Code-Level Model Checking in the Software Development Workflow

Nathan Chong
Amazon

Byron Cook
Amazon
UCL

Konstantinos Kallas
University of Pennsylvania

Kareem Khazem
Amazon

Felipe R. Monteiro
Amazon

Daniel Schwartz-Narbonne
Amazon

Serdar Tasiran
Amazon

Michael Tautschnig
Amazon
Queen Mary University of London

Mark R. Tuttle
Amazon

ABSTRACT

This experience report 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, e.g., 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. All proofs discussed in this paper are publicly available on GitHub.

CCS CONCEPTS

• **Software and its engineering** → **Formal software verification**; *Model checking*; Correctness; • **Theory of computation** → Program reasoning.

KEYWORDS

Continuous Integration, Model Checking, Memory Safety.

ACM Reference Format:
Nathan Chong, Byron Cook, Konstantinos Kallas, Kareem Khazem, Felipe R. Monteiro, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. 2020. Code-Level Model Checking in the Software Development Workflow. In *Software Engineering in Practice (ICSE-SEIP '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3377813.3381347

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. They improve 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. 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 four years developing code-level proofs in Amazon Web Services (AWS), 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 9 key modules in the AWS C Common[1] library over a period of 24 weeks (see Sec. 3.2 for a more detailed description of this library). All specifications, proofs, and related artifacts (such as continuous integration reports), described in this paper have been integrated into the main AWS C Common repository on GitHub, and are publicly available at https://github.com/awslabs/aws-c-common/.

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

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICSE-SEIP '20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7123-0/20/05.
https://doi.org/10.1145/3377813.3381347

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

features. Increasingly, the development team has been writing formal specifications themselves and adding them to their code base. In this paper, 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 (Sec. 4.3).

Write proofs-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 make life easier for the verification team: having a recipe for how to write proofs meant that new team members with little familiarity were 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 (Sec. 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 done in the teams' continuous integration;
- 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 at any point in time (Sec. 4.4).

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 importantly, delivering bug fixes helps to build a trustful relationship with the development team, which becomes more responsive and receptive to the verification team's feedback (Sec. 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, plus 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 Sec. 3.2 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 Fig. 1. The flattening that occurs at the end of August represents us having reached our verification target. Now, further verification only occurs on an as-needed basis.

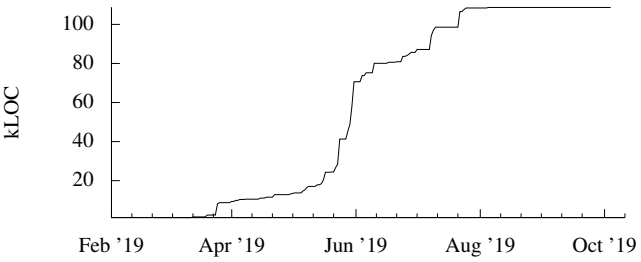

Figure 1: Cumulative number of LOC proven.

Increased rate of bugs found and fixed. As part of this effort, we found and fixed 83 issues (see Tab. 1 and Sec. 4.2 for further details). Our rate of finding bugs increased as we refined our methodology (see Fig. 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.

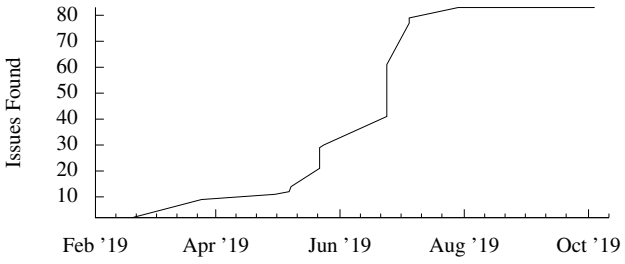

Figure 2: Cumulative number of issues found.

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 (Fig. 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, depicted in Fig. 4. Section 4.3 explains why our style of writing code contracts has been easy for developers to adopt.

2 RELATED WORK

There are a number of powerful static analysis tools such as Infer [6] and Coverity [4]. These are highly effective bug-hunting tools, but they do not prove that a program satisfies a specification. This paper 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. [1], but this paper focuses on improving the interaction between formal verification and development teams.

There has been significant work on proving conformance to specification. For example, Chudnov et al. [9] demonstrate the conformance of the s2n HMAC to a formal HMAC specification. In our world the formal specification does not exist: this paper 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 [10, 15]. As a consequence, several software verification frameworks have emerged in the literature [5]; with many applications in the industrial setting [7, 12, 14, 27]. Some papers [8, 20, 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. [3] 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. [27] 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 [26] describes 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 [14], which is reaffirmed by our work (cf. Sec. 4.4).

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 [29]. 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) [24] and ESC/Java [18] 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 [13], network communication protocols, real-time operating system code, an SDK implementing data structures, and an SDK facilitating principled use of cryptographic primitives.

3.1 CBMC and Program Verification

We verify code using CBMC [11], a bit-precise bounded model checker for C. CBMC is by default a bounded model checker, but we run CBMC with the `--unwinding-assertion` flag, which ensures we have fully checked the model. Given a C program and loop unwinding bounds, 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. The assertions are derived from user-defined properties (using `assert` statements) or automatically generated specifications, such as absence of memory-safety violations. CBMC then uses a SAT solver such as MiniSat [16] 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. The third possibility is that CBMC (including the underlying SAT solver) runs out of resources (e.g., 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.

The main technical challenge of using CBMC is writing the proof harness. These challenges are:

(i) Determining the correct loop-unwinding bounds for the bounded model checker; *(ii)* Correctly constraining the inputs to the entry point being verified., and *(iii)* Using techniques, such as modular verification and limited inputs sizes, to address cases where the solver runs out of resources.

3.2 The AWS C Common Library

We discuss the rest of the paper in the context of proving the memory safety of AWS C Common. AWS C Common is an open-source C99 package that provides cross platform configuration, data structures, and error handling support to a range of other AWS C libraries, including widely used AWS SDKs. The library is the foundation of many security related libraries, such as the AWS Encryption SDK for C. Verifying it is a critical first step towards ensuring the security of those libraries. In particular, we focused on the modules used by the AWS Encryption SDK for C. We have proven 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).

3.3 Properties Proved

We proved that key components of AWS C Common are memory safe, i.e. do not suffer from issues such as buffer overflow, use after free, or invalid pointer dereferences. Memory safety errors are routinely listed among the most critical security concerns by industry groups monitoring CVEs [17, 25, 28, 30]. In addition to memory safety, our proofs guarantee the absence of a subset of undefined C behavior [22], like division-by-zero and arithmetic overflow.

4 METHODOLOGY

This section describes the four pillars of our methodology, and how they have resulted in our successful interaction with developers. We take the position that proofs, invariants, and code contracts need not be arcane and inscrutable; we have actively strived 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.

Section 4.1 presents our proof style and explains why our proofs are familiar and easy to read by developers. Apart from proof material, we often contribute code in the form of bug fixes and other refactoring. Section 4.2 describes these contributions and how they have helped to earn the developers' trust. While these contributions were welcome, the most direct validation of our utility to developers has been their contributing 89 (and counting) code specifications for functions that they wrote. These specifications, like our proofs, are written in the C language; Sec. 4.3 describe the specifications and explains their keen adoption by developers. Finally, our following of a style that developers find familiar doesn't stop at our code and proof contributions: Sec. 4.4 describes how our proof-creation processes emulate those that developers use.

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

```
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):

```
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

We have developed a style of writing proofs that we believe is readable, maintainable, and modular. This style was driven by feedback from developers, and addresses the need to communicate *exactly what we are proving* to developers and users. 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 [21]), 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:

```
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 developed initializers for all verified data structures using a consistent naming scheme:
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. Sec. 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., `length` < `capacity` and `capacity` ≠ 0 ⇒ `buffer` ≠ `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:

```
bool aws_array_list_is_valid(
       const struct aws_array_list *list) {
  if (!list) return false;
  size_t required_size = 0;
  bool required_size_is_valid =
      (aws_mul_size_checked(list->length,
                            list->item_size,
                            &required_size)
       == AWS_OP_SUCCESS);

  bool current_size_is_valid =
      (list->current_size >= required_size);
  bool data_is_valid =
      ((list->current_size == 0 && list->data == NULL)
      || AWS_MEM_IS_WRITABLE(list->data, list->
          current_size));
  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 Sec. 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 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, formal specification enables a proof that the bug is fixed. 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, i.e. 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 and Coverity can generate bug reports faster than developers can fix them [26]. 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 depicts the lines of *non-proof* code we have contributed to the code base, in the form of bug-fixes, refactoring, and other improvements. It neatly complements Fig. 4, which shows *developers* adding contracts to their own code. These results show the result of proof-writers assuming the role of software developers, and *vice versa*: developers contribute their detailed understanding of the program's specification in the form of code contracts, and proof-writers improve the original code base through the knowledge gained during the proof process.

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 methodology underlying it. From

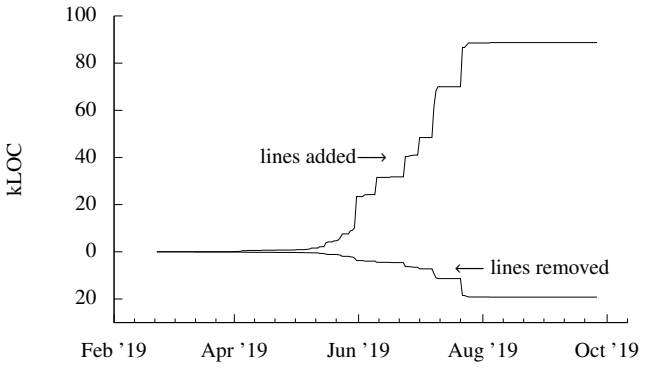

Figure 3: Cumulative number of non-proof LOC added to code base by the verification team.

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 (3%)	31 (37%)	50 (60%)

the point of our relationship with the development team, showing that we have taken the time to understand their code, and by showing our solutions, 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 9 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, and 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 overflow issues in aws_array_list. 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 Sec. 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`.

One of the most unexpected wins from code-refactoring came from the treatment of `static inline` functions, which are used extensively in AWS C Common, but were causing 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 intented 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 Sec. 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, i.e., 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 [19].

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.

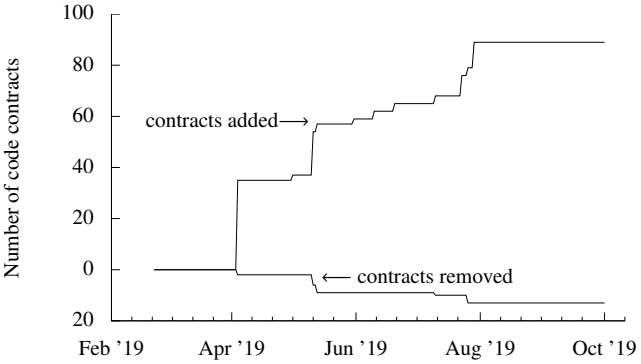

Figure 4: Cumulative number of function contracts the development team added to their own code base.

```
int aws_array_list_get_at_ptr(
        const struct aws_array_list* 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);
}
```

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 assertions when the tests are run and when the code is executed in debugging mode. The function also contains assertions about its postconditions, i.e., it preserves the validity invariant of the input list. This increases our confidence about the validity of the specifications, as they are checked in 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[2] (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 don't have to learn a new specification language. Although it is more verbose to express properties in C and it might be impossible to express some properties at all, this is a trade-off that has to be made so that developers get involved in the process.

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

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.

4.4 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 developers are familiar with. This decreases developers' cognitive burden and allows them to treat our proofs as 'just another test suite,' a very thorough test suite. 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.

4.4.1 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.

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 [2]. Having the proofs as publicly-consumable artifacts provides another motivation for the developers to understand and engage with the proofs.

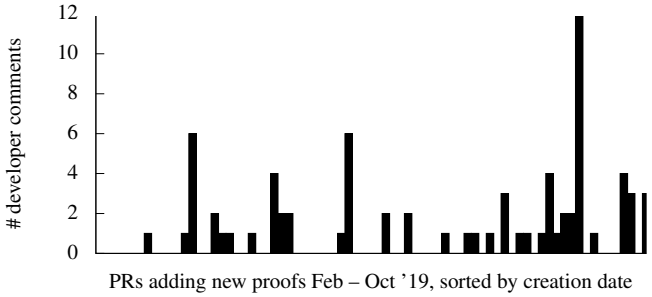

Figure 5: Number of developer comments on each pull request.

4.4.2 Proof Review. The previous paragraph noted that 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; thirteen 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.

4.4.3 Continuous Formal Verification. Rather than 'proving code correct' and moving on, a core aspect of our activity is ensuring that code *stays* correct after our initial proof. We thus created a continuous integration system that runs all proofs in the repository. 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.

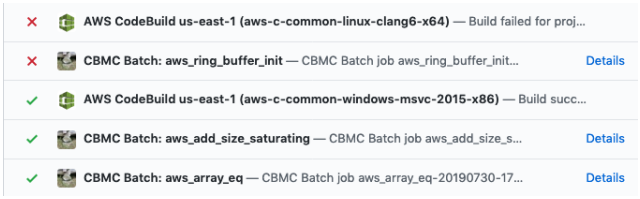

Figure 6: Continuous integration results on GitHub.

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 Fig. 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. Developers thus have 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.

4.4.4 Reliability and Responsiveness of Proof in CI. 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 [26]. O'Hearn mentions the importance of low proof result latency (the time between the developer publishing a change request, and getting feedback from the analysis tool). This number depends not only on the speed of the analysis tool and the size of the proof; it is also increased by the overhead that the CI system introduces.

We thus designed the CI system to take advantage of our proof-writing style, to the end of displaying the proof result to the developer as quickly as possible. Our proofs may all be run in parallel, so our CI system spawns a pool of proof jobs that all begin running immediately and report their result back to GitHub as soon as they are each finished. Therefore, the time taken to run all proof jobs is very similar to the time taken to run a the longest one. The median proof takes 88 seconds to complete. CI startup latency is 359 seconds on average. For some of the projects we work on, our entire proof suite completes before the developers' own tests.

Our adherence to best practices for services means that our CI works reliably; having high uptime is also important to developers' experience with the system. Because the system is event-driven — reacting to developers posting pull requests on GitHub — the demands on the system fluctuate over the week, as Fig. 7 depicts. Developers rarely post pull requests over the weekend, but the system experiences a high load during working hours. We thus used a serverless architecture that is able to seamlessly respond to periods of high demand, and develop the system in a separate beta account that we promote to production once we are confident of its reliability.

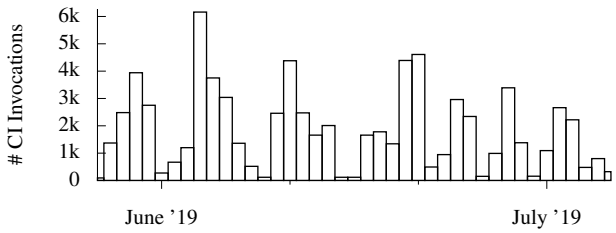

Figure 7: Number of CI invocations per day over 40 days.

5 CONCLUSIONS & 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've 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.

The 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.

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?

REFERENCES

[1] Mark D. Aagaard, Robert B. Jones, Thomas F. Melham, John W. O'Leary, and Carl-Johan H. Seger. 2000. A Methodology for Large-Scale Hardware Verification. In *Formal Methods in Computer-Aided Design*, Warren A. Hunt and Steven D. Johnson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 300–319.
[2] Supriya Anand. 2018. *Daniel Schwartz-Narbonne shares how automated reasoning is helping achieve the provable security of AWS boot code*. Amazon Web Services. https://aws.amazon.com/blogs/security/automated-reasoning-provable-security-of-boot-code-tlarg/
[3] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. 2004. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In *Integrated Formal Methods*, Eerke A. Boiten, John Derrick, and Graeme Smith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–20.

[4] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. https://doi.org/10.1145/1646353.1646374

[5] Dirk Beyer. 2019. Automatic Verification of C and Java Programs: SV-COMP 2019. In *Tools And Algorithms For The Construction And Analysis Of Systems (LNCS)*, Vol. 11429. Springer International Publishing, Cham, 133–155.

[6] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods (Lecture Notes in Computer Science)*, Vol. 6617. Springer International Publishing, Cham, 459–465. https://doi.org/10.1007/978-3-642-20398-5_33

[7] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods (Lecture Notes in Computer Science)*, Vol. 9058. Springer International Publishing, Cham, 3–11. https://doi.org/10.1007/978-3-319-17524-9_1

[8] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, New York, NY, USA, 332–343. https://doi.org/10.1145/2970276.2970347

[9] Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, Aaron Tomb, and Eddy Westbrook. 2018. Continuous Formal Verification of Amazon s2n. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Vol. 10982. Springer International Publishing, Cham, 430–446. https://doi.org/10.1007/978-3-319-96142-2_26

[10] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. 2018. *Handbook Of Model Checking*. Springer International Publishing, Cham, Chapter Introduction To Model Checking, 1–26.

[11] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS (Lecture Notes in Computer Science)*, Vol. 2988. Springer International Publishing, Cham, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15

[12] Byron Cook. 2018. Formal Reasoning About the Security of Amazon Web Services. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 38–47.

[13] Byron Cook, Kareem Khazem, Daniel Kroening, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. 2018. Model Checking Boot Code from AWS Data Centers. In *Computer Aided Verification (CAV)*. Springer International Publishing, Cham, 467–486.

[14] Dino Distefano, Manuel Fahndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Communications of ACM* 62 (2019), 62–70.

[15] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. 2008. A Survey of Automated Techniques for Formal Software Verification. *IEEE Trans. on CAD of Integrated Circuits and Systems* 27, 7 (2008), 1165–1178. https://doi.org/10.1109/TCAD.2008.923410

[16] Niklas Eén and Niklas Sörensson. 2003. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers (Lecture Notes in Computer Science)*, Vol. 2919. Springer, 502–518. https://doi.org/10.1007/978-3-540-24605-3_37

[17] Common Weakness Enumeration. 2019. *CWE Top 25 Most Dangerous Software Errors*. Technical Report. The MITRE Corporation. Retrieved 2019-09 from https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html

[18] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 234–245. https://doi.org/10.1145/512529.512558

[19] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *European Conference on Computer Systems (EuroSys)*. ACM, New York, NY, USA, 328–343. https://doi.org/10.1145/3064176.3064183

[20] Mark Harman and Peter W. O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, Madrid, Spain, 1–23. https://doi.org/10.1109/SCAM.2018.00009

[21] John Harrison. 1998. Proof Style. In *International Workshop on Types for Proofs and Programs (TYPES '96)*. Springer-Verlag, Berlin, Heidelberg, 154–172. http://dl.acm.org/citation.cfm?id=646537.695887

[22] C. Hathhorn and G. Rosu. 2019. Dealing With C's Original Sin. *IEEE Software* 36 (2019), 24–28.

[23] Shriram Krishnamurthi and Tim Nelson. 2019. The Human in Formal Methods. In *Formal Methods – The Next 30 Years*, Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira (Eds.). Springer International Publishing, Cham, 3–10.

[24] Microsoft. 2016. Using SAL Annotations to Reduce C/C++ Code Defects. https://docs.microsoft.com/en-us/visualstudio/code-quality/using-sal-annotations-to-reduce-c-cpp-code-defects

[25] Matt Miller. 2019. *Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape*. Technical Report. Microsoft Security Response Center.

[26] Peter W. O'Hearn. 2018. Continuous Reasoning: Scaling the Impact of Formal Methods. In *ACM/IEEE Symposium on Logic in Computer Science (LICS '18)*. ACM, New York, NY, USA, 13–25. https://doi.org/10.1145/3209108.3209109

[27] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at Google. *Commun. ACM* 61, 4 (2018), 58–66.

[28] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, USA, 48–62.

[29] Nikolai Tillmann and Wolfram Schulte. 2005. Parameterized Unit Tests. In *European Software Engineering Conference*, Vol. 30. ACM, 253–262.

[30] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. 2012. Memory Errors: The Past, the Present, and the Future. In *Research in Attacks, Intrusions, and Defenses*, Davide Balzarotti, Salvatore J. Stolfo, and Marco Cova (Eds.). Springer, Berlin, Heidelberg, 86–106.