analyzed because it has mer...		Details
✓	CBMC Batch: Proof jobs starting — Successfully started proof jobs	Required	
✓	CBMC Batch: s2n_blob_init — CBMC Batch job s2n_blob_init-20200608-202322-report SUCCEE...		Details
✓	CBMC Batch: s2n_blob_is_growable — CBMC Batch job s2n_blob_is_growable-20200608-20233		Details

Squash and merge
You can also open this in GitHub Desktop or view command line instructions.

FIGURE 6 Continuous integration reporting proof harnesses results on GitHub [Colour figure can be viewed at wileyonlinelibrary.com]

6.1 CI requirements

Valuable as continuous formal verification may be, it becomes much less *useful* to developers if it slows down or otherwise burdens their development process. Our experience is similar to O'Hearn, who also describes the continuous application of a static analysis tool in an industrial context.[25] O'Hearn mentions the importance of low proof-result latency (i.e., the time between the developer publishing a change request, and getting feedback from the analysis tool). This latency not only depends on the speed of the analysis tool and the size of the proof but it is also increased by the overhead that the CI system introduces.

Our experience integrating proofs into the developer workflow led to several requirements for an effective CI system:

- CI has to finish quickly, within the normal latency for reviewing a pull request (approximately 20 minutes);
- the software developer should have actionable information when a failure is detected;
- CI must be seamlessly integrate into the existing software development workflow.

At a high level, these requirements are the same as what is needed for any CI system for unit tests or integration tests. However, checking proofs differ from both unit and integration tests in ways that affect the design of the CI system. In the open-source projects we work on, unit tests tend to be numerous (e.g., hundreds of tests), but each unit test is inexpensive to run. The goal is that a developer ought to be able to run the unit-test suite as part of their build process during development. The entire unit-test suite, therefore, can run quickly on a single, low-powered machine. The CI system may launch several such machines to run tests across several operating systems (e.g., Windows/Linux/OSX) and compilers (Visual Studio[10], gcc[11], clang[12]), but this typically only requires a single-digit number of machines.

By contrast, integration tests may take significant computing resources to run, but there are not many of them. A small number of homogeneous machines can efficiently handle the integration test load.

Proofs, however, are both numerous (e.g., AWS C Common has over 170 proofs) and heterogeneous, that is, proof runtime ranges from seconds to tens of minutes, with memory usage ranging from megabytes to tens of gigabytes. A CI system must not wait until all of the proofs are finished before start reporting results back to the developer. Therefore, we need a CI system that supports a very high degree of parallelism so that it reports back on the fastest proofs while the long-running proofs continue running in the background. This combination of factors proved to be a poor fit for off-the-shelf CI systems such as Travis[13] or Jenkins.[14]For instance, Travis puts limits on machine size, job runtime, and parallelism. On earlier projects, it easily reached scalability limits, leading to numerous timeouts. Jenkins and AWS CodeBuild, at the time that our CI system was designed, did not have good support for parallelism. Furthermore, our customers tended to submit pull requests in bursts. There might be times a project receives several pull requests within minutes of each other or no pull requests at all for days. Thus, the workload is difficult to provision ahead of time.

[10]https://visualstudio.microsoft.com/
[11]https://gcc.gnu.org/
[12]https://clang.llvm.org/
[13]https://docs.travis-ci.com/
[14]https://jenkins.io/

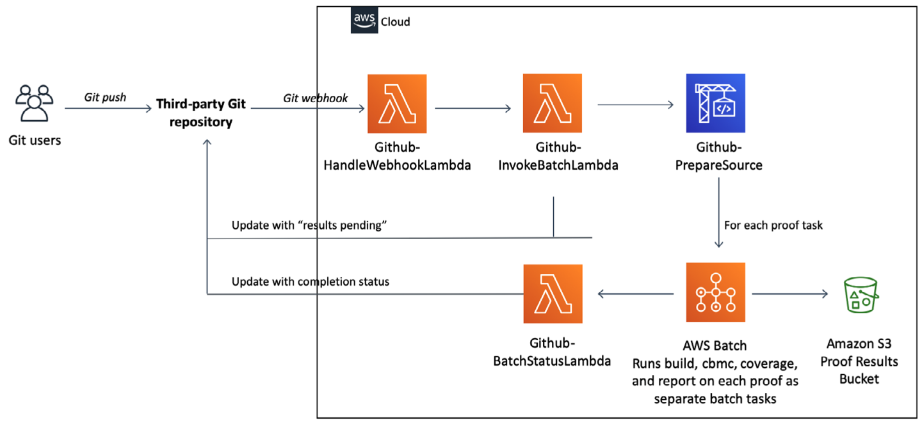

Cloud
Git users
Git push
Third-party Git repository
Git webhook
Github-HandleWebhookLambda
Github-InvokeBatchLambda
Github-PrepareSource
Update with "results pending"
For each proof task
Update with completion status
Github-BatchStatusLambda
AWS Batch
Runs build, cbmc, coverage, and report on each proof as separate batch tasks
Amazon S3 Proof Results Bucket

FIGURE 7 Continuous integration system architecture [Colour figure can be viewed at wileyonlinelibrary.com]

6.2 | System architecture

The considerations discussed in Section 6.1 led us to a highly parallel *serverless* CI architecture, shown in Figure 7 and explained in detail in Section 6.2.1. The parallelism of this architecture allows it to report back on proofs as they finish. We specifically built our architecture around serverless technology, which allows systems to be built in a declarative style, that is, the system designer specifies what computation should happen, and the underlying cloud infrastructure automatically executes the tasks. This approach also allows systems to easily and robustly scale while remaining cost-effective, as discussed in Section 6.2.2.

The CI system is built upon a handful of AWS services, notably AWS Lambda[15] and AWS Batch.[16]AWS Lambda is a service for running short, low-latency tasks without having to worry about the actual computing resources needed to run those tasks. We use this to do all of our event handling and communication back and forth with GitHub, the standard hosting platform for our open-source projects. We have AWS Lambdas for responding to webhooks, posting updates, and monitoring the status of longer-running jobs. For the proofs themselves, which may run for longer and need more powerful machines, our CI system uses AWS Batch. This service is suitable for long-running asynchronous jobs and allows us to specify the size of the machine we need for a particular task; thus, cost effective. We also used other AWS services like AWS API Gateway for triggering CI runs, AWS CodeBuild for creating batch jobs, AWS CloudTrail for collecting metrics, AWS CloudWatch for handling logs, and Amazon S3 for backend storage.

6.2.1 | System workflow

Our CI architecture is designed to be integrated with GitHub. Thus, whenever a developer creates a pull request or pushes a commit to a project repository, GitHub sends a REST message to the CI system (called a *webhook*) with the commit hash of the change and other details. This message passes through the AWS API Gateway service that can process webhooks and route them to other services. The webhook triggers an AWS Lambda function called *HandleWebhookLambda*, which packages a response to GitHub and invokes a second AWS Lambda function, called *InvokeBatchLambda*. The latter prepares a request for our batching system, which consists of two parts: an AWS CodeBuild task and a set of AWS Batch jobs. The AWS CodeBuild task receives data from a webhook about a pull request, including the commit hash (verification target). It then updates the pull request with "Pending" statuses for all proofs, to indicate that the proofs have been triggered correctly, and spins off AWS Batch jobs to run all proofs. There are time-consuming tasks in the proof-running process; therefore, our proof effort is split into four jobs:

1. build the proof-harness binaries (fast);
2. verify all properties using CBMC (slow);
3. check for code coverage (slow);
4. generate a detailed, user-friendly report on the proof results (fast).

[15]https://aws.amazon.com/lambda/
[16]https://aws.amazon.com/batch/

Steps 2 and 3 depend on step 1, but both could be run in parallel with each other, which significantly speeds up this process. Each of these AWS Batch jobs upload their results into a shared Amazon S3 bucket directory. Once these AWS Batch jobs have completed, the *BatchStatusLambda* function reports proof results back to the pull request, replacing the “Pending” statuses with “Pass” or “Fail.” Due to the parallelism of AWS Batch, we are able to update the status of each proof independently so the developer can see the results as they are generated.

This infrastructure is deployed using AWS CloudFormation, which is an “infrastructure as a service” solution. Essentially, the entire CI system is defined as a series of YAML templates that can be deployed on any AWS account. This gives us flexibility in terms of creating and managing AWS accounts.

6.2.2 | Benefits of serverless systems

Using serverless systems provided a number of benefits:

Continuous scaling. The serverless technologies we use automatically provision resources as needed. The number of active AWS Lambda functions scales automatically with the number of active requests. Similarly, the number of machines handling AWS Batch jobs scales automatically as the pool of queued jobs increases, providing effectively unlimited parallelism to our CI system.

Flexible performance. Both AWS Lambda and AWS Batch provide parameters which can be turned to match the size (e.g., CPU cores or available RAM memory) provisioned to the need.

Pay as you go pricing. Pay as you go pricing allows us to only only pay for the times we actually need to use our CI system. Open source development work tends to be “bursty.” Pay-as-you-go pricing means during lulls when no new code needs to be verified, no infrastructure is kept active, and the cost of operating the system is zero.

Infrastructure as code. Instead of having to actually manage physical or virtual servers, we can specify which AWS resources we need in YAML templates and maintain those as code in repositories. These AWS CloudFormation YAML files can be deployed to any AWS account, allowing us to spin up or tear down our CI system with a single command, and manage changes to the infrastructure using GitHub.

6.3 | CI system management

Initially, we planned a centralized CI system for all of our verification targets, but soon we realized this was inadequate to cope with the active development of proof tools. For instance, there are cases where different projects required different versions of the same tool. Therefore, we need an adaptive CI system to run different tooling on different verification targets, while also allowing us to bring these projects into sync if necessary.

First, we generate customized CI configurations for different verification targets with the ability to rollback or roll-forward with different tool versions. Second, we keep multiple versions of the proof tools in a common “tools account,” which allows us to easily synchronize multiple CI projects. In addition, we maintain a β version for each verification target where tool releases and CI configurations are tested, before launching them into production.

To deploy a CI system, we generate a “snapshot” of all of the CI configurations (AWS CloudFormation templates), as well as the specific versions of all the proof tools (e.g., CBMC). We then place that snapshot in the shared Amazon S3 bucket and assign it a unique ID. This allows us to deploy that snapshot into β, and then to easily promote that same snapshot to production once it is approved. If we upgrade a new snapshot into β and discover errors, we can simply redeploy the most recent working snapshot and quickly rollback to a stable state. All infrastructure to manage our CI systems is publicly available at AWS CBMC Batch project.[17]

Although AWS CloudFormation automates the deployment of a stack on a particular account, we also need to coordinate passing information and parameters across several different AWS accounts and regions. While there are many deployment tools available (e.g., Ansible[18] and AWS CodeDeploy[19]), none of them currently provide the set of features

[17]https://github.com/awslabs/aws-batch-cbmc
[18]https://www.ansible.com
[19]https://aws.amazon.com/codedeploy/

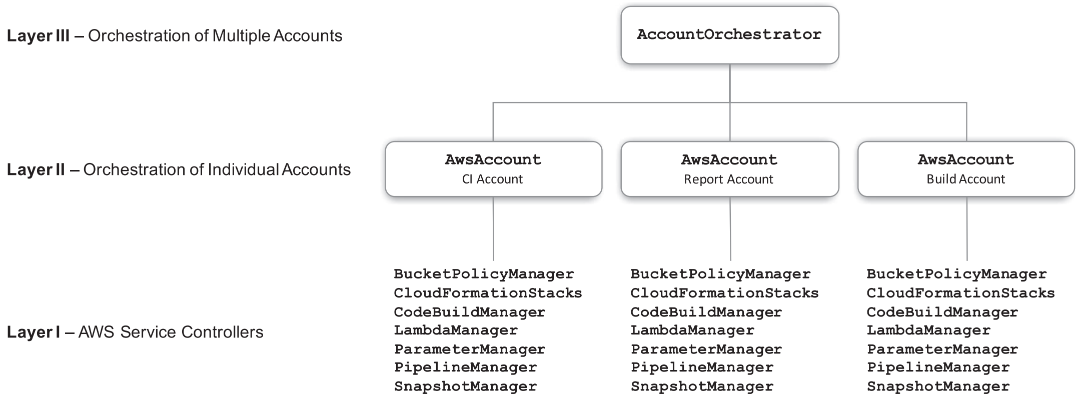


FIGURE 8 Architecture of CI system deployment modules

we needed to manage this coordination between accounts. We also need to track exactly which versions of AWS CloudFormation templates, proof tools and Docker container images we are running for specific verification targets. Therefore, we built our on set of modules for managing CI deployments, which has three layers of classes as shown in Figure 8.

6.3.1 | Layer I—AWS service controllers

The bottom layer is an abstraction over the basic AWS services. It simplifies the creation and management of AWS CloudFormation stacks, triggering and monitoring pipelines and managing Amazon S3 bucket policies, and setting environment variables in various services.

CloudFormationStacks. This class provides a thin layer above the `boto3`[20]client, which provides everything to do with deploying, waiting for, checking the status of and retrieving output values from stacks in AWS CloudFormation.

SnapshotManager. In order to ensure that we can maintain consistency across many different CI accounts, we need to have a concept of a snapshot. The purpose of this class is to maintain an image of all of the AWS CloudFormation templates (CI configurations), AWS Lambda functions, and a specific version of every proof tool we use in CI. Our system then packages them all together in an archive, give it an unique ID, and uploads it into a specific folder in Amazon S3. This allows us to compare two accounts and check whether they are running precisely the same tools, being triggered by the same AWS Lambda functions, and so on. Similarly, it allows us to easily promote a successful snapshot from β to production, or to rollback to a previous state if something goes wrong. This structure allows the higher levels of our architecture to deal with snapshots only as an ID, and abstracts away all of the details of how they work and where they are stored.

BucketPolicyManager. We store our snapshots in a shared Amazon S3 bucket, which will be accessed by every CI account. Thus, bucket policies must give read access to CI accounts in a secure way. Whenever a new CI account is deployed, our system securely grant it the right permission to access the shared bucket.

PipelineManager. When new code gets merged into the CI system repository or if a new change is pushed into CBMC, and we want to use this code in a CI account, this triggers pipelines that build and package the new versions of the code. This module provides functionality for triggering, waiting, and managing those pipelines. For instance, if we want to deploy the latest version of our CI or the latest CBMC version, we can have our scripts wait for all pipelines to complete so that we can know that we have deployed the latest code.

ParameterManager. The parameters that we use when filling in an AWS CloudFormation template can come from several different sources. Firstly, when we do a deployment, we take in a JSON file that has any project specific parameters. This file is indexed by account ID to ensure that we do not accidentally pass parameters from one account into another.

[20]https://aws.amazon.com/sdk-for-python/

These parameters contain information such as the name of the project, whether or not the project should post its results to GitHub, and which branch it should be checking. In the case of a build account, we have information about which repository we are getting the AWS CloudFormation templates from and which repository we get each of the tools we use from for our build pipelines. We also have other sources of data that can be use to fill AWS CloudFormation parameters. For instance, we could get our parameters from the outputs of stacks that already exist either in a shared build account or a CI account somewhere. We also fill parameters with data from other AWS services, such as which accounts we would like to allow in an Amazon S3 bucket policy and secrets from the AWS Secret Manager like `OAth` tokens for GitHub.

6.3.2 | Layer II—Orchestration of individual accounts

The second layer orchestrates all AWS services. This second layer is the *AwsAccount* module, where the idea is to have a single object that tracks all the necessary information and parameters for an account, and exposes methods to deploy and manage that account. This generic interface deploys several stacks concurrently, and then waits for them to be in a stable state. It also allows us to specify whether we want to wait for pipelines to complete once a particular stack has been updated or what input parameters each stack takes. We can even specify a *ParameterManager*that will determine how those parameters get assigned (since they often come from multiple sources).

The most important method from this module is deploy_stacks(self, stacks_to_deploy, s3_template_source=None, ove rrides=None), where stacks_to_deploy is the specifications for which stacks we want to deploy and which templates they need. These templates can either come from an Amazon S3 in the shared account, Amazon S3 in the proof account, or locally depending on the s3_template_source value. The default behaviour is to look for the templates locally in the current folder. The specification takes the form of a dictionary that looks like the following:

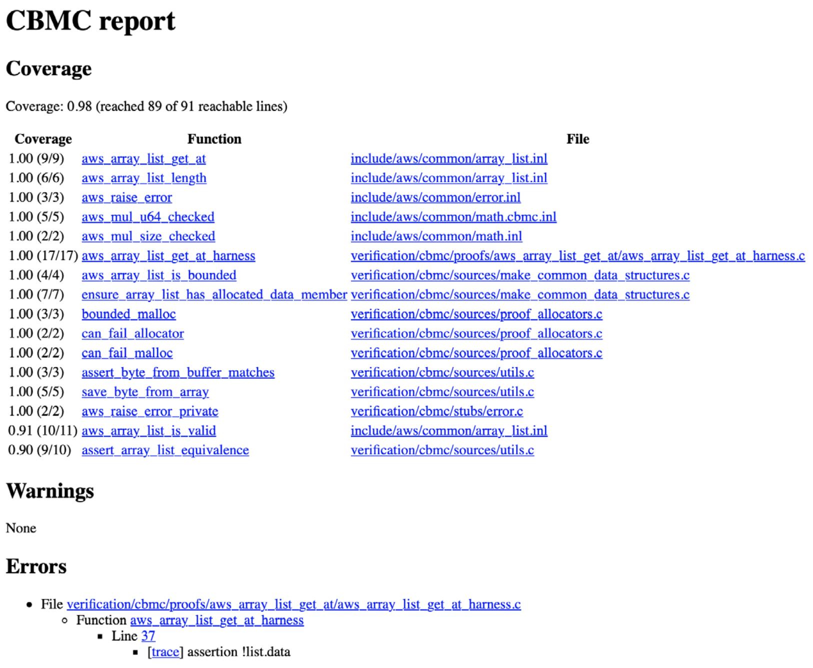
```
BUILD_TOOLS_CLOUDFORMATION_DATA = {
    "build-batch": {
        TEMPLATE_NAME_KEY: "build-batch.yaml",
        PARAMETER_KEYS_KEY: ['BatchRepositoryBranchName',
                             'BatchRepositoryName',
                             'BatchRepositoryOwner',
                             'GitHubToken',
                             'S3BucketName'],
        PIPELINES_KEY: ["Build-Batch-Pipeline"]
    },
    "build-viewer": {
        TEMPLATE_NAME_KEY: "build-viewer.yaml",
        PARAMETER_KEYS_KEY: ['GitHubToken',
                             'S3BucketName',
                             'ViewerRepositoryBranchName',
                             'ViewerRepositoryName',
                             'ViewerRepositoryOwner'],
        PIPELINES_KEY: ["Build-Viewer-Pipeline"]
    }
}
```

which specifies to concurrently start deploying “build-batch” and “build-viewer” templates, as well as it gives the parameters that each of these templates need. These parameters are simply the input parameters of the AWS CloudFormation template. The parameters will be supplied using the *ParametersManager*. It also specifies that once we deploy these templates and the stacks are stable, we would like to wait for the “Build-Batch-Pipeline” and the “Build-Viewer-Pipeline” to complete.

6.3.3 | Layer III—Orchestration of multiple accounts

The top layer is the *AccountOrchestrator* module, which manages the interactions between the shared build account and the CI account, and optionally, a third account that contains the website. The purpose of the *AccountOrchestrator* is to ensure that information that has to be shared between many accounts can be passed to an individual CI account, but that information specific to a particular project’s CI account is never leaked to the shared account.

When deploying proof accounts, we need access to the shared tool account to do things like download templates and update the bucket policy. This is particularly complicated handling cross region resources. The *AccountOrchestrator*



CBMC report

Coverage

Coverage: 0.98 (reached 89 of 91 reachable lines)

Coverage	Function	File
1.00 (9/9)	aws_array_list_get_at	include/aws/common/array_list.inl
1.00 (6/6)	aws_array_list_length	include/aws/common/array_list.inl
1.00 (3/3)	aws_raise_error	include/aws/common/error.inl
1.00 (5/5)	aws_mul_u64_checked	include/aws/common/math.cbmc.inl
1.00 (2/2)	aws_mul_size_checked	include/aws/common/math.inl
1.00 (17/17)	aws_array_list_get_at_harness	verification/cbmc/proofs/aws_array_list_get_at/aws_array_list_get_at_harness.c
1.00 (4/4)	aws_array_list_is_bounded	verification/cbmc/sources/make_common_data_structures.c
1.00 (7/7)	ensure_array_list_has_allocated_data_member	verification/cbmc/sources/make_common_data_structures.c
1.00 (3/3)	bounded_malloc	verification/cbmc/sources/proof_allocators.c
1.00 (2/2)	can_fail_allocator	verification/cbmc/sources/proof_allocators.c
1.00 (2/2)	can_fail_malloc	verification/cbmc/sources/proof_allocators.c
1.00 (3/3)	assert_byte_from_buffer_matches	verification/cbmc/sources/utils.c
1.00 (5/5)	save_byte_from_array	verification/cbmc/sources/utils.c
1.00 (2/2)	aws_raise_error_private	verification/cbmc/stubs/error.c
0.91 (10/11)	aws_array_list_is_valid	include/aws/common/array_list.inl
0.90 (9/10)	assert_array_list_equivalence	verification/cbmc/sources/utils.c

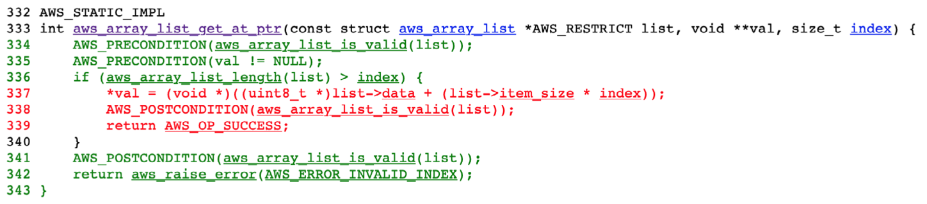
Warnings

None

Errors

- File verification/cbmc/proofs/aws_array_list_get_at/aws_array_list_get_at_harness.c
  - Function aws_array_list_get_at_harness
    - Line 37
      - [trace] assertion !list.data

FIGURE 9 Results report from a CBMC-proof harness [Colour figure can be viewed at wileyonlinelibrary.com]



```
AWS_STATIC_IMPL
int aws_array_list_get_at_ptr(const struct aws_array_list *AWS_RESTRICT list, void **val, size_t index) {
    AWS_PRECONDITION(aws_array_list_is_valid(list));
    AWS_PRECONDITION(val != NULL);
    if (aws_array_list_length(list) > index) {
        *val = (void *)((uint8_t *)list->data + (list->item_size * index));
        AWS_POSTCONDITION(aws_array_list_is_valid(list));
        return AWS_OP_SUCCESS;
    }
    AWS_POSTCONDITION(aws_array_list_is_valid(list));
    return aws_raise_error(AWS_ERROR_INVALID_INDEX);
}
```

FIGURE 10 Code snippet from a proof-results report highlighting the covered code (in green) and the uncovered code (in red) [Colour figure can be viewed at wileyonlinelibrary.com]

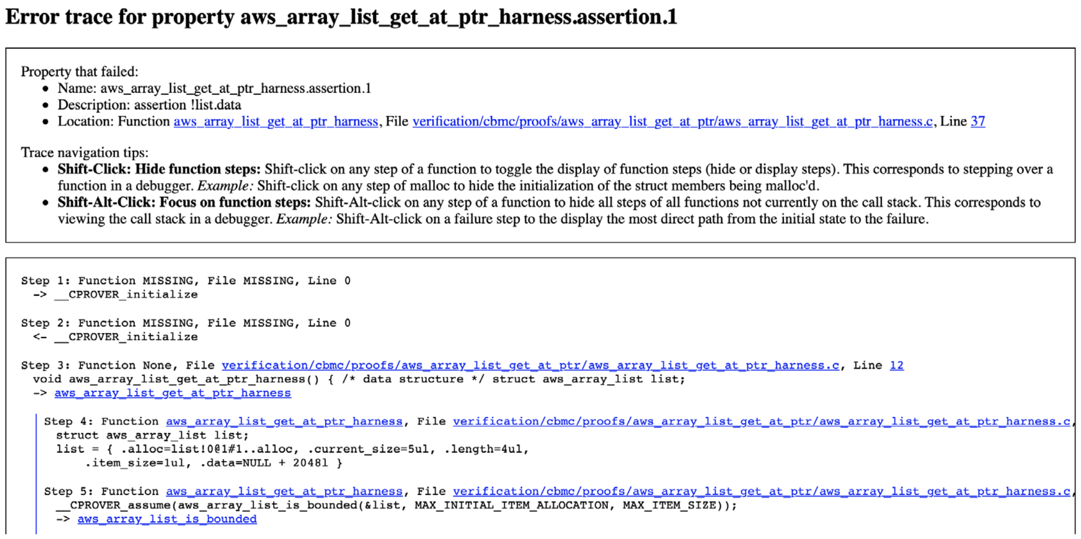
keeps an *AwsAccount* object for any profile that we deal with. This handles all of the logic of passing the appropriate stacks_to_deploy specifications, as well as any data that must be passed from one account to another. This minimizes the chances that we will accidentally pass project specific data to a shared account or some other careless error. It makes our decisions explicit and visible, while also giving flexibility in case we realize we need to transfer more or less data in the future.

6.4 | Reporting proof results

We want to empower software engineers to write proofs and have these guarantees continuously run on their code; thus, we need to give them tools to understand what is happening when those proofs fail and quickly efficiently fix the code. For every failed proof, CBMC will generate counterexample traces that document precisely how the property violation occurred. These counterexamples are not necessarily easy for a general software engineer to understand and debug. Therefore, we built a tool called CBMC Viewer that spits out a much more intuitive HTML representation of CBMC-proof results (see Figure 9). Effectively, it provides to the user the benefit of an interactive debugger run in the browser, using information that we already collect during CI.

CBMC Viewer gives us a straightforward and easy to understand representation of what percentage of the program is covered and uncovered by the proof harness (see Figure 10). The report shows that a portion of the code was not reachable from the proof entry point, and if we click on the hyperlink, we can see the exact lines of code CBMC could not reach. CBMC Viewer colors unreachable code in red and reachable code in green, which makes it clear to developers (see Figure 10). It also gives us the option of adhere software engineering best practices by integrating with existing code coverage tools (e.g., Codecov[21]).

[21]https://docs.codecov.io/docs/about-code-coverage



Error trace for property aws_array_list_get_at_ptr_harness.assertion.1

Property that failed:
- Name: aws_array_list_get_at_ptr_harness.assertion.1
- Description: assertion !list.data
- Location: Function aws_array_list_get_at_ptr_harness, File verification/cbmc/proofs/aws_array_list_get_at_ptr/aws_array_list_get_at_ptr_harness.c, Line 37

Trace navigation tips:
- **Shift-Click: Hide function steps:** Shift-click on any step of a function to toggle the display of function steps (hide or display steps). This corresponds to stepping over a function in a debugger. *Example:* Shift-click on any step of malloc to hide the initialization of the struct members being malloc'd.
- **Shift-Alt-Click: Focus on function steps:** Shift-Alt-click on any step of a function to hide all steps of all functions not currently on the call stack. This corresponds to viewing the call stack in a debugger. *Example:* Shift-Alt-click on a failure step to the display the most direct path from the initial state to the failure.

```
Step 1: Function MISSING, File MISSING, Line 0
  -> __CPROVER_initialize

Step 2: Function MISSING, File MISSING, Line 0
  <- __CPROVER_initialize

Step 3: Function None, File verification/cbmc/proofs/aws_array_list_get_at_ptr/aws_array_list_get_at_ptr_harness.c, Line 12
  void aws_array_list_get_at_ptr_harness() { /* data structure */ struct aws_array_list list;
  -> aws_array_list_get_at_ptr_harness

    Step 4: Function aws_array_list_get_at_ptr_harness, File verification/cbmc/proofs/aws_array_list_get_at_ptr/aws_array_list_get_at_ptr_harness.c,
      struct aws_array_list list;
      list = { .alloc=list!0@1#1..alloc, .current_size=5ul, .length=4ul,
          .item_size=1ul, .data=NULL + 20481 }

    Step 5: Function aws_array_list_get_at_ptr_harness, File verification/cbmc/proofs/aws_array_list_get_at_ptr/aws_array_list_get_at_ptr_harness.c,
      __CPROVER_assume(aws_array_list_is_bounded(&list, MAX_INITIAL_ITEM_ALLOCATION, MAX_ITEM_SIZE));
      -> aws_array_list_is_bounded
```

FIGURE 11 Counterexample trace with links to code locations for easy debugging [Colour figure can be viewed at wileyonlinelibrary.com]

CBMC Viewer also gives us a way to step through a counterexample trace and debug it if a proof fails (as shown in Figure 11). The developer can click on any line of the trace to see the code that the line refers to, which gives a user experience reminiscent of an interactive debugger. This feature is a huge help when debugging proofs, and vastly improve the developer's ability to diagnose problems quickly.

One of the challenges with providing a web-based report for proof results is that there is a tension between maintaining security and giving developers easy access to reports. Our CI system runs in a virtual private network and maintains a high-security level, but once it finishes, we need a way to publish the proof reports back to developers. Our proofs are currently in open-source projects; thus, we need to make these reports accessible to the public. Our CI system generates proof reports and stores them in Amazon S3 buckets as HTML files. It would be dangerous to access this bucket because someone might unintentionally upload sensitive data, not realizing it is publicly reachable.

In order to securely provide developers access to the proof reports generated by our CI system, we use AWS CloudFront[22] and AWS Lambda@Edge.[23]These are AWS solutions to efficiently and securely serve content from S3 buckets. AWS CloudFront sets up a server that gives an URL to expose files in particular directories of an Amazon S3 bucket. It does that without giving information about the name of the bucket or directories without explicit permission. It also handles caching the HTML pages on edge nodes, close to the user, giving fast response times. AWS Lambda@Edge is the variant of AWS Lambda that runs on edge nodes in the AWS infrastructure in response to AWS CloudFront requests. This tool allows us to maintain security by verifying that any page we display to the user must have metadata showing that it comes from our CI system and is intentionally released publicly. AWS Lambda@Edge gives us confidence that even if somehow sensitive data were to get into the Amazon S3 bucket, we would not leak that data to the public.

6.5 | Reliability and responsiveness of proofs in CI

Our CI system spawns a pool of proof jobs that start simultaneously and report their results back to the pull request as soon as they finished; therefore, the time taken to run all proof jobs is very similar to the time taken to run the longest one. On average, a median proof harness takes 88 seconds to complete, and CI startup latency is 359 seconds. In some of our verification targets, our entire proof suite completes even before the developers' tests.

Our CI system is event-driven—reacting to developers posting pull requests on GitHub—thus, the system's demands fluctuate over the week, as Figure 12 depicts. Developers rarely post pull-requests over the weekend, but the system experiences a high load during working hours. High uptime improves the developers' experience with the system. We

[22]https://aws.amazon.com/cloudfront/
[23]https://aws.amazon.com/lambda/edge/

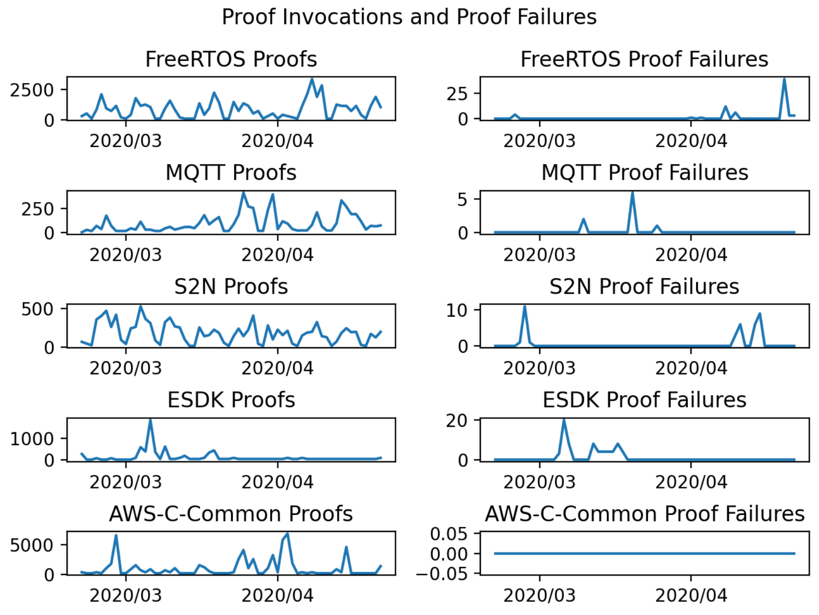


FIGURE 12 Proof invocations and proof failures per day over a 60-days period across multiple verification targets [Colour figure can be viewed at wileyonlinelibrary.com]

can also see those proof failures happen in bursts. Interestingly, proof failures do not necessarily correlate with proof invocations. It is also worth noting that during 60 days, there were no proof failures at all in AWS C Common, that is, the changes developers made do not break any the proofs. There are usually spikes in proof failures when there is an ongoing development in the proofs themselves.

Thus, we used a serverless architecture that can seamlessly respond to periods of high demand. We also maintain a system in a separate β account that we promote to production once we are confident of its reliability.

6.6 | Real-world challenges

As we extended our CI system to manage proof builds from multiple teams in multiple repositories, we found several challenges related to scaling and idiosyncrasies of different build environments and CI configurations. Indeed, these idiosyncrasies were a significant part of our operational burden. Most of these issues do not directly impact developers but rather increase the team's operational burden maintaining the CI system, which could slow down the team. Ultimately, this issue could also make it more difficult to onboard new projects interested in using proof tools in their CI. Another problem is that when there are many CI failures, even if they do not impact developers, there is a risk that these extra alarms will make it more difficult to notice more severe problems.

Force pushes to GitHub. Pull requests on a given project are not evenly distributed over time. When developers are actively working on an issue, they will sometimes make many different revisions to a pull request or many pushes to the repository, all in a short time-frame. This behavior presents a challenge to our system. By the time we process an event and are ready to pull and verify the commit, the commit might have been removed by a git "force push" from the developer, which replaces the commit with another. Force pushes occur regularly and are among the most common causes of errors in our CI system, becoming a significant source of unnecessary alarms. During peak development seasons, we generate so many failures and alarms that it can be difficult for the on-call person to understand what is going on and distinguish between failures that need an investigation from the ones that are simply the result of normal "force-pushes." The solution was to make an initial attempt to pull the commit, and on failure, check to see if the commit still exists by using git cat−file −e COMMIT−SHA. If it exists, we know that we have a legitimate error and alarm on it; otherwise, we can safely assume it is no longer relevant to check.

GitHub update limits. We started to run into problems related to peak development seasons because we started to exceed GitHub's rate-limit for status updates. We were posting every proof as a separate update to GitHub, and as the number of proofs grew and there were many commits in a short time, we would exceed the 5000 updates per hour that GitHub allows. This issue had a severe impact on developers since once we reached this GitHub API limit, they will stop seeing any additional updates to their CI statuses from proofs. The result of this is that a proof would be marked as "pending" indefinitely, which severely diminishes developers' trust in our CI system. We have solved this problem by adding a queueing layer in between the CI system and GitHub. We pull GitHub update requests of the queue and push them only if we have sufficient remaining API limits, which also has the advantage of allowing retries on failure.

Verification over multiple branches. One of the assumptions we made when we first designed our CI system was that every team would want us to run proofs on any pull request or push to any repository branch. In reality, some teams wanted us to run CI on all branches, while others wanted us only to run on a single branch or a subset of branches. At first, this would lead to false alarms where CI would fail because it was running on a branch that the teams did not expect us to be checking. This issue leads to two problems. From the developers' perspective, they will see failed proofs on pull requests into branches where they are not interested in running the proofs. For the CI maintainers, this means we are using resources checking branches unnecessarily and will have to respond to alarms related to proof failures that are not meaningful.

Diversity in git and makefile structures. Different teams also have a diversity of approaches for how to include and organize dependencies. We initially assumed that all of the code that we needed to run the proofs would be in the same repository. In general, this is not true. Many of the repositories we work with have submodules that need to be cloned recursively. For instance, the FreeRTOS Kernel repository has a structure where they store the kernel code itself in one repository and tests in another repository with the kernel as a submodule. If we wanted to verify the FreeRTOS Kernel, it would be difficult to integrate with our CI approach since we need to clone the more general test repository with the latest commit and then clone the kernel submodule with the commit that we are verifying. We are not currently running CI for this project. The AWS Encryption SDK does not use submodules but instead uses CMake[24]to do dependency installation. CMake allows a feature to take prebuilt dynamic libraries and links to them, which is currently impossible when running CBMC. CBMC needs to have access to the source code itself to build and link to it. Our CI system has to clone the AWS C Common library to properly run proofs in CI for the AWS Encryption SDK proofs. The direct real-world impact of these issues is that it complicates and delays the delivery of CI systems to new projects because we need to adapt our CI to the particular project structure.

Refactorings. A final issue we ran into involved substantial refactorings of a codebase, modifying the repository's directory structure. Our proofs rely heavily on Makefiles that are dependent on the particular directory structure of the project. Thus, when developers refactor the codebase under verification, it leads to proof failures, which are not for lack of correctness, but for dependency issues on source code locations.

7 | CONCLUSIONS AND FUTURE WORK

We have described a proof development style that embeds the proof creation and maintenance process in the software development cycle to deeply engage software developers and, ultimately, help make formal verification a routine activity. We have found that each time we have made choices that provided value to the development team, they also improved the ability of the verification team to effectively perform our verification tasks. We highlight four takeaways from this experience:

1. make specifications explicit in the code;
2. write unit-test like proofs in declarative style;
3. integrate proof artifacts into the developer work-flow; and
4. fix bugs instead of just reporting them.

These takeaways emphasize that the human factors and social processes of software development are as important as the technical aspects of formal verification. Indeed, this has been a reciprocal activity where we have increased our proof activity and we have seen developers take part in writing specifications for their code as they became accustomed to the process.

[24]https://cmake.org

As future work, we want to prove deeper properties for even broader software code bases, while maintaining developer engagement in the proof process. As a call-to-action for the community, a critical future challenge that we see is the long-term maintenance cost of proofs to ensure lasting software quality. How can proof artifacts be kept in-sync with the code base after the verification team has moved onto other projects? And what can we do to ensure that future developers fix proofs as a matter-of-course, just as they would fix a unit test? We hope to explore these questions in the future as we scale proofs to become a standard part of the developer's toolbox for foundational code.

ORCID

Nathan Chong https://orcid.org/0000-0001-7843-9556
Felipe R. Monteiro https://orcid.org/0000-0001-9420-9056
Daniel Schwartz-Narbonne https://orcid.org/0000-0002-0453-2552

REFERENCES

1. Moy Y, Ledinot E, Delseny H, Wiels V, Monate B. Testing or formal verification: DO-178C alternatives and industrial experience. *IEEE Softw*. 2013;30(3):50-57.
2. Cook B. Automated formal reasoning about AWS systems. *Proceedings of the 2017 Formal Methods in Computer Aided Design (FMCAD)*. Vol 7. Vienna, Austria: IEEE; 2017 https://doi.org/10.23919/FMCAD.2017.8102231.
3. Cook B. Automated formal reasoning about Amazon web services (Keynote). Paper presented at: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software SPIN 2017; 2017:9; Association for Computing Machinery, New York, NY.
4. Backes J, Bolignano P, Cook B, et al. Semantic-based automated reasoning for AWS access policies using SMT. Paper presented at: Proceedings of the 2018 Formal Methods in Computer Aided Design; 2018:1-9. https://doi.org/10.23919/FMCAD.2018.8602994.
5. Backes J, Bayless S, Cook B, et al. Reachability analysis for AWS-based networks. In: Dillig I, Tasiran S, eds. *Computer Aided Verification*. Cham, Switzerland: Springer International Publishing; 2019:231-241 https://doi.org/10.1007/978-3-030-25543-5_14.
6. Chong N, Cook B, Eidelman J, et al. Code-level model checking in the software development workflow. Paper presented at: Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP); 2020; ACM, New York, NY. https://doi.org/10.1145/3377813.3381347.
7. Chong N, Cook B, Eidelman J, et al. Code-level model checking in the software development workflow – replication package; 2020. https://doi.org/10.5281/zenodo.4090155.
8. Calcagno C, Distefano D. *Infer: An Automatic Program Verifier for Memory Safety of C Programs. Lecture Notes in Computer Science*. Vol 6617. Cham, Switzerland: Springer International Publishing; 2011:459-465 https://doi.org/10.1007/978-3-642-20398-5_33.
9. Bessey A, Block K, Chelf B, et al. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun ACM*. 2010;53(2):66-75. https://doi.org/10.1145/1646353.1646374.
10. Aagaard MD, Jones RB, Melham TF, O'Leary JW, Seger C-JH. A methodology for large-scale hardware verification. In: Hunt Warren A, Johnson Steven D, eds. *Formal Methods in Computer-Aided Design*. Berlin/Heidelberg, Germany: Springer; 2000:300-319.
11. Chudnov A, Collins N, Cook B, et al. *Continuous Formal Verification of Amazon s2n, Lecture Notes in Computer Science*. Vol 10982. Cham, Switzerland: Springer International Publishing; 2018:430-446.
12. Clarke EM, Henzinger TA, Veith H. *Handbook of Model Checkingch. Introduction to Model Checking*. Cham, Switzerland: Springer International Publishing; 2018:1-26 https://doi.org/10.1007/978-3-319-10575-8.
13. D'silva V, Kroening D, Weissenbacher G. A survey of automated techniques for formal software verification. *IEEE Trans CAD Integrat Circ Syst*. 2008;27(7):1165-1178. https://doi.org/10.1109/TCAD.2008.923410.
14. Beyer D. Automatic verification of C and java programs: SV-COMP 2019. *LNCS*. Vol 11429. Cham, Switzerland: Springer International Publishing; 2019:133-155 https://doi.org/10.1007/978-3-030-17502-3_9.
15. Post H, Küchlin W. *Integrated Static Analysis for Linux Device Driver Verification Lecture Notes in Computer Science*. Vol 4591. New York, NY: Springer; 2007:518-537 https://doi.org/10.1007/978-3-540-73210-5_27.
16. Cordeiro L, Fischer B, Marques-Silva J. Continuous verification of large embedded software using smt-based bounded model checking. Paper presented at: Proceedings of the 2010 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems; 2010:160-169. https://doi.org/10.1109/ECBS.2010.24.
17. Calcagno C, Distefano D, Dubreil J, et al. *Moving Fast with Software Verification. Lecture Notes in Computer Science*. Vol 9058. Cham, Switzerland: Springer International Publishing; 2015:3-11.
18. Cook B. Formal reasoning about the security of amazon web services. In: Chockler H, Weissenbacher G, eds. *Computer Aided Verification*. Cham, Switzerland: Springer International Publishing; 2018:38-47.
19. Distefano D, Fähndrich M, Logozzo F, O'Hearn PW. Scaling static analyses at Facebook. *Commun ACM*. 2019;62(8):62-70. https://doi.org/10.1145/3338112.
20. Sadowski C, Aftandilian E, Eagle A, Miller-Cushon L, Jaspan C. Lessons from building static analysis tools at Google. *Commun ACM*. 2018;61(4):58-66. https://doi.org/10.1145/3188720.
21. Christakis M, Bird C. What developers want and need from program analysis: an empirical study. Paper presented at: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering; 2016:332-343; New York, NY, ACM. http://doi.acm.org/10.1145/2970276.2970347.

22. Harman M, O'Hearn PW.. From start-ups to scale-ups: opportunities and open problems for static and dynamic program analysis. Paper presented at: Proceedings of the 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM); 2018:1-23; Madrid, Spain. https://doi.org/10.1109/SCAM.2018.00009.
23. Krishnamurthi S, Nelson T. The human in formal methods. In: Beek MH, McIver A, Oliveira JN, eds. *Formal Methods – The Next 30Years*. Lecture Notes in Computer Science 11800. Cham, Switzerland: Springer, Springer International Publishing; 2019:3-10. https://doi.org/10.1007/978-3-030-30942-8_1.
24. Ball T, Cook B, Levin V, Rajamani SK. SLAM and static driver verifier: technology transfer of formal methods inside microsoft. In: Boiten EA, Derrick J, Smith G, eds. *Integrated Formal Methods*. Berlin/Heidelberg, Germany: Springer; 2004:1-20.
25. O'Hearn PW. Continuous reasoning: scaling the impact of formal methods. *LICS '18*. New York, NY: ACM; 2018:13-25 http://doi.acm.org/10.1145/3209108.3209109.
26. Tillmann N, Schulte W. Parameterized unit tests. *ESEC/FSE-13*. New York, NY: Association for Computing Machinery; 2005:253-262 https://doi.org/10.1145/1081706.1081749.
27. Robertson C. *Using SAL Annotations to Reduce C/C++ Code Defects*. Microsoft; 2016. https://web.archive.org/web/20200828151221/https://docs.microsoft.com/e%n-us/cpp/code-quality/using-sal-annotations-to-reduce-c-cpp-code-defects?view=%vs-2019.
28. Flanagan C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., & Stata, R. Extended static checking for java. Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation; 2002:234-245; ACM, New York, NY. https://doi.org/10.1145/512529.512558.
29. Cook B, Khazem K, Kroening D, Tasiran S, Tautschnig M, Tuttle MR. Model checking boot code from AWS data centers. In: Chockler H, Weissenbacher G, eds. *Computer Aided Verification*. Cham, Switzerland: Springer International Publishing; 2018:467-486 https://doi.org/10.1007/978-3-319-96142-2_28.
30. Athanasiou K, Cook B, Emmi M, MacCarthaigh C, Schwartz-Narbonne D, Tasiran S. SideTrail: verifying time-balancing of cryptosystems. In: Piskac R, Rümmer P, eds. *Verified Software. Theories, Tools, and Experiments*. Cham, Switzerland: Springer International Publishing; 2018:215-228 https://doi.org/10.1007/978-3-030-03592-1_12.
31. Chong N. *Ensuring the Memory Safety of FreeRTOS Part 1*. Amazon Web Services, Inc; 2020. https://www.freertos.org/2020/02/ensuring-the-memory-safety-of-freertos%-part-1.html.
32. Chong N. *Ensuring the Memory Safety of FreeRTOS Part 2*. Amazon Web Services, Inc; 2020. https://www.freertos.org/2020/05/ensuring-the-memory-safety-of-freertos%-part-2.html.
33. Clarke EM, Kroening D, Lerda F. *A Tool for Checking ANSI-C Programs. Lecture Notes in Computer Science*. Vol 2988. Cham, Switzerland: Springer International Publishing; 2004:168-176 https://doi.org/10.1007/978-3-540-24730-2_15.
34. Eén N, Sörensson N. *An extensible SAT-solver. Lecture Notes in Computer Science*. Vol 2919. New York, NY: Springer; 2003:502-518 https://doi.org/10.1007/978-3-540-24605-3_37.
35. Cuoq P, Kirchner F, Kosmatov N, Prevosto V, Signoles J, Yakobowski B. Frama-C: a software analysis perspective. *SEFM'12*. Berlin/Heidelberg, Germany: Springer-Verlag; 2012:233-247 doi.org/10.1007/978-3-642-33826-7_16.
36. Vogels F, Jacobs B, Piessens F. Featherweight VeriFast. *Log Methods Comput Sci*. 2015;11(3:19):1-57. https://doi.org/10.2168/LMCS-11(3:19)2015.
37. Klein G, Elphinstone K, Heiser G, et al. SeL4: formal verification of an OS kernel. *SOSP '09*. New York, NY: Association for Computing Machinery; 2009:207-220 https://doi.org/10.1145/1629575.1629596.
38. Hawblitzel C, Howell J, Kapritsos M, et al. IronFleet: proving practical distributed systems correct. Paper presented at: Proceedings of the 25th Symposium on Operating Systems Principles; 2015; New York, NY, ACM - Association for Computing Machinery. https://doi.org/10.1145/2815400.2815428.
39. Gurfinkel A, Kahsai T, Komuravelli A, Navas JA. The SeaHorn verification framework. In: Kroening D, Păsăreanu CS, eds. *Computer Aided Verification*. Cham, Switzerland: Springer International Publishing; 2015:343-361 https://doi.org/10.1007/978-3-319-21690-4_20.
40. Gadelha MR, Monteiro FR, Morse J, Cordeiro LC, Fischer B, Nicole DA. ESBMC 5.0: an industrial-strength C model checker. Paper presented at: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering; 2018:888-891; ACM; New York, NY. https://doi.org/10.1145/3238147.3240481.
41. Rakamarić Z, Emmi M. SMACK: decoupling source language details from verifier implementations. In: Biere A, Bloem R, eds. *Computer Aided Verification*. Cham, Swizerland: Springer International Publishing; 2014:106-113. https://doi.org/10.1007/978-3-319-08867-9_7.
42. Enumeration Common Weakness. *CWE Top 25 Most Dangerous Software Errors*. The MITRE Corporation; 2019. https://web.archive.org/web/20200828151626/https://cwe.mitre.org/top25/%archive/2019/2019_cwe_top25.html.
43. Miller M. *Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape*. Microsoft Security Response Center; 2019. https://web.archive.org/web/20200828152254/https://github.com/microsoft%/MSRC-Security-Research/blob/master/presentations/2019_08_WOOT/WOOT19%20-%20Tr%ends%20and%20challenges%20in%20vulnerability%20mitigation.pdf.
44. Szekeres L, Payer M, Wei T, Song D. *SoK: Eternal War in Memory*. Washington, DC: IEEE Computer Society; 2013:48-62.
45. Veen V, Sharma N, Cavallaro L, Bos H. Memory errors: the past, the present, and the future. In: Balzarotti D, Stolfo SJ, Cova M, eds. *Research in Attacks, Intrusions, and Defenses*. Berlin/Heidelberg, Germany: Springer; 2012:86-106 https://doi.org/10.1007/978-3-642-33338-5_5.
46. Hathhorn C, Rosu G. Dealing with C's original sin. *IEEE Softw*. 2019;36:24-28. https://doi.org/10.1109/MS.2019.2921226.
47. Harrison J. Proof style. *TYPES '96*. Berlin/Heidelberg, Germany: Springer-Verlag; 1998:154-172.
48. Fonseca P, Zhang K, Wang X, Krishnamurthy A. An empirical study on the correctness of formally verified distributed systems. *EuroSys*. New York, NY: ACM; 2017:328-343. https://doi.org/10.1145/3064176.3064183.

49. Anand S. Daniel Schwartz-Narbonne shares how automated reasoning is helping achieve the provable security of AWS boot code. Amazon Web Services; 2018. https://web.archive.org/web/20200828152554/https://aws.amazon.com/blogs%/security/automated-reasoning-provable-security-of-boot-code-tlarg/.

How to cite this article: Chong N, Cook B, Eidelman J, et al. Code-level model checking in the software development workflow at Amazon Web Services. *Softw: Pract Exper*. 2021;51:772–797. https://doi.org/10.1002/spe.2949