Unlocking the Power of Environment Assumptions for Unit Proofs

Siddharth Priya[1], Temesghen Kahsai[2], and Arie Gurfinkel[1]

University of Waterloo[1] and Amazon[2]

Abstract. Clearly articulating the assumptions of the execution environment is crucial for the successful application of code-level formal verification. The process of specifying a model for the environment can be both laborious and error-prone, often requiring domain experts. In contrast, when engineers write unit tests, they frequently employ mocks (tMocks) to define the expected behavior of the environment in which the function under test operates. These tMocks describe how the environment behaves, e.g., the return types of an external API call (stateless behaviour) or the correct sequence of function calls (stateful behaviour). Mocking frameworks have proven to be highly effective tools for crafting unit tests. In our work, we draw inspiration from tMocks and introduce their counterpart in the realm of formal verification, which we term *vMocks*. vMocks offer an intuitive framework for specifying a *plausible* environment when conducting code-level formal verification. We implement a vMock library for the verification of C programs called SeaMock. We investigate the practicality of vMocks by, first, comparing specifications styles in the communication layer of the Android Trusty Trusted Execution Environment (TEE) open source project, and second, in the verification of mbedTLS, a widely used open source C library that provides secure communication protocols and cryptography primitives for embedded systems. Based on our experience, we conclude that vMocks complement other forms of environment models. We believe that vMocks ease adoption of code-level formal verification among developers already familiar with tMocks.

1 Introduction

Formal verification can provide stronger guarantees for software than testing. To scale and efficiently perform code-level formal verification, it is imperative to establish the boundary of a verification task. This delineation defines what falls within the purview of the verification and what lies beyond, essentially giving rise to two core components: the System Under Verification (SUT[1]) and the assumptions governing the SUT's operating environment.

Code-level formal verification in industry is typically lead by verification experts and takes months to become reliable. Even with expertise, effective environment models can take months to years to fine-tune in the context of complex

[1] We use the more familiar SUT as an acronym instead of the more accurate SUV.

industrial codebases [23,5]. The difficulty of designing environments is specially pronounced in new development projects where a reference implementation of the environment may be unavailable.

Contrarily, in test driven development (TDD), developers use testing mocking frameworks (a.k.a., tMocks) to implement environments for SUTs. tMock environments do not specify a complete environment. Instead, they are plausible only for a single function (the SUT) in a unit test. This methodology circumvents the problem of defining a complete test environment operationally (which may be complex to get right and may change over time) by instead describing how it *behaves* (e.g., how many times an environment function may be invoked or the sequence of environment function calls).

Inspired by the success of tMocks, we introduce vMocks – a mocking framework for code-level formal verification. Developers can use vMocks to clearly and efficiently model the environment for the SUT when performing code-level verification. We believe that vMocks will be transformative in enabling the adoption of code-level formal verification early in the software development cycle.

The practice of mocking rests on an effective Domain Specific Language (DSL) for describing externally visible *actions* taken by a mock. It is important to have the right DSL design and implementation. First, the DSL must be *concise* since each unit test has its own environment and a verbose language may result in developer fatigue and introduce specification errors. Second, the DSL must be *embedded* in the host programming language to avoid complexity of translating between the DSL semantics and the SUT. Finally, the specification (mock) must be *executable* with the chosen testing framework.

Existing tMock frameworks satisfy these characteristic with their intended usage scenario. For example, frameworks like cMocka [12] and GoogleMock [25] for C and C++ respectively offer concise DSLs for concrete execution, are embedded in the programming language, and are executable with unit testing frameworks. However, they do not fit well with verification tools. Such tools employ symbolic execution environments. Here, a single unit proof[2] may explore multiple concrete executions and thus the DSL must express non-determinism in the mock environment. Additionally, valid design choices for concrete execution contexts can become very expensive in symbolic executions. For example, cMocka uses a runtime map of function names to mock actions that is queried on each function call to the mock environment. GoogleMock uses dynamic dispatch to wire virtual calls to corresponding mock implementation. Therefore, using existing frameworks has considerable friction.

To address these problems, we have developed, to our knowledge, the first, compile-time mocking framework, called SeaMock. Its GoogleMock-inspired DSL uses C++ meta-programming to provide actions (including non–determinism) to describe vMock behaviour. However, unlike GoogleMock, all function calls are resolved at compile time. Thus, no cost is incurred at runtime.

Our empirical evaluation is on benchmarks in C thus the choice of C++ to write mocks may seem un-intuitive. However, C++ metaprogramming is the

[2] A *unit proof* is a symbolic unit test used in code-level verification.

best possible solution in our opinion. First, C++ enables a compile time DSL that is similar to GoogleMock but constructs a mock function at compile-time. This cannot be accomplished using C macros alone to the best of our knowledge. An alternative would be a custom processor for the DSL. However, this would be complex since it would need to parse C code that may be contained in a DSL program. Second, Using a DSL that allows dropping down to C++ enables full use of C/C++ expressiveness when needed, without the user learning new a language to be productive. Third, though SeaMock uses advanced metaprogramming, the user needs only a basic understanding of template metaprogramming. The complexity is hidden behind an object oriented DSL.

SeaMock, released as an open source library[3], is well suited for code-level verification tools that piggyback on a modern compilation strategy like LLVM [24]. In our evaluation, we use SeaHorn [29], but expect SeaMock to work with other verification tools such as SMACK [31], KLEE [9], or Crux [1]. SeaMock is intended to be used in verification contexts but is not necessarily limited to it and we think it may be useful in unit testing as well.

To test the efficacy of SeaMock, we use it to verify memory safety properties, which are important properties for security and reliability. First, we evaluate how mocks differ from existing environment specification styles by writing different environments for the same unit proof that verifies a message handling function in the Android Trusty Trusted Execution Environment (TEE) [14]. Second, we verify 31 public functions of the SSL messaging component of the mbedTLS [13] library. This component has around 6 000 lines of code. We compare the unit proof development in mbedTLS with similar industrial verification projects [10,17] and find that, *prima facie*, utilizing vMocks makes unit proof development at least three times faster.

In summary, the paper makes the following contributions: (1) introduces the idea of using mocking (vMocks) in the context of code-level formal verification, (2) a novel implementation of vMocks optimized for symbolic environments – SeaMock, and, (3) illustration of efficacy of the vMock philosophy and the SeaMock implementation through evaluation on an open source project.

The rest of the paper is organized as follows: In Section 2, we motivate vMocks by comparing its advantages to other environment styles in the Android Trusty project. In Section 3, we present the design and implementation of SeaMock library. In Section 4, we describe an extensive evaluation of vMocks on verification of memory safety of mbedTLS. We discuss usability issues in Section 5, related work in Section 6, and conclude in Section 7. App. B provides background on mocking and how unit proofs are setup in verification.

2 Motivation

To motivate the utility of mocks, we use our experience from verifying the Android Trusty Trusted Execution Environment (TEE) [14] that provides a framework to build secure applications. These applications must communicate with

[3] `https://github.com/seahorn/seamock`

unsecure applications running outside the TEE. The communication happens through an `ipc` layer. Fig. 1a is a simplified representation of a core function `do_handle_msg` in the `ipc` layer. It reads an incoming message from the TEE kernel and calls an appropriate application function. We want to verify that `do_handle_msg` has no out-of-bounds memory accesses and that it does not modify the message bytes before they are passed to the application. These expectations are encoded in the unit proof in Fig. 1b. The environment consists of the following functions provided by the TEE. Functions `create_channel` and `wait_for_msg` create a collection of channels and choose a channel non-deterministically, respectively. The `get_msg` function returns the length of the incoming message. The `read_msg` function copies the message contents into a provided buffer. The `put_msg` function retires the message indicating that it is not accessed further by the application. We use the SeaHorn BMC engine for verifying `do_handle_msg`. Before looking at various styles for specifying the environment, we provide some background on the verification tool itself.

SeaHorn. The SeaHorn BMC engine [29] is a bit precise bounded model checker for LLVM programs. It uses `Clang` to compile C programs to LLVM bitcode. It provides a number of builtins to enable writing specifications in the style of C functions. **`sassert`** and **`assume`** codify verification assertions and assumptions respectively. Undefined but declared functions starting with `nd_` return non deterministic values of the declared return type. The `is_deref` builtin checks if a memory access is within allocated bounds. The SeaHorn pipeline automatically adds an `is_deref` check before every memory access. The `is_modified` builtin checks if an allocation was modified since `reset_modified` was called on that allocation. The `memhavoc` builtin havocs a newly created memory allocation.

Unit proof. A verification task needs a harness to setup the environment and start the SUT with valid arguments passed to the chosen entry point. A methodology for designing function specific harnesses is discussed in [10]. In [30], this harness specification is called a unit proof, after unit tests. Fig. 1b shows a typical unit proof. The unit proof is packaged as a C `main` function. Note that the specification language for pre-and-post conditions is C, the same language as the function under verification. This is for ease of developers already familiar with the target language. A detailed rationale is found in [30], which calls it Code-As-Specification (CaS). The setup involves creating two channels. One of the channels is chosen using `wait_for_msg` at line 9. Finally the `do_handle_msg` is called, passing in a message handler and the chosen channel. In production, `do_handle_msg` passes the incoming message to the application using a passed-in message handler. For the unit proof, the message handler is defined in `test_msg_handler`. Line 3 checks that the given `msg` arg has not been modified since being reset. The reset is part of the environment and hence not part of Fig. 1b. The `test_msg_handler` is free to return an `error` (< 0) or `ok` ($>= 0$) code. Thus, we can return any non deterministic integer at line 4 using `nd_int`.

Environment specification styles. A unit proof requires an environment to function correctly. The environment, operationally, is a sequence of function calls that supply valid data to the SUT. When it comes to modeling the environment,

```
1  extern int get_msg(int, size_t *);
2  extern int read_msg(int, char *);
3  extern int put_msg(int);
4
5  #define MAX_SIZE 4096
6  typedef int (*msg_handler_t)(char* msg,
7                               size_t msg_size);
8  #define FREE_AND_RET(msg, rc) \
9    do { free(msg); return rc; } while(0)
10 #define ERROR -1
11 int do_handle_msg(
12     msg_handler_t msg_handler,
13     int chan) {
14   char *msg = (char *)malloc(MAX_SIZE);
15   size_t msg_len;
16   int rc = get_msg(chan, &msg_len);
17   if (rc < 0) FREE_AND_RET(msg, ERROR);
18   rc = read_msg(chan, msg);
19   put_msg(chan);
20   if (rc < 0) FREE_AND_RET(msg, ERROR);
21   if (((size_t) rc) < msg_len)
22     FREE_AND_RET(msg, ERROR);
23   rc = msg_handler(msg, msg_len);
24   FREE_AND_RET(msg, rc);
25 }
```

(a) C function under verification.

```
1  static int test_msg_handler(
2      char *msg, size_t msg_size) {
3    sassert(!sea_is_modified((char *)msg));
4    return nd_int(); }
5  int main(void) {
6    // create 2 channels
7    create_channel();
8    create_channel();
9    int chan = wait_for_msg();
10   do_handle_msg(
11       test_msg_handler, chan);
12   return 0;
13 }
```

(b) Unit proof for do_handle_msg.

Fig. 1: An SUT and its unit proof from Android Trusty TEE.

there are several design choices at hand. The environment can either be stateless, where it provides fresh, non–deterministic values with each function call, or stateful, in which case it holds a more comprehensive specification of the environment required for the SUT to function correctly. In the case of a stateless environment, each function call can be represented using a function summary, which is a declarative specification assuming a simple stateless environment. However, if the environment requires state, it can be modeled using a construct referred to as a *fake*. We explore these ways of specifying the environment next.

Option1: Faking the environment. The most general approach to modelling the environment is by creating a fake – a replica of the original environment with simpler internals. A fake environment for `do_handle_msg` is defined in Fig. 2a. This environment fakes incoming messages by providing an array of messages indexed by a channel.

To simulate multiple clients connecting to the TEE, the function `create_channel` creates a channel with a pending message and adds it to a global queue of messages `msgs`. Line 20 chooses a non–deterministic length for the message. The `memhavoc` builtin at line 22 marks the message buffer as containing non–deterministic sequence of bytes. `memhavoc` ensures that `do_handle_msg` does not depend on a specific byte sequence, especially, if we want to check that a message is never altered by the function. The function `wait_for_msg` chooses from channels available currently. Thus, when `create_channel` function is called from the unit proof in Fig. 1b, it choose from one of two channels.

The `get_msg` function may return an error so the return value is non–deterministic at line 37. The subsequent **`sassert`** statement checks that the function pre-conditions are met.

The `read_msg` function checks pre-conditions and copies the (non–deterministic) message from the internal message buffer to the one passed in, i.e. `msg`. It also

invokes a builtin `sea_reset_modified` to mark `msg` as being sensitive to modification. Henceforth, any store to `msg` will taint the memory and cause the corresponding `sea_is_modified` builtin to return true.

Finally, the function `put_msg` retires the message. In our simple fake, we only want a channel to provide a single message so we mark the channel as processed using a counter `g_already_processed_channel`.

Though a fake faithfully represents the actual environment, there are two problems that arise in using them in verification. First, the fake may require complicated internal data structures that are expensive to symbolically execute (in a BMC tool). In this example, the choice of the array data structure is important since it is efficient. A simple associative array data structure suffices because a channel is really an opaque type `handle_t`. Since it is opaque, we are free to map it to `int` and use an array instead of a more expensive data structure. Second, fakes themselves may become complex enough to require their own verification. This adds to the verification burden of the project. To circumvent the problems posed by fakes, we next look at how a minimum environment can be modelled using function summaries.

Option 2: Function summaries. A stateful environment is complex. A stateless environment is easier to model. Figure 2b defines the summaries for our running example. We use function summaries to (1) declare the preconditions for each function in the environment and, (2) to declare valid values that the function can output through output arguments and return statements. Notice that we don't need to provide any summary for `create_channel` and `wait_for_msg` since these functions are never called from `do_handle_msg`. Unfortunately, this summary specification is not strong enough to guarantee correctness of `do_handle_msg`. The summary of `read_msg` chooses a non–deterministic integer for `len` in line 16. The length maybe greater than the size of `msg` buffer. This condition is not handled in `do_handle_msg`. It also leads to undefined behaviour in `memcpy`. Thus, we discover that a correctly functioning environment has a stronger specification – that the length returned by `get_msg` is less–than–or–equal to that used in `read_msg`.

Towards mock environments. We have learned a valuable lesson from the exercise so far. On the one hand, a fake environment works in practice but is expensive to construct. On the other hand, using function summaries enables us to create an inexpensive partial environment but it may need minimum state to satisfy client requirements. Thus, we want a partial environment that is just right – with function summaries that encourage non–determinism and minimum state for the correct functioning of the program function.

For our example, Fig. 2c shows a partial environment that records the length when `get_msg` is called and recalls it for `read_msg` at line 18. It is similar to Fig. 2b in other respects. This satisfies our immediate goals. However, on closer examination, we see that we have really designed the new environment in response to how it is to *behave* with `do_handle_msg`. In particular, it implicitly assumes that `get_msg` is called before `read_msg` so that message length (`g_msg_size`) is set before it is read. In concrete testing, such behavorial specifications are codified using

Expectation (method)	Meaning
`times(Predicate<N>())`	Predicate ($<, >, =$) applied to number of mock calls and N is satisfied.
`returnFn(f)`	Mock function returns output of function f.
`captureArgAndInvoke<N>(f)`	Mock function captures Nth positional argument, applies function f on it.
`invokeFn(f)`	Mock function invokes function f with all arguments.

Table 1: vMock expectation DSL.

tMocks. We thus want their counterpart in verification i.e., vMocks. We also note a key difference between a fake and a mock.

Discussion. We summarise our experience in going from a fake to a mock. Developing a fake required the following steps: (1) understanding the `ipc` layer, corresponding environment interface and implicit contracts between the SUT and environment, (2) implementing a fake that is faithful to external contracts but internally simple for verification, and, (3) developing unit proofs for the fake, finding and fixing bugs. Thus, developing a fake became its own project with the usual develop-debug-fix cycle. In contrast, we developed a vMock environment just for `do_handle_msg`. Thus, it was simple enough to not require verification. Additionally, with SeaMock, the implementation was auto-generated, removing the opportunity for trivial errors. To make vMocks practical, we require a DSL that satisfies two criteria. First, it must be similar to how mocks are specified in concrete tests. This would make it attractive to developers who already have experience with mocks. Second, it must be inexpensive to execute symbolically (e.g., in a BMC tool). Thus, whatever language constructs it provides, must produce minimum overhead during verification. To satisfy these requirements, we describe the SeaMock framework for creating user friendly and efficient vMocks in the next section.

3 The SeaMock framework

Requirements analysis. tMock frameworks are designed to be executed in concrete environments. The same design choices may not be ideal for symbolic environments. For example, in gMock, calls to mock functions are resolved using dynamic dispatch. Resolving such calls dynamically in a symbolic environment is, in general, hard. Another framework, cMocka [12] for C programs aims to have minimal dependencies on libraries or latest compiler features for wide applicability. It uses a runtime map keyed by function names to store expected behaviours. This runtime map is similarly expensive to use in a symbolic environment.

In some cases, it may be possible to adapt symbolic environments to work well with existing mock frameworks. However, we take a novel approach of creating a gMock inspired framework and DSL where function calls are resolved at compile time while retaining the familiar call-chain syntax. This is done using modern C++ template meta-programming[4]. We use C++ meta-programming instead of a custom preprocessor because the template mechanism is supported by industrial C++ compilers is widely available and avoids dependencies on tools that may not be available on a particular platform.

[4] SeaMock is built against C++17.

```c
#define MAX_CHANNELS 10

// The next channel number to initialize
// Incremented on every create_channel call
static int g_next_available_channel;
// The first yet unprocessed channel
// Incremented on every put_msg call
static int g_already_processed_channel;

typedef struct msg {
  char* buf;
  size_t len;
} MSG;

static MSG msgs[MAX_CHANNELS];

int create_channel() {
  sassert(g_next_available_channel <
    MAX_CHANNELS);
  size_t len = nd_size_t();
  char *msg = (char *)malloc(len);
  memhavoc(msg, len);
  int chan = g_next_available_channel++;
  msgs[chan].buf = msg;
  msgs[chan].len = len;
  return chan;}

int wait_for_msg() {
  int channel = nd_int();
  assume(
    channel > g_already_processed_channel &&
    channel < g_next_available_channel);
  return channel;}

int get_msg(int chan, size_t *len) {
  int err_code = nd_int();
  if (err_code < 0) return err_code;
  sassert(chan > 0 &&
      chan < MAX_CHANNELS &&
      len != NULL);
 *len = msgs[chan].len;
  return 0; }

int read_msg(int chan, char *msg) {
  sassert(chan > 0 &&
      chan < g_next_available_channel &&
      msg != NULL);
  memcpy(msg, msgs[chan].buf, msgs[chan].len);
  sea_reset_modified(msg);
  return 0;}

int put_msg(int chan) {
  sassert(chan > 0 && chan <
      g_next_available_channel);
  int err_code = nd_int();
  if (err_code < 0) return err_code;
  free(msgs[chan].buf);
  g_already_processed_channel++;
  return 0;}
```

(a) A fake environment.

```c
#define MAX_SIZE 4096

int create_channel() {/* NOP */ return nd_int(); }
int wait_for_msg() {/* NOP */ return nd_int(); }

int get_msg(int chan, size_t *len) {
  sassert(chan > 0 && len != NULL);
  *len = nd_size_t();
  return nd_int(); // error_code
}

int read_msg(int chan, char *msg) {
  sassert(chan > 0 && msg != NULL);
  char buf[MAX_SIZE];
  memhavoc(&buf);
  size_t len = nd_size_t();
  assume(len < MAX_SIZE);
  memcpy(msg, buf, len);
  sea_reset_modified(msg);
  return nd_bool() ? -1 : len; }

int put_msg(int chan) {
  sassert(chan > 0);
  return nd_int(); }
```

(b) Function summaries for environment.

```c
#define MAX_SIZE 4096

int create_channel() {/* NOP */ return nd_int(); }
int wait_for_msg() {/* NOP */ return nd_int(); }

static size_t g_msg_size;

int get_msg(int chan, size_t *len) {
  sassert(chan > 0 && len != NULL);
  g_msg_size = nd_size_t();
  return nd_int(); }

int read_msg(int chan, char *msg) {
  sassert(chan > 0 && msg != NULL);
  char buf[MAX_SIZE];
  memhavoc(&buf);
  size_t len = nd_size_t();
  assume(len <= g_msg_size);
  memcpy(msg, buf, len);
  sea_reset_modified(msg);
  return len; }

int put_msg(int chan) {
  sassert(chan > 0);
  return nd_int(); }
```

(c) A mock environment.

Fig. 2: Different environment specification styles for `do_handle_msg`.

```
1  static size_t g_msg_size;
2  // *** Begin: Mock arg capture ***
3  static constexpr auto set_ptr_fn_get_msg = [](size_t *len) {
4    *len = nd_size_t();
5    g_msg_size = *len;
6  };
7  static constexpr auto set_ptr_fn_read_msg = [](char *msg) {
8    char *blob = (char *)malloc(g_msg_size);
9    memhavoc(blob, g_msg_size);
10   sassert(msg);
11   memcpy((size_t *)msg, (size_t *)blob, g_msg_size);
12   sea_reset_modified(msg);
13 };
14 // *** End: Mock arg capture ***
15 // *** Begin: mock expect definition ***
16 constexpr auto get_msg_expectations = ExpectationBuilder()
17                                       .times(Eq<1>())
18                                       .returnFn(nd_int)
19                                       .captureArgAndInvoke<1>(set_ptr_fn_get_msg)
20                                       .build();
21
22 constexpr auto read_msg_expectations = ExpectationBuilder()
23                                       .times(Lt<2>())
24                                       .returnFn(MOCK_UTIL_WRAP_VAL(g_msg_size))
25                                       .captureArgAndInvoke<1>(set_ptr_fn_read_msg)
26                                       .build();
27 // *** End: mock expect definition ***
28 extern "C" {
29 MOCK_FUNCTION(get_msg, get_msg_expectations, int, (int, size_t *))
30 MOCK_FUNCTION_W_ORDER(read_msg, read_msg_expectations, MAKE_PRED_FN_SET(get_msg), int, (int, char *))
31 LAZY_MOCK_FUNCTION(put_msg, int, (int))
32 SETUP_POST_CHECKS((get_msg, read_msg, put_msg))}
```

Fig. 3: vMock example using SeaMock.

It is important to note that SeaMock does not aim to be a general purpose mocking framework. It may be useful to adapt it to unit testing, however the current implementation depends on a compiler with modern C++ features and has been tested on the `Clang` compiler toolchain only. We do expect SeaMock to work with other verification tools such as SMACK [31], KLEE [9], or Crux [1] that use LLVM.

Usage. The SeaMock DSL is embedded in C++. It is in the spirit of gMock which in-turn is inspired by jMock [11] and EasyMock [16]. jMock and gMock use a declarative scheme to build expectations on a mock object using a call chain syntax (builder pattern) [19]. The declarative scheme to specify tMock behavior is well established in industry and thus familiar to developers. Therefore SeaMock DSL also uses a declarative builder scheme as shown in Table 1. An `ExpectationBuilder` constructs an expectation object using methods to setup expectations before calling `build()`. For example, the method call `times(Eq<3>())` creates an expectation that a mock function must be called `3` times in all execution paths. A built expectation can be attached to a mock function using `MOCK_FUNCTION(name, expectation, return_type, (argument_type, ...))` where `name` is the function name, `expectation` is the user expectation, `return_type` is the return type of the function, and (`argument_type`,...) a tuple of arguments for the function. To constraint order of execution, the macro `MOCK_FUNCTION_W_ORDER(..., (MAKE_PRED_FN SET(predecessor_fn)), ...)` is used.

In Fig. 3, line 29 shows how the mock `get_msg` function is specified. The expectation is built above in line 16 to line 20. The `ReturnFn` for `get_msg` is set in line 18. It returns a non–deterministic **`size_t`** value. We would like to capture the pointer

$$
\begin{aligned}
insert(M,k,v) &\triangleq M \cup (k,v),\ where\ M : C_v \to C_v, k : C_v, v : C_v \\
atkey(M,k) &\triangleq v \text{ if } (k,v) \in M,\ where\ M : C_v \to C_v, k : C_v, v : C_v \\
putReturnFn(E, retfn) &\triangleq insert(M, ReturnFn, retfn),\ where\ E : C_s \to C_v, ReturnFn : C_s, retfn : C_f \\
getReturnFn(E) &\triangleq atkey(E, ReturnFn),\ where\ E : C_s \to C_v, ReturnFn : C_s \\
skeletal(E) &\triangleq getReturnFn(E)()
\end{aligned}
$$

Fig. 4: Expectation map (M) definition and mechanism for wiring a return function.

$$
\begin{aligned}
putCaptureMap(E, capmap) &\triangleq insert(E, CaptureMap, capmap),\ where\ E : C_s \to C_v, \\
&\qquad CaptureMap : C_s, capmap : C_n \to C_f \\
getCaptureMap(E) &\triangleq atkey(E, CaptureMap),\ where\ E : C_s \to C_v, CaptureMap : C_s \\
args &\triangleq \langle arg_1, arg_2, ..., \arg_n \rangle \\
invokeWithCapturedArg(args, E) &\triangleq \{f_j(args[i_j]) \mid (i_j, f_j) \in getCaptureMap(E)\}
\end{aligned}
$$

Fig. 5: High-level mechanism for `captureArgAndInvoke`.

`len` and set it to a non–deterministic value. This is the first (starting from zero) argument for `get_msg`. This is setup in two steps. First, the effect of the capture is defined using a lambda at line 3. Second, the lambda is wired to the correct positional argument using a setter at line 19. SeaMock provides built–in logic like call cardinality and sequencing that have to be hand–coded otherwise.

In the spirit of need driven development [18], the user only has to specify the absolute minimum required. If a particular expectation is unspecified then a mock function is assembled with defaults. For example, we do not specify the order constraint in Fig. 3, line 29 since we don't expect any other mock function to be called before `get_msg`. The default (no expectation) is assumed when a particular constraint is unspecified. We do specify an order for the `read_msg` mock in line 30 because we expect that if it is called in an execution, then it is always after `get_msg`. Last, if we just want to specify a function summary – that just returns non–deterministic values then a mock can be declared lazy using `LAZY_MOCK_FUNCTION` with no expectations specified as in line 31.

Implementation. SeaMock is implemented using C++ metaprogramming. The metaprogram constructs mock functions from the expectations setup through the DSL. It is developed using the Boost Hana library [15], which provides a functional programming layer over the base metaprogramming environment. Before explaining the details particular to C++, we discuss the high level design of the library. Each mock function requires a mapping of an expectation to an action. For example, a return function expectation results in returning the value of a particular function application on mock function exit. This mapping, called the expectation map, is setup by the user using the SeaMock DSL. The map is a compile-time entity and thus after the C++ template pre-processor executes, a mock function is wired with values gotten from its expectation map and the map ceases to exist. Specifically, there is no lookup at run-time.

We present an example of how a return function expectation is setup internally in the framework in Fig. 4 using a compile-time expectation map (E). Basic types available to the compile time metaprogram are constant naturals (C_n), constant string (C_s) and a function object (C_f). These collectively represent con-

stant value types (C_v). The expectation map stores an action (data) keyed by an expectation, which is of type C_s. The *putReturnFn* function inserts a value (a 0–arity function object) for the *ReturnFn* key. The *getReturnFn* function extracts an inserted function object for the *ReturnFn* key. The *getReturnFn* function is applied in a *skeletal* function, that is the core of a mock function.

In Fig. 5, the mechanism of `captureArgAndInvoke<N>` is defined. Through the DSL, the user provides a mapping (*CaptureMap*) from function argument positions to invoked functions. The framework constructs a tuple (*args*) from the given function arguments. The *skeletal* uses *CaptureMap* and *args* to match functions to arguments and apply the functions left-to-right.

The Boost Hana functional C++ metaprogramming library is used instead of the bare-bones template metaprogramming environment to improve programmer efficiency and correctness since C++ metaprogramming can get tedious and error-prone to work with in advanced use cases. Note that all Hana abstractions are compiled away and thus incur no runtime cost. Hana requires C++14 or beyond. SeaMock uses C++17 to allow the use of `constexpr` lambdas, which are not necessarily used currently but maybe useful in future. Only the mock environment needs a C++ compiler. Once this unit is compiled to LLVM IR, it can be linked to the SUT written in C/C++ that has also been compiled to LLVM IR. Thus, the choice of implementing SeaMock in C++ generally does not limit the SUT source language and C may be used. An exception to this rule is when a C header file contains a function definition that may be accepted by the C compiler but not the C++ compiler. We found this case during evaluation and discuss it in Section 4.

A simplified low-level implementation of the SeaMock internals is shown in Fig. 6a. The core of the SeaMock implementation is the `skeletal` lambda function at line 29. The `skeletal` function creates a mock function with desired actions at compile time based on the expectations setup by the user. The user sets up expectations using the `MOCK_FUNCTION` macro. For simplicity, only a return function expectation is setup. A mock function is constructed in the following way at compile time. (1) A return function is added to the expectation map at Section 3 through the user DSL. (2) A mock function definition is created at line 24. (3) The definition applies the `skeletal` function to the expectation map and the arguments passed to the mock function at line 25. This application creates a specialized `skeletal` lambda for the given mock function.

Developers use tMocks through a fluent interface (builder like pattern). To enable easy adoption of vMocks, we encapsulate the metaprogramming in the same pattern. Specifically SeaMock uses a compile-time builder pattern (available in C++11 and beyond). A simplified version is shown in Fig. 6b. Each call to a setter method adds a key,value pair to the expectations map. The fully constructed expectation maps is returned using the `build` method. In summary, SeaMock provides a familiar interface to developers and they need not know advanced C++ metaprogramming or the Boost Hana framework to be productive. In the next section, we describe our experience in using SeaMock to verify an open source industrial project, mbedTLS.

```
constexpr auto ReturnFn = [](auto ret_fn_val, auto expectations_map) {
  auto tmp = hana::erase_key(expectations_map, RETURN_FN);
  return hana::insert(tmp, hana::make_pair(RETURN_FN, ret_fn_val));
};
// Use: idx=i, type=T1 -> T1 argi
#define CREATE_PARAM(r, data, idx, type) (type BOOST_PP_CAT(arg, idx))
// Use: (int, float, char) ->
// int arg0, float arg1, char arg2
#define UNPACK_TRANSFORM_TUPLE(func, tuple) \
  BOOST_PP_TUPLE_ENUM(BOOST_PP_SEQ_TO_TUPLE( \
  BOOST_PP_SEQ_FOR_EACH_I(func, DONT_CARE, \
      BOOST_PP_TUPLE_TO_SEQ(tuple))))
// Expectation map key
#define RETURN_FN BOOST_HANA_STRING("return_fn")
static constexpr auto Defaults = hana::make_map();
#define CREATE_ND_FUNC_NAME(name, type) \
    BOOST_PP_CAT(nd_, \
    BOOST_PP_CAT(name, BOOST_PP_CAT(type, _fn)))

#define MOCK_FUNCTION(name, \
    expect_map, ret_type, args_tuple) \
    ret_type name( \
    UNPACK_TRANSFORM_TUPLE(CREATE_PARAM, \
        args_tuple)) {\
    return hana::apply(skeletal, expect_map,\
        hana::make_tuple( \
        UNPACK_TRANSFORM_TUPLE(CREATE_ARG, args_tuple))); \
  }
static auto skeletal = [](auto &&expectations_map,
    auto &&args_tuple) {
  auto ret_fn = hana::at_key(expectations_map,
      RETURN_FN);
  return ret_fn();
};
```

(a) Wiring of Simplified `MOCK_FUNCTION` implementation.

```
template
<typename MapType=decltype(DefaultExpectationsMap)>
s ExpectationBuilder {
ate:
MapType expectationsMap;
ic:
// Constructor to initialize with an existing map
constexpr ExpectationBuilder(
    const MapType& map) : expectationsMap(map) {}

// Default constructor
constexpr ExpectationBuilder() :
    expectationsMap(DefaultExpectationsMap) {}

template<typename ReturnFnType>
constexpr auto returnFn(ReturnFnType i) const {
  auto updatedMap = ReturnFn(i, expectationsMap);
  return ExpectationBuilder<
      decltype(updatedMap)>(updatedMap);
}

// Finalize build
constexpr auto build() const {
    return expectationsMap;
}
```

(b) Expectation Builder.

Fig. 6: Seamock implementation. User interacts using the Builder interface (Fig. 6b).

4 Evaluation

To evaluate vMocks, we verified a core component of the mbedTLS open source project using SeaMock and SeaHorn. The mbedTLS project is a C library that implements cryptographic primitives, SSL/TLS and DTLS protocols. Its small code footprint and multiple configuration options makes it suitable for embedded systems. The choice of mbedTLS was guided by our industrial partners, who use the library extensively in their codebases.

In mbedTLS, we undertook the exercise of verifying functions dealing with SSL messages. These functions are present in the `ssl_msg.c` source file which is 6 000 lines of code including comments. The goal of this exercise was to understand the tradeoffs provided by vMocks in formal verification. Thus, our aim was *not* to find bugs in the implementation or prove their absence. Our goals were to study the following: (1) The practicality of developing unit proofs for a *legacy* codebase (without unit tests), (2) the flexibility of using SeaMock to develop environments, and, (3) a qualitative understanding of how code structure affects development of unit proofs.

The `ssl_msg.c` compilation unit contains 44 public (non-static) functions. For our evaluation, we decided to verify low level properties, specially memory safety for these public functions. We developed unit proofs for 31 functions. Of the remaining 13 public functions, 9 functions are trivial – getters, setters and simple boolean checks. The remaining 4 either were specifications themselves or did not have memory accesses. See App. A for details. We were able to develop unit

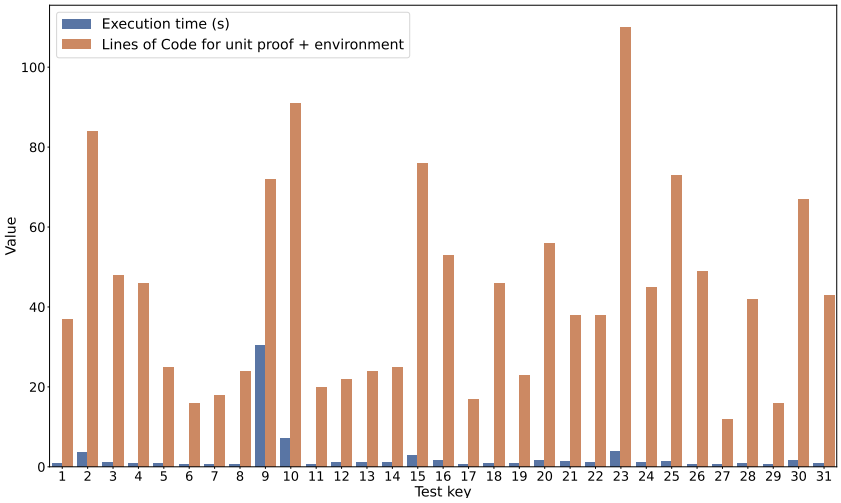

Fig. 7: Performance and size of unit proofs. Key to test name map in App. A, Table 3.

proofs utilizing vMocks for all functions of interest. Almost all unit proofs used the SeaMock DSL, the remaining few had to be hand coded in C because their environments needed a header file (`ssl_misc.h`) that compiled under a C compiler but not under a C++ compiler. SeaMock can be used for all functions once this problem is corrected at source in the mbedTLS project. The experiments were conducted on an Intel(R) Xeon(R) CPU E5-2680 running at 2.70GHz with 64 GiB of main memory. Total build time for all unit proofs was 5 seconds.

The results are given in Fig. 7. All execution times are single seconds except `mbedtls_ssl_read` (key=9) where the unit proof checks that all bytes are copied from one buffer to another. This is a more involved proof and takes closer to a half a minute to run. Of particular interest is the source code complexity for the unit proof and the environment. The average unit proof (harness) size is 24 lines of code and the average environment size is 47 lines of code. Both excluding comments. This is an indication that developing a single unit proof does not involve a lot of coding. The artifacts from the study are open sourced [28]. We discuss our findings next.

Process. Here we describe our process of verifying the SSL message component of the mbedTLS library. First, useful pre-and-post conditions were intuited by breaking the overall verification goal into unit proofs aimed at verifying individual functions. Next, using a list of callees of the function, we decided on functions to mock – we sometimes mocked only a subset of callees to verify more of the production code. Lastly, a decision was made on how *lazy* a mock function should be. Implementations written using `LAZY_MOCK_FUNCTION` did not add any expectations on the mock function. For verifying memory safety, if the production function accessed passed-in buffers, then the mock function checked that such buffers were large enough to accommodate all accesses.

The vMock language. We found that in verifying a legacy codebase (without unit tests) written by a third party was challenging because the pre-and-post

```
1  static size_t nb_bytes;
2
3  constexpr auto invoke_fn_mbedtls_ssl_recv_t = []
4  (void *ctx, unsigned char *buf, size_t len) {
5    if (buf != NULL) {
6      sassert(sea_is_deref(buf, len));
7    }
8    int ret = nd_int();
9    assume(ret <= 0 || ret >= (int)nb_bytes);
10   return ret;
11 };
12 extern "C" {
13 void set_min_recv_bytes(size_t num_bytes) { nb_bytes = num_bytes; }
14 constexpr auto expectations_mbedtls_ssl_recv_t =
15     seamock::ExpectationBuilder()
16         .times(seamock::Lt<2>())
17         .invokeFn(invoke_fn_mbedtls_ssl_recv_t)
18         .build();
19 MOCK_FUNCTION(ssl_recv_fn, expectations_mbedtls_ssl_recv_t, int,
20     (void * /* ctx */, unsigned char * /* buf */, size_t /* len */))
21 SETUP_POST_CHECKS((ssl_recv_fn))}
```

Fig. 8: `ssl_recv_fn` mock used in the unit proof for `ssl_msg_fetch_input`.

Project	**Unit Proofs(U)**	**Weeks(W)**	**Persons(P)**	**Rate** $= U/(W \times P)$
mbedTLS (vMocks)	31	5	1	6.2
aws-c-common	171	24	3	2.4
firecracker-vmm	27	30	2	0.5

Table 2: Comparison of engineer efficiency across verification projects.

conditions were not made explicit and the verification engineer could not explicate what these invariants could be. Thus we found that coming up with the declarative behaviour of an environment was complex. Instead, it was practical to use the `InvokeFn` action to invoke a simple function stub. Thus, most environments were written as such rather than using more behaviorial actions like `After` and `ReturnFn`. We posit that in a new software project, a rich set of actions in the mocking DSL is more useful since the environment is yet unspecified and behaviors are easier to model.

A typical example of a mock function is shown in Fig. 8. Here we see the use of `InvokeFn` to invoke a function at line 3. The function checks that the passed-in `buf` can be dereferenced upto `len` bytes. We also want to provide at least as many bytes as requested by the unit proof. To wire this logic in, the unit proof calls the `set_min_recv_bytes` function before calling the SUT. Now when `ssl_recv_fn` is called, we make sure that atleast that many bytes are returned.

Outcome and learning. A single verification engineer with no previous familiarity with the codebase was able to verify 31 functions is the SSL message component of mbedTLS in five weeks. To put this into perspective, we looked at similar projects using the unit proof methodology of code-level verification for BMC but does not use a DSL based mocking methodology – the aws-c-common verification project from AWS [10] in C and the Firecracker Virtual Machine Monitor verification project [17] in Rust, again from AWS. The comparative rate of developing unit proofs is given in Table 2. The properties under verification and the complexity of the codebase vary across projects. Thus it is hard to make an apples-to-apples comparison. These numbers are only indicative that vMocks may be useful, especially bootstrapping a verification project.

With mbedTLS, it was challenging to determine pre-and-post conditions for functions since most of the considered functions did not specify them in com-

ments or employ unit tests. In essence, verifying legacy code using unit proof faces similar problems as faced by developers retrofitting unit tests into such codebases. The next section discusses the use case for vMocks based on industrial experience with tMocks.

5 Usability of Mocks

tMocks and TDD. tMocks are an essential tool for test driven development (TDD). TDD is a methodology where unit tests are written before the SUT function. The test initially fails since the function is not implemented. The function and tests are then refined in lockstep until all the unit tests pass. tMocks play a vital role in this refinement process since they specify how the function interacts with the environment. There has been continued discussion on the efficacy of TDD over the last twenty years [6,26,32]. TDD is not a silver bullet to improve software quality. However, used judiciously, it has shown to improve software quality outcomes [27]. Moreover, it is a promoted methodology of code development in industry [33].

In TDD, tMocks inform the design of object interfaces. As [18] states, "By testing an object in isolation, the programmer is forced to consider an object's interactions with its collaborators in the abstract, possibly before those collaborators exist. TDD with Mock Objects guides interface design by the services that an object requires, not just those it provides." Thus, tMocks compel the SUT to evolve into a form that is robust and well tested. In contrast, current approaches to formal verification assume the SUT has already reached a robust design, having undergone sufficient testing. It is in this context that formal verification is applied to software, often by a team of verification specialists who are different from the original developers [9,10,20].

Using the lessons of TDD, this work present vMocks, to lower the startup costs of formal verification such that developers themselves use formal verification early on in software projects. With mocking, the program under verification and the environment need not be fully fleshed out to gain confidence in design and implementation. SUT functions can be verified in isolation as they are developed, i.e., using verification driven development (VDD).

Lifecycle of Mocks. We know from industrial experience that tMocks become difficult to maintain once the project reaches maturity and environment interfaces are fully fleshed out[5]. At this point, investing time and effort in fakes is better for the long term. We imagine that vMocks will have a similar lifecycle. They will be invaluable early on in a project when developers are designing a system incrementally or refactoring a legacy codebase. However, once a SUT is fully fleshed out with the aid of mocking, using fake environments will be natural and serve the project better in the long run. We see VDD supplementing TDD since the startup cost of TDD will be lower. Where VDD will find its utility rel-

[5] https://abseil.io/resources/swe-book/html/ch13.html#prefer_realism_over_isolation

ative to TDD is unexplored. This paper contributes a methodology and tooling to conduct research in this area.

6 Related Work

This work stands on both research in formal verification and the practice of test driven development. Designing abstract environments is an integral part of any code-level formal verification effort. Environments were described using function summaries in the static driver verification project (SDV) [5] from Microsoft Research. As another design example, a fake filesystem was implemented by the KLEE authors for verifying coreutils [9]. Angelic verification [23] from MSR, is a methodology to auto generate environments. It was a response to the long lead times in delivering software environments in the SDV project. However, automatic generation of environments is a program synthesis problem and is inherently hard for complex environments.

In test driven development, the need to test with an environment before it can be defined led naturally to tMocks [18]. Their applicability with respect to test driven development has been studied over the last two decades [6,26,32]. More interestingly, tMocks have been widely promoted in industry. There is a large body of advice of how to use them, including caveats [33]. In this vein, the reader may find various ToTT [2] blog posts (e.g., [3,4]) from Google interesting. It is hard to maintain parity between a tMock and the real environment as a project evolves. To mitigate the effort required to maintain tMocks, there has been interest in academia [8] and industry (e.g., [7,21]) on automatically generating tMocks. Similar techniques may be useful in generating vMocks.

Software verification, including modeling environments for verification, remains a specialized endeavour as highlighted in a recent case study by AWS [10]. To our knowledge, there has been no published work on adapting testing mocks (tMocks) for verification (vMocks). This presentation takes the first step towards remedying the situation.

7 Conclusion

This work takes inspiration from mocks in testing (tMocks) and introduces their counterpart in verification (vMocks). Like tMocks, vMocks define environments behaviorally. We hope that vMocks catalyze formal methods in software development. The presented case study uses vMocks for writing proofs using vMocks for public functions of the SSL messaging layer of the mbedTLS project with low engineering effort. For the case study, we developed SeaMock, an open-source vMock DSL library embedded in C++ for the SeaHorn BMC tool. A future direction would be to adapt static analysis of tools like SeaHorn to reason about runtime polymorphism efficiently enabling the use of tMock frameworks like gMock in symbolic execution. Alternatively, the use of SeaMock could be extended to concrete testing.

References

1. Crux (2023), https://crux.galois.com/
2. Authors, T.: Introducing "Testing on the Toilet" (2007), https://testing.googleblog.com/2007/01/introducing-testing-on-toilet.html
3. Authors, T.: Tott: Partial mocks using forwarding objects (2009), https://testing.googleblog.com/2009/02/tott-partial-mocks-using-forwarding_19.html
4. Authors, T.: Testing on the toilet: Don't mock types you don't own (2020), https://testing.googleblog.com/2020/07/testing-on-toilet-dont-mock-types-you.html
5. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: Berbers, Y., Zwaenepoel, W. (eds.) 2006 EuroSys Conference, Leuven, Belgium, April 18-21, 2006, Proceedings. pp. 73–85. ACM (2006). https://doi.org/10.1145/1217935.1217943
6. Beck, K.: Test-driven development: by example. Addison-Wesley Professional (2003)
7. Boeira, M.: Using sourcery to automatically generate mocks (2019), https://medium.com/@mdboeira/using-sourcery-to-automatically-generate-mocks-73ced75b2863
8. Bragg, N.F.F., Foster, J.S., Roux, C., Solar-Lezama, A.: Program sketching by automatically generating mocks from tests. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12759, pp. 808–831. Springer (2021). https://doi.org/10.1007/978-3-030-81685-8_38, https://doi.org/10.1007/978-3-030-81685-8_38
9. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) 8th USENIX Symposium, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings. pp. 209–224. USENIX Association (2008), https://dl.acm.org/doi/10.5555/1855741.1855756
10. Chong, N., Cook, B., Eidelman, J., Kallas, K., Khazem, K., Monteiro, F.R., Schwartz-Narbonne, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: Code-level model checking in the software development workflow at amazon web services. Softw. Pract. Exp. **51**(4), 772–797 (2021). https://doi.org/10.1002/spe.2949
11. jMock Developers: jmock (2019), http://jmock.org/
12. cmocka Developers.: cmocka website (2023), https://cmocka.org/
13. mbedTLS Developers: mbedtls project (2023), https://github.com/Mbed-TLS/mbedtls
14. Developers, A.: Trusty tee (2023), https://source.android.com/docs/security/features/trusty
15. Developers, B.H.: Boost.hana (2023), https://boostorg.github.io/hana/index.html
16. Developers, E.: Easymock (2022), https://easymock.org/
17. Developers., K.: Using kani to validate security boundaries in aws firecracker (2024), https://model-checking.github.io/kani-verifier-blog/2023/08/31/using-kani-to-validate-security-boundaries-in-aws-firecracker.html
18. Freeman, S., Mackinnon, T., Pryce, N., Walnes, J.: Mock roles, objects. In: Vlissides, J.M., Schmidt, D.C. (eds.) OOPSLA 2004, Vancouver, BC, Canada. pp. 236–246. ACM (2004). https://doi.org/10.1145/1028664.1028765

19. Freeman, S., Pryce, N.: Evolving an embedded domain-specific language in java. In: Tarr, P.L., Cook, W.R. (eds.) OOPSLA 2006, Portland, Oregon, USA. pp. 855–865. ACM (2006). https://doi.org/10.1145/1176617.1176735
20. Hamza, J., Felix, S., Kuncak, V., Nussbaumer, I., Schramka, F.: From verified scala to STIX file system embedded code using stainless. In: Deshmukh, J.V., Havelund, K., Perez, I. (eds.) NFM 2022, Pasadena, CA, USA. Lecture Notes in Computer Science, vol. 13260, pp. 393–410. Springer (2022). https://doi.org/10.1007/978-3-031-06773-0_21
21. Kraus, S.: Using sourcery to automatically generate mocks (2019), https://medium.com/iqoqo-engineering/this-library-uses-your-jest-tests-to-generate-mocks-c07c322c58e3
22. Kula, E., Greuter, E., van Deursen, A., Gousios, G.: Factors affecting on-time delivery in large-scale agile software development. IEEE Trans. Software Eng. **48**(9), 3573–3592 (2022). https://doi.org/10.1109/TSE.2021.3101192
23. Lahiri, S.K., Lal, A., Gopinath, S., Nutz, A., Levin, V., Kumar, R., Deisinger, N., Lichtenberg, J., Bansal, C.: Angelic checking within static driver verifier: Towards high-precision defects without (modeling) cost. In: FMCAD 2020, Haifa, Israel. pp. 169–178. IEEE (2020). https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_24
24. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis and transformation. pp. 75–88. San Jose, CA, USA (Mar 2004)
25. LLC., G.: Googletest user's guide (2023), https://google.github.io/googletest/
26. Munir, H., Wnuk, K., Petersen, K., Moayyed, M.: An experimental evaluation of test driven development vs. test-last development with industry professionals. In: Shepperd, M.J., Hall, T., Myrtveit, I. (eds.) 18th International Conference on Evaluation and Assessment in Software Engineering, EASE '14, London, United Kingdom, May 13-14, 2014. pp. 50:1–50:10. ACM (2014). https://doi.org/10.1145/2601248.2601267
27. Nagappan, N., Maximilien, E.M., Bhat, T., Williams, L.A.: Realizing quality improvement through test driven development: results and experiences of four industrial teams. Empir. Softw. Eng. **13**(3), 289–302 (2008). https://doi.org/10.1007/s10664-008-9062-z
28. Priya, S., Gurfinkel, A., Kahsai, T.: verify-mbedtls-artifact (Jun 2024). https://doi.org/10.6084/m9.figshare.26122222, https://figshare.com/articles/dataset/verify-mbedtls-artifact/26122222/1
29. Priya, S., Su, Y., Bao, Y., Zhou, X., Vizel, Y., Gurfinkel, A.: Bounded model checking for LLVM. In: Griggio, A., Rungta, N. (eds.) FMCAD 2022, Trento, Italy. pp. 214–224. IEEE (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_28
30. Priya, S., Zhou, X., Su, Y., Vizel, Y., Bao, Y., Gurfinkel, A.: Verifying verified code. Innov. Syst. Softw. Eng. **18**(3), 335–346 (2022). https://doi.org/10.1007/s11334-022-00443-9, https://doi.org/10.1007/s11334-022-00443-9
31. Rakamaric, Z., Emmi, M.: SMACK: Decoupling Source Language Details from Verifier Implementations. In: CAV 2014, Vienna. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 106–113 (2014)
32. Romano, S., Zampetti, F., Baldassarre, M.T., Penta, M.D., Scanniello, G.: Do static analysis tools affect software quality when using test-driven development? In: Madeiral, F., Lassenius, C., Conte, T., Männistö, T. (eds.) ESEM '22: ACM / IEEE Helsinki, Finland. pp. 80–91. ACM (2022). https://doi.org/10.1145/3544902.3546233

33. Wright, H., Winters, T.D., Manshreck, T.: Software Engineering at Google. O'Reilly (2020)

A Appendix

Key	Test name
1	ssl_msg_flight_free
2	ssl_msg_fetch_input
3	ssl_msg_recv_flight_completed
4	ssl_msg_handle_message_type
5	ssl_msg_transform_free
6	ssl_msg_dtls_replay_update
7	ssl_msg_set_outbound_transform
8	ssl_msg_get_record_expansion
9	ssl_msg_read
10	ssl_msg_flight_transmit
11	ssl_msg_start_handshake_msg
12	ssl_msg_set_inbound_transform
13	ssl_msg_parse_change_cipher_spec
14	ssl_msg_prepare_handshake_record
15	ssl_msg_decrypt_buf
16	ssl_msg_write_record
17	ssl_msg_dtls_replay_check
18	ssl_msg_buffering_free
19	ssl_msg_write_change_cipher_spec
20	ssl_msg_write
21	ssl_msg_close_notify
22	ssl_msg_flush_output
23	ssl_msg_encrypt_buf
24	ssl_msg_send_alert_message
25	ssl_msg_write_handshake_msg_ext
26	ssl_msg_check_record
27	ssl_msg_pend_fatal_alert
28	ssl_msg_finish_handshake_msg
29	ssl_msg_send_flight_completed
30	ssl_msg_read_record
31	ssl_msg_handle_pending_alert

Table 3: Key to testname map for Fig. 7.

The 4 function that either were specifications themselves or did not have memory accesses were `mbedtls_ssl_reset_in_out_pointers`, `mbedtls_ssl_update_in_pointers`, `mbedtls_ssl_update_out_pointers`, `mbedtls_ssl_write_version`.

B Background

GoogleMock DSL. It is useful to see how expectations are setup in a mocking DSL. We thus illustrate with an example usage of GoogleMock which is a

```
1 class Turtle {
2   ...
3   virtual ~Turtle() = default;
4   void PenUp() {...}
5   virtual void PenDown() = 0;
6 };
```

(a) Class to be mocked.

```
1  #include "turtle.h" // Brings in Turtle class
2  #include "gmock/gmock.h" // Brings in gMock.
3  #include "gtest/gtest.h"
4
5  class MockTurtle : public Turtle {
6   public:
7    ...
8    MOCK_METHOD(void, PenDown, (), (override));
9  };
10 using ::testing::AtLeast;
11
12 TEST(PainterTest, CanDrawSomething) {
13   MockTurtle turtle;
14   EXPECT_CALL(turtle, PenDown()).Times(AtLeast(1));
15   Painter painter(&turtle);
16   EXPECT_TRUE(painter.DrawCircle(0, 0, 10));
17 }
```

(b) A gMock usage example.

Fig. 9: An adaptation from GoogleTest documentation.

popular mocking framework for C/C++. The DSL language is typical of other mocking frameworks also. Fig. 9a shows a `Turtle` class that is the environment we want to test against. The SUT is the `Painter` class. The function to be unit tested is `Painter::DrawCircle`, which needs the `Turtle` environment. The unit test is shown in Fig. 9b. We create a mock `Turtle` at line 5. The methods to be mocked are defined using `MOCK_METHOD`. Moving to the actual unit test `CanDrawSomething`, a behavior expectation on the mock turtle is defined at line 14. We expect that the subsequent `painter.DrawCircle` call will invoke the `Turtle::PenDown` method at least once. If this expectation is not met then the test fails.

From the example, we make two key observations. First, the tMock is partial, i.e., it need not define all methods of a class. For example, `PenUp` is not mocked since we never expect it to be called in the unit test. Second, a tMock specifies only how the environment is to behave with external observers; it does not pretend to mimic the environment operationally.

Overall gMock works well in the context it was designed for since: (1) the gMock DSL enables concise specification of environment behaviours in terms of actions, (2) the DSL uses standard C++ tooling enabling a familiar coding environment for the same developers who write production code, and, (3) the framework produces executable code that fits together with the unit proof to make a closed testing environment.

Bounded Model Checking and SeaHorn. Code level Bounded Model Checking (BMC) is a path sensitive and precise technique to verify software implementations. The *bounded* part of the name indicates that the states of a program are checked only to a certain depth. Thus, loops are bound to a depth determined by the user. BMC is used to check safety properties of software. In case a bad state is entered (within bounds), the technique generates a finite length counterexample showing how the state was reached.

The SeaHorn BMC engine [29] is a bit precise bounded model checker for LLVM programs based on the above principles. It uses `Clang` to compile C/C++ programs to LLVM bitcode. It provides a number of builtins to enable writing specifications in the style of C/C++ functions, for example, **`sassert`** and **`assume`** codify verification assertions and assumptions respectively. Undefined but de-

```
1 unsigned water(
2     uint32_t qty) {
3  uint32_t p = 0;
4  while (p < qty) {
5    p += pour();
6  }
7  return p;
```

(a) Water plant SUT.

```
1 uint32_t pour() {
2   uint32_t v =
3     nd_uint32_t();
4   assume(v < 3);
5   return v;
6 }
```

(b) Environment for SUT.

```
1 int main(void) {
2   //pre
3   uint32_t e = nd_uint32_t();
4   assume(e <= 10);
5   //call SUT
6   uint32_t p = water(e);
7   //post
8   sassert(p >= e);
9 }
```

(c) Unit proof for SUT.

Fig. 10: An example unit proof setup in a plant watering program.

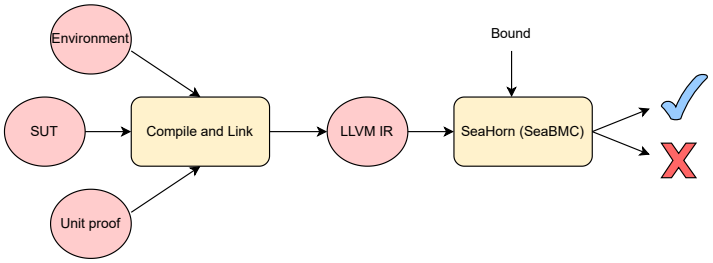

Fig. 11: How a proof is assembled. SeaHorn outputs pass (UNSAT) or fail (SAT).

clared functions starting with `nd_` return non–deterministic values of the declared return type. The `is_deref` builtin checks if a memory access is within allocated bounds. The SeaHorn pipeline automatically adds an `is_deref` check before every memory access. The pair of `reset_modified`, `is_modified` builtins mark and check unexpected mutation of buffers designated as read-only. The `memhavoc` builtin *havocs* a newly created memory allocation, filling it with non–deterministic byte(s).

Unit proof. BMC needs a harness to setup and check pre–and–post conditions respectively, and, call the SUT with valid arguments passed to the chosen entry point. A methodology for designing harnesses on a per function basis in discussed in [10]. In [30], this harness specification is called a *unit proof*, after unit tests.

We explain the concept with an example in C/C++ shown in Fig. 10. The `water` function in Fig. 10a is the SUT. It takes in a quantity of water to pour on the plant and calls an environment function `pour` that releases small amounts of water to the plant repeatedly in a loop. The loop exits when the required quantity of water has been released. The `pour` environment function is defined in Fig. 10b. It creates a non–deterministic unsigned value in line 3, constraints it using an **`assume`** and returns one of `0`, `1`, or, `2`. Finally, the setup is closed by a unit proof in Fig. 10c inside a C/C++ `main` entrypoint function. First the function sets the pre–condition that the quantity of water is between `0` and `10` in line 3 and line 4. It then calls `water` and checks the post–condition that atleast the required quanity of water was released. Note the following aspects of unit proofs.

(1) The specification language for pre-and-post conditions is C/C++, the same language as the function under verification. This is for ease of developers already familiar with the target language. A detailed rationale is found in [30], which calls it Code-As-Specification (CaS).
(2) In this example `pour` is considered the environment. However, where to partition the SUT and the environment is a design choice. At one extreme, all

immediate callee functions of an SUT may be an environment. Alternatively, an obvious lower layer (e.g., OS API) may be the environment. Though, in all cases, an environment must be chosen for the verification to be practical.

(3) The precision of BMC usually comes at a cost of scalability. As increasing amounts of code is brought into the purview of a unit proof, tool running times become impractical. Thus, constructing a unit proof requires engineering to get the right amount of code covered within the limits of tools.

The third aspect is supported by data from two recent projects that use the unit proof methodology. The verification of the aws-c-common library from AWS resulted in 171 unit proofs and was developed by 3 engineers over 24 weeks [10]. Another project from AWS that verified the firecracker virtual machine monitor produced 27 unit proofs in 30 weeks using 2 engineers [17]. These are long lead times to build confidence in software implementations where product release cadence can be a couple of weeks to a couple of months [22]. In this paper, we show that mocks can ease writing unit proofs and, hopefully, shorten development times for verification projects.

Figure 11 illustrates how the SUT, environment, and, unit proof is assembled for verification by SeaHorn. First the different parts are compiled and linked together using the `Clang` compiler toolchain to create a closed LLVM IR program. Then, SeaHorn is run with a bound set by the user. For the `water` unit proof, the bound will be set to 10. SeaHorn generates verification conditions in the SMT language. Finally, these are given to a solver (e.g., z3), which returns either UNSAT (proof holds) or SAT (proof has a counterexample). In the case a counterexample is found, SeaHorn generates a program trace of how the safety property is violated.