Software Verification with SMT

Prof. Arie Gurfinkel
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

UNIVERSITY OF WATERLOO CERTORA

Acknowledgements

Post-docs

- Isabel Garcia-Contreras
- Yuyan Bao

PhD Students

- Siddharth Priya
- Yusen Su
- Joseph Tafese

MSc Students

- Scott Wesley
- Xiang Zhou
- Jakub Kuderski

Collaborators

- Yakir Vizel (Technion)
- Sharon Shoham (Tel Aviv University)
- Temesghen Khasai (Amazon)
- Jorge A. Navas (Certora)
- Chandrakana Nandi (Certora)
- Giuliano Losa (Stellar)
- Graydon Hoare (Stellar)

Why Formal Verification – Because AI

https://vitalik.eth.limo/general/2026/05/18/fv.html

A shallow dive into formal verification

2026 May 18 See all posts

vitalik.eth
@VitalikButerin

This year, the EF is decreasing its budget by roughly 40%, which entails stop finalizing. We are increasingly exploring moving more pieces of the protocol to a different security strategy: AI-assisted formal verification.

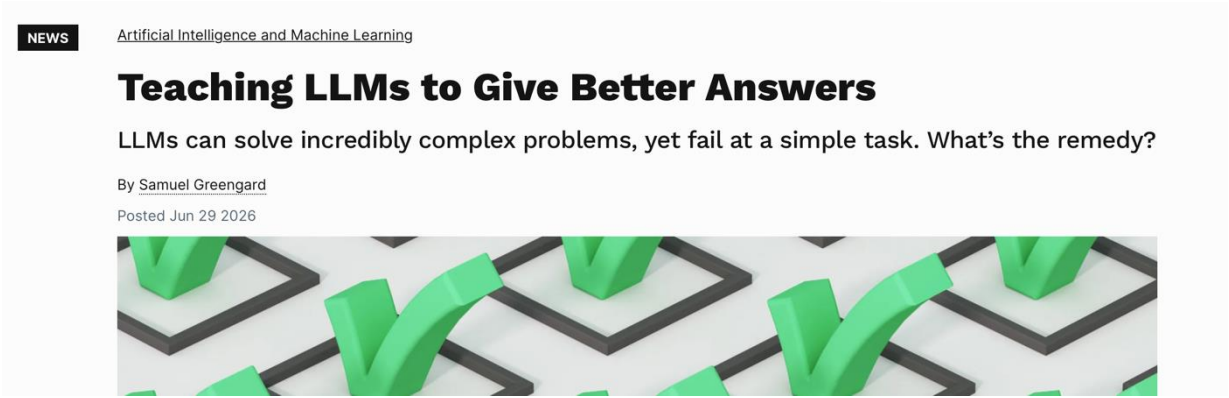
https://www.developersdigest.tech/blog/what-hacker-news-gets-right-about-ai-coding-agents-2026

TL;DR

Hacker News keeps arguing about Claude Code, Codex, skills, MCP, and orchestration. Under the noise, the same four truths keep surfacing: workflows matter more than demos, verification is the bottleneck, skills beat prompts, and orchestration matters more than raw autonomy.

FV == trusted AI generated code

UNIVERSITY OF WATERLOO



NEWS

Artificial Intelligence and Machine Learning

Teaching LLMs to Give Better Answers

LLMs can solve incredibly complex problems, yet fail at a simple task. What's the remedy?

By Samuel Greengard

Posted Jun 29 2026

To be sure, the path to better reasoning is emerging. Concluded Barrett: "Without greater rigor, we risk being buried by mountains of buggy code and inaccurate information. Using these methods, we have a tremendous opportunity to bring rigorous reasoning to software and systems through AI."

Samuel Greengard *is an author and journalist based in West Linn, OR, USA.*

https://cacm.acm.org/news/thats-logical-teaching-llms-to-give-better-answers/

WHAT IS FORMAL VERIFICATION

FV is Theorem Proving

FV via foundational theorem prover

- LEAN, Rocq, Isabelle, HOL, Agda, PVS, ACL2, …

Pros: Limited by your imagination and skill

- applies to foundational theorems as well as concrete code
- one tool to rule them all

Cons: (Very) hard to use

- most significant proof == paper / thesis
- LLMs might be changing this

```
-- reverse is involutive
theorem rev_rev (xs : List α) :
    xs.reverse.reverse = xs := by
  induction xs with
  | nil => rfl
  | cons x xs ih =>
    simp [reverse_cons, ih]
```

theorem reverse_reverse — proof in Lean 4

leanprover.github.io/theorem_proving_in_lean4

UNIVERSITY OF WATERLOO

FV is Auto-active Verification

FV via specialized theorem prover for code

- Dafny, Why3, Verifast, Viper, VeriFast, KeY, Verus, SPARK, …

Pros: verification is part of programming

- very expressive with almost unlimited possibilities
- highly automated via SMT solvers

Cons: usability

- high annotation/verification burden
- flaky automation – SMT brittleness

```
// SMT discharges this automatically
method Abs(x: int) returns (y: int)
  ensures y >= 0
  ensures y == x || y == -x
{
  if x < 0 { y := -x; }
  else { y := x; }
}
```

method Abs — verified in Dafny

dafny.org/latest/OnlineTutorial/guide

FV is Software Model Checking

FV via automatic reduction to SAT/SMT

- CBMC, SeaHorn/SeaBMC, ESBMC, Kani, **see** SV-COMP for many additional tools

Pros: verification is highly automated (and intuitive)

- unit proofs ~ unit tests
- reasons about real unsafe ugly code directly

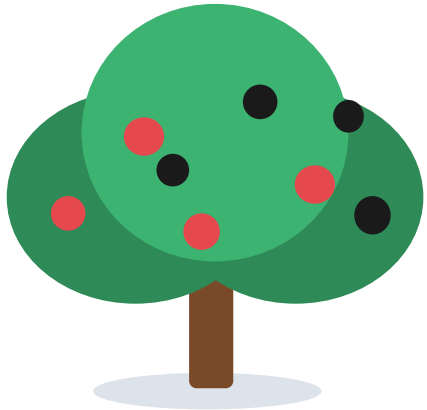
Cons: scalability (and unpredictability)

- needs (some) expertise to use effectively
- need more expertise to close scalability gaps

```c
#include <seahorn/seahorn.h>
int main() {
  struct aws_array_list list;
  initialize_bounded_array_list(&list);
  assume(aws_array_list_is_valid(&list));
  void *val = bounded_malloc_havoc(sz);
  sea_tracking_on();
  // operation under verification
  if (!aws_array_list_front(&list, val)) {
    sassert(AWS_BYTES_EQ(val, list.data, sz));
    sassert(list.length);
  }
  sassert(aws_array_list_is_valid(&list));
  return 0;
}
```

SeaHorn unit proof — aws-c-common array_list_front

github.com/seahorn/verify-c-common



Case Study: which tool to use

VERIFYING RED-BLACK TREE

Red-Black Trees (RBT)

Self-balancing binary search tree

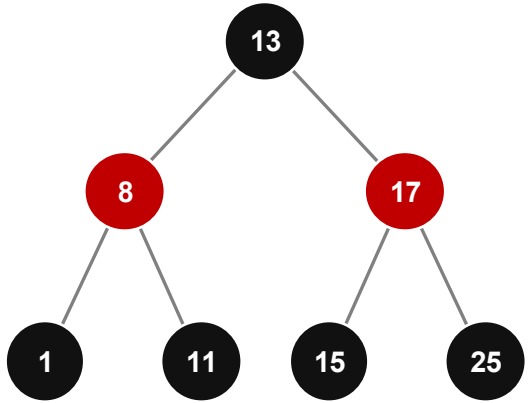
- every node carries one extra bit of color — red or black

Balance rules:

- the root is black, a red node's children are black (no two reds in a row), and every path from a node to its leaves crosses the same number of black nodes (its black height)

Thm: longest path is at most twice the shortest

- height stays O(log n)



Example red–black tree: root is black; no red node has a red child.

en.wikipedia.org/wiki/Red–black_tree

Simple-looking data structure, but notoriously hard to get right!

Client: Verify that our implementation of RBT is correct

RBT: Foundational Correctness

Show key properties of RBT

- root black · no red node has a red child · equal black-height on every path · maintains BST order

Reason at the conceptual level

- argue over the abstract tree datatype — not pointers, memory, or client structs

Thm: invariants ⇒ height ≤ $2 \cdot \log_2(n + 1)$

- balanced shape gives O(log n) search, insert, delete

Best tool: foundational theorem prover

- Lean, Rocq, Isabelle — define the datatype, prove each operation preserves the invariants

Problem: the proof is about the foundational theory, not the actual client code

RBT: Auto-Active Correctness

Prove the real implementation

- verify executable SPARK (Ada) code — functional correctness + RBT invariants, not just an abstract model

Annotate, don't hand-prove

- invariants become contracts, loop invariants and ghost code; automatic provers discharge the goals

Thm: code is run-time-error-free and meets its spec

- GNATprove checks each function modularly against its contract

Best tool: auto-active verifier

- SPARK / GNATprove, Dafny, Why3 — SMT automation guided by user annotations

Ref: Dross & Moy, Auto-Active Proof of Red-Black Trees in SPARK (AdaCore)

UNIVERSITY OF WATERLOO

Problem: code is not in ADA + short on time

RBT: Software Model Checking

certora

Check the real implementation

- the Certora Prover verifies actual red-black tree code — no separate abstract model

Explore all states, not test cases

- compile the code to logic, then SMT solvers search every path for a property violation

Spec: conformance to reference implementation

- `remove_fix`, `insert_fix`, `swap`, …

Best tool: software model checker / prover

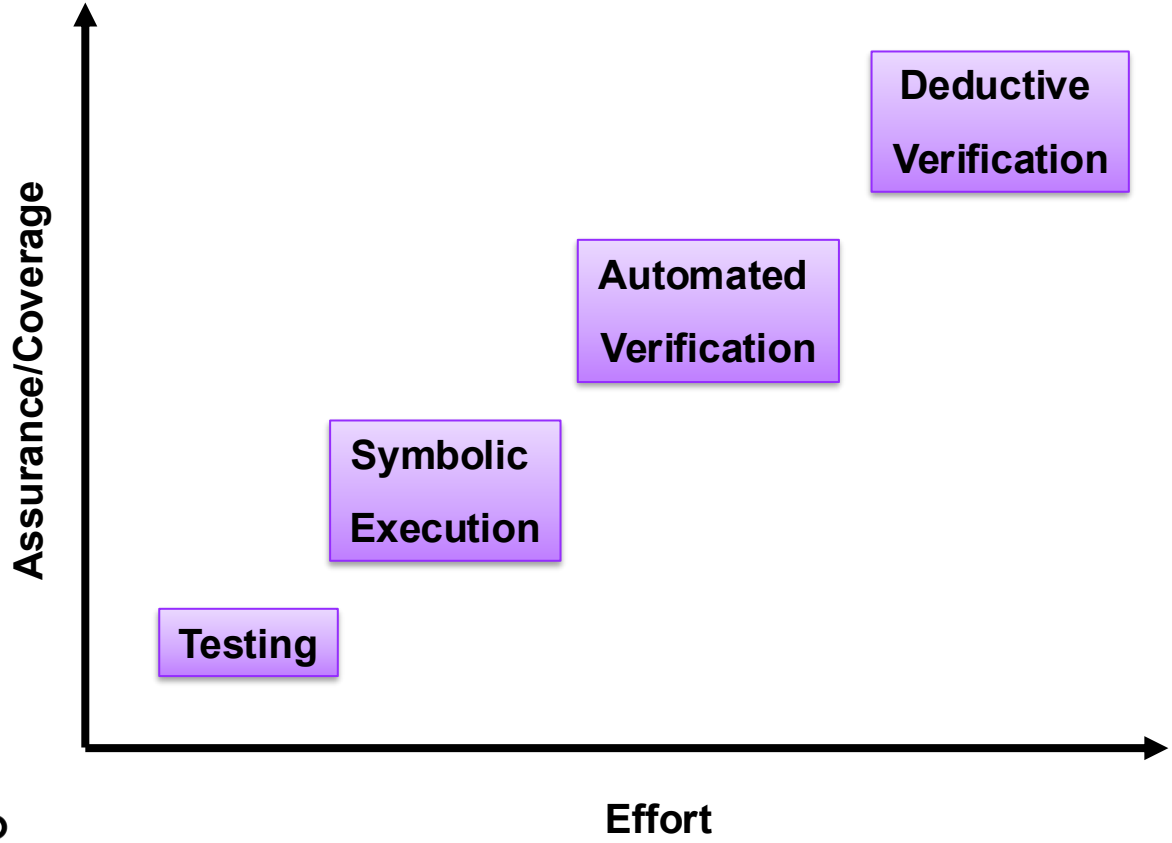
- Certora Prover, SeaHorn, CBMC — automated and counterexample-driven

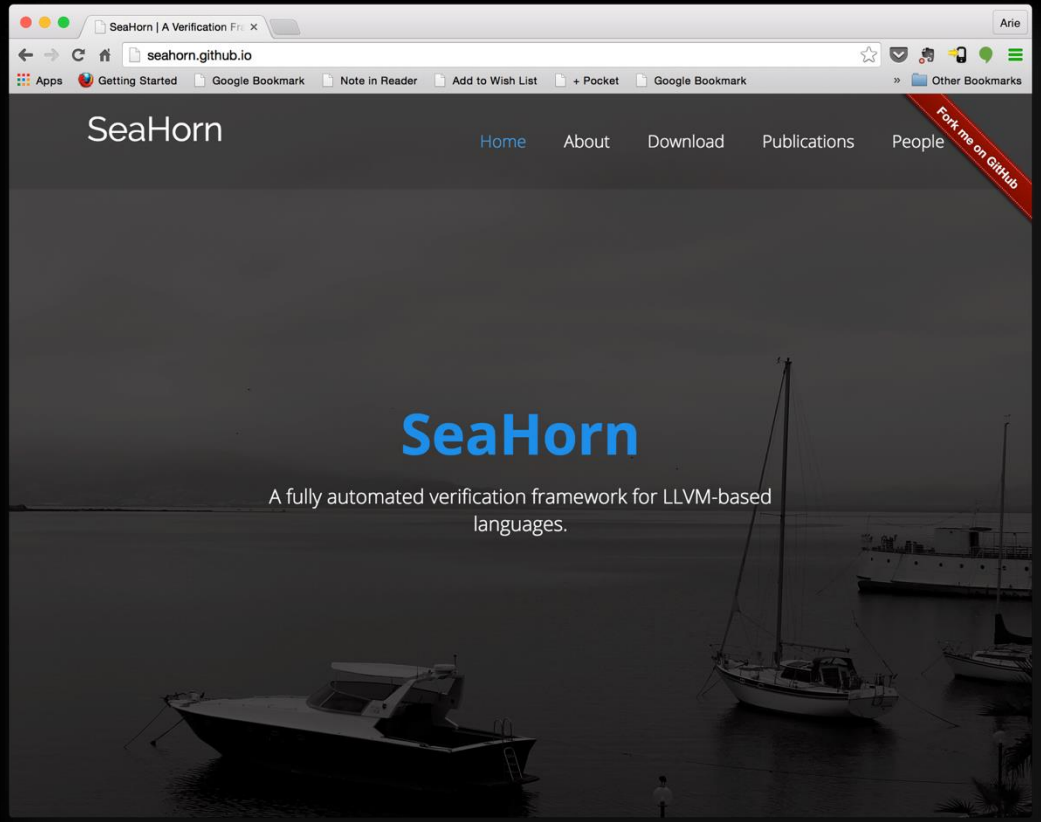
Ref: Certora, Manifest Smart-Contract Audit & FV Report (Nov 2024), p.57

Bug found (p.26): **removal mis-colors a node → an unbalanced (but valid) tree — fuzzing missed it, the prover caught it**

UNIVERSITY OF WATERLOO

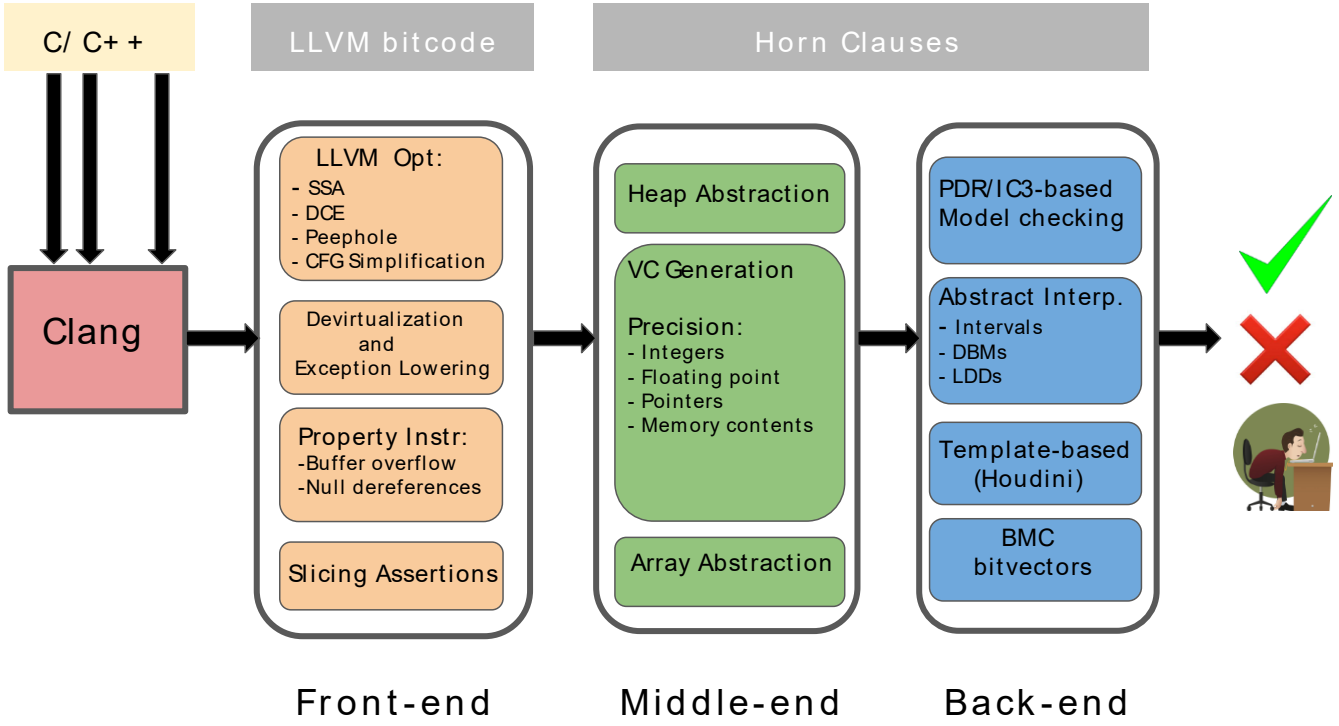
(User) Effort vs (Verification) Assurance





http://seahorn.github.io

Architecture of Seahorn



Bounded Model Checking (BMC)

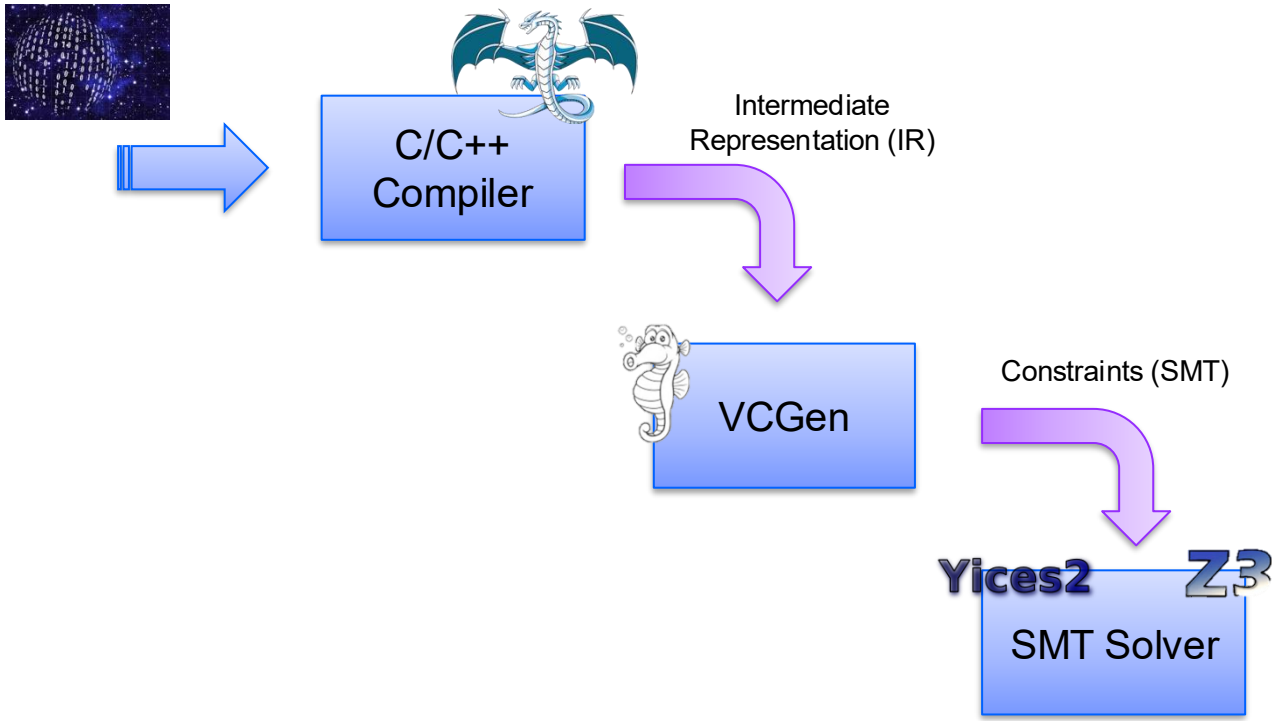
BMC: is a precise static analysis (i.e., verification) technique
- reduce verification to constraint solving with SAT- and SMT-solvers

Pros
- precision, including path sensitivity, machine arithmetic, bit-vector operations, etc.
- ease of use – everything can be modeled in code

Cons
- scalability (scales to thousands LOC, but not millions)
- requires “unit proofs” and ”mocks” to be effective

Well suited for security properties
- spatial memory safety, information flow, side-channels

Backend: Verification Condition Generation



SeaBMC: BMC for LLVM

SeaHorn-based Open-sourced BMC engine for LLVM

- bit-precise, byte-precise, path-sensitive

Supports many different encodings of verification conditions

- different encodings are better for different SMT solvers
- different encodings are better for different properties

Supports verification-specific extension to computer architecture

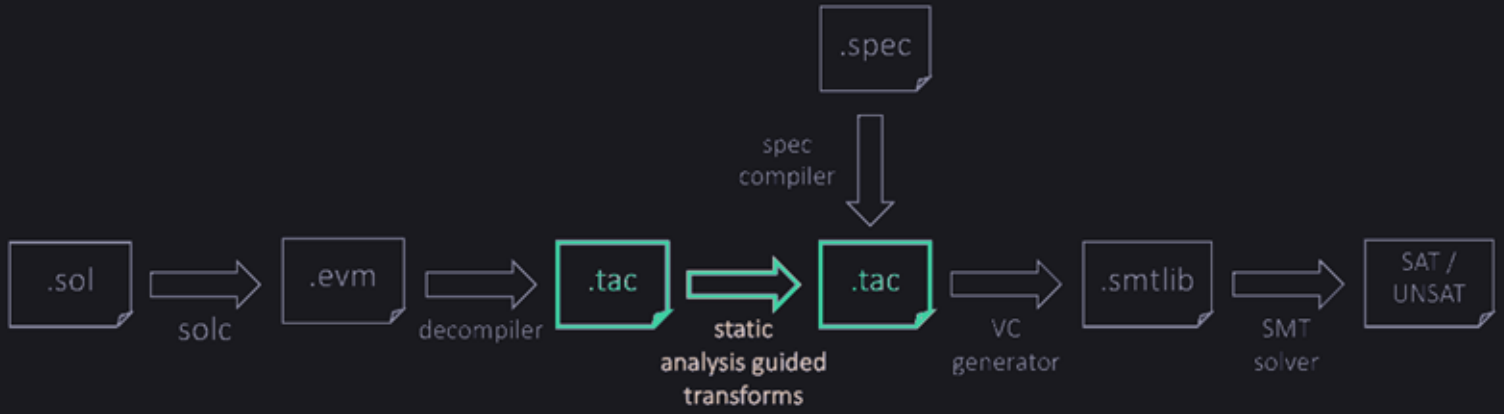
- store pointer-specific information directly with a pointer (i.e., fat-pointer)
- store memory object specific information directly with the memory object (i.e., shadow memory)
- extensions are done at the semantic level and exposed to developer via simple API

The Certora Prover Architecture

From source to proof

certora

The EVM Pipeline (OOPSLA ’24, Fig. 2)



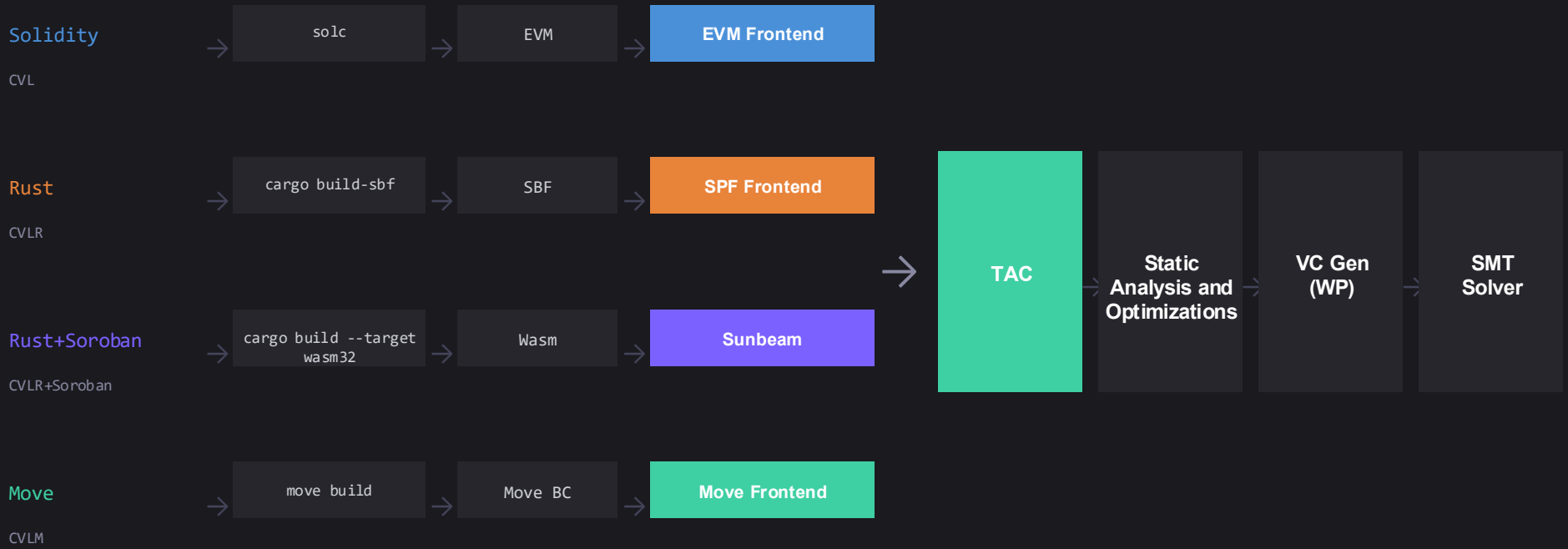
This is the EVM pipeline. But what about Solana, Stellar, Sui?

Key insight: TAC is language-agnostic. We only need new **frontends** to compile each bytecode format down to the same IR.

Everything after TAC (static analysis, VC generation, SMT solving) is shared.

certora

Four Frontends, One Backend



CVLR specs compile with the Rust code. CVLM specs are Move packages. CVL has its own compiler.

certora

Three Specification Models

CVL (EVM)

```
// Separate .spec file
rule deposit_ok {
  env e; uint256 amt;
  deposit(e, amt);
  assert ...;
}
```

Own compiler. Spec + bytecode compiled separately, linked at TAC.

CVLR (Solana / Soroban)

```
// Embedded in Rust source
#[cfg(feature = "certora")]
#[rule]
pub fn deposit_ok() {
  let x = nondet();
  cvlr_assert!(x > 0);
}
```

Specs are Rust. Compiled together via cargo build-sbf (or wasm32). Single binary.

CVLM (Sui / Aptos)

```
// Separate Move package
fun cvlm_manifest() {
  rule(b"deposit_ok");
}
fun deposit_ok(...) {
  cvlm_assert(cond);
}
```

Specs are Move packages. Rules declared via manifest. Compiled by Move compiler.

certora

From Solidity to TAC: A Concrete Example

```
// Solidity (slot 0)
mapping(address => uint256)
  balances;

function deposit(uint256 amt)
  external {
  require(amt > 0);
  balances[msg.sender]
    += amt;
}
```


```
// EVM bytecode (stack machine)
PUSH1 0x04  CALLDATALOAD
DUP1  ISZERO
PUSH2 revert  JUMPI
CALLER
PUSH1 0x00
MSTORE  // mem[0]=sender
PUSH1 0x20  MSTORE // mem[32]=0
PUSH1 0x40  PUSH1 0x00
KECCAK256 // hash 64 bytes
DUP1  SLOAD  // old balance
DUP1  DUP4  ADD  // old+amt
DUP1  DUP3  LT  // overflow?
PUSH2 revert  JUMPI
SWAP2  POP  SSTORE
```


```
// TAC (register-based)
s0 := CALLDATALOAD(0x04)
s1 := ISZERO(s0)
JUMPI s1 revert_bb

s2 := CALLER
MSTORE tacM[0x00] := s2
MSTORE tacM[0x20] := 0x00
s5 := SHA3(tacM, 0x00, 0x40)

s6 := SLOAD sM[s5]
s7 := ADD(s6, s0)
s8 := LT(s7, s6)
JUMPI s8 revert_bb
SSTORE sM[s5] := s7
```

Key: mapping slot = keccak256(sender, 0) requires writing to memory (tacM) then hashing.

Even a simple += touches tacM 3 times — this is why memory splitting matters.

certora

TAC: The Shared Intermediate Representation

```
// TAC Syntax (OOPSLA '24)
Variables    x ∈ V
Immediates   i ::= c | x
Expressions  e ::= i | x[i] | x[i:=i]

Basic        I_b ::= x ← e
                   | BRANCH x l l
Memory       I_m ::= x ← MLOAD x
                   | MSTORE x i
Storage      I_s ::= x ← SLOAD x
                   | SSTORE x i
```

Why TAC?

Register-based
No stack machine — explicit variable names for all operands

Unstructured control flow
Basic blocks + branches. No for/while — loops are explicit edges

Types
Bit256, ByteMap, WordMap, CVLArray, GhostMap, Structs

Shared backend
Static analysis and optimizations, VC generation, and SMT solving are language-agnostic once in TAC

All four frontends (EVM, Solana/SPF, Wasm/Soroban, Move) compile to TAC → one prover backend.

certora

SOTA Competition

CBMC – Bounded Model Checker for C

- started at CMU and Oxford, now supported by diffblue
- oldest, mature, actively used in industry (Amazon)
- custom C parser, some semantic particularities

Kani

- CBMC backend, Rustc frontend

KLEE – Symbolic Execution for LLVM

- mature, actively used in academic community
- de-facto symbolic execution engine in LLVM
- unlike BMC, targets bug finding rather than verification

Symbiotic

- combines KLEE with slicing for scalability
- winner of multiple SV-COMP competitions

See SV-COMP for many other tools!

Five Common Misconceptions of Bounded Model Checking

1.  Bounded MC ≠ Bounded Proof
2.  Symbolic Testing ≠ Formal Verification
3.  Verified ≠ Correct
4.  Useful ≠ Sound
5.  Automatic ≠ Slow

1 BMC ≠ Bounded Proof

"Bounded" means that the code that is being analyzed automatically is bounded

- but all proofs are bounded – they have to be written down
- this does not mean that properties that are established are bounded

Checking that a candidate formula is an inductive invariant is a BMC problem!

Checking equivalence to a reference implementation is a BMC problem

Our example of RBT is a bounded proof of unbounded equivalence

Proofs can be protected by validating that all loops are either summarized or unrolled upto maximum bound

2 Symbolic Testing ≠ Formal Verification

In BMC, specifications (unit proofs) look very similar to unit tests

- setup pre-state; execute code; assert post-condition

However, not all such symbolic tests are formal verification

- must establish some top-level property of interest, not just validate computation
- must cover all execution paths (i.e., setup must be sufficiently non-deterministic)
- must be efficiently symbolically executable (i.e., no loops, no complex data structures, …)
- AI agents often confuse this and write symbolic tests claiming that it is the only thing that is possible

3 Verified ≠ Correct

Formal Verification checks that Code satisfies Spec

- verification result is only as good as the specification
- good specification is very hard to get right
  - shares-to-assets ratio should be non-increasing over time
  - `assert (shares / assets <= shares' / assets')`
  - initial `shares == assets == 0`

From user-perspective specification is simple

- my system should be safe
- no assets should be lost (but a little rounding error is ok…)

Formalizing these into verifiable properties is very hard

- vacuity, coverage, environment model, …

4 Useful ≠ Sound

Formal Verification tools must be sound

- an unsound tool cannot be trusted
- (unless it finds a bug that is reproducible)

A sound tool that never returns is not useful

Scalable effective verification relies on assumptions

- simplified model of the environment or attacker
- mocking (not modeling) difficult parts of the environment
  - e.g., order book with 3 orders, only allows limited operations, in limited order

5 Scalability Requires (Human) Insight

Algorithmic Automation of Hard problems cannot be scalable

- some problems are hard – there are no polytime solutions
- SAT/SMT solvers are fantastic, sometimes scalable, but often unpredictable

Corollary: do not expect to magically verify a sizeable system by pointing BMC to its entry point, or by simulating number of complex steps

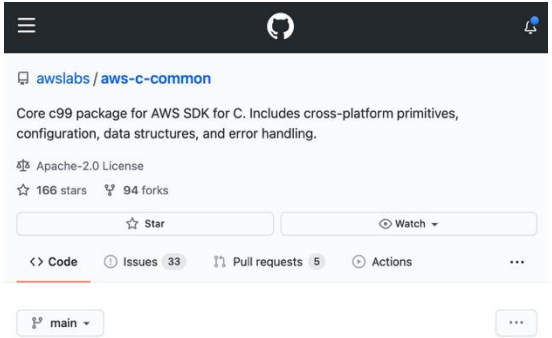
Determine top-level specification, break it into smaller sub-properties, keep at it until have properties that fit scalability of your automated tool

- replace ”forall” properties by individuals – i.e., true for some unspecified instance
- model concurrent environment by interference – i.e., some variables just change

Verifying aws-c-common library with SeaBMC

CASE STUDY

Case Study: aws-c-common library



Core C99 package for AWS SDK
- cross-platform primitives
- configuration
- data structures
- error handling

Self-contained

Low-level and platform specific C

Extensively verified using CBMC
- >160 unit proofs
- verify memory safety, representation invariants, basic operations

https://github.com/awslabs/aws-c-common

Code as Spec (CaS): A Unit Proof

```
1.  int main() {
2.  /* data structure */
3.  struct aws_array_list list;
4.  initialize_bounded_array_list(&list);
5.  /* assumptions */
6.  assume(aws_array_list_is_valid(&list));
7.  assume(list.item_size > 0);
8.  ...
9.  /* perform operation under verification */
10. size_t capacity = aws_array_list_capacity(&list);
11. /* assertions */
12. assert(aws_array_list_is_valid(&list));
13. assert(capacity == list.current_size /
    list.item_size);
14. ...
15. return 0;
16. }
```

initialization

pre-condition

function to be verified

post-condition

https://github.com/seahorn/verify-c-common/blob/master/seahorn/jobs/array_list_capacity/aws_array_list_capacity_harness.c

Code-as-Spec (CaS) features

Use code to write pre-and-post conditions
- empower developers to write and maintain specifications
- share specifications between multiple tools and techniques
- structure verification effort around unit proofs

A unit proof (like unit test)
- sets the environment for verification (pre-condition)
- calls function under verification
- validates the result (post-condition)

Extend programming language with specification primitives
- non-deterministic (i.e., symbolic) input
- `verifier.assume()` built-in to specify desired pre-condition
- `verifier.assert()` built-in to specify statically checked assertions

https://github.com/seahorn/verify-c-common

Research Questions

RQ1: Does CaS empower multiple (analysis) tools?

- **Yes** – we use BMC, Symbolic Execution, and Fuzzing all at once
- **But** – semantics, semantics, semantics
- **And** – need to design specs with multiple tools in mind

RQ2: Are there bugs in verified code?

- **Yes (and No)**
  - found serious bugs in specifications (but they did not hide bugs in code)
  - found (potential) bugs that might manifest in the future

RQ3: Can CaS specs be improved?

- **Yes**
  - writing effective specifications is challenging, no matter what the language
  - need built-ins specific for verification to directly express required concepts
    - size of referenced memory, modifiability of memory region, etc.

RQ1: Semantics is important!

CBMC

```
void list_get_at_ptr_harness() {
  struct List l;

  assume(list_is_bounded(&l));
  ensure_list_has_allocated_data_member(&l);
  void **val = can_fail_malloc(sizeof(void *));
  size_t index;
  assume(list_is_valid(&l) && val != NULL);
  if (list_get_at_ptr(&l, val, index) == SUCCESS)
    assert(l.data != NULL && index < l.length);
  assert(list_is_valid(&l));
}
```

Undefined behavior (uninitialized variables) are resolved based on the internal semantics of CBMC

UNIVERSITY OF WATERLOO

CBMC

RQ1: Semantics is important!

SEAHORN

```
void list_get_at_ptr_harness() {
  struct List l;
  memhavoc(&l, sizeof(struct List));
  assume(list_is_bounded(&l));
  ensure_list_has_allocated_data_member(&l);
  void **val = can_fail_malloc(sizeof(void *));
  size_t index = nd_size_t();
  assume(list_is_valid(&l) && val != NULL);
  if (list_get_at_ptr(&l, val, index) == SUCCESS)
    assert(l.data != NULL && index < l.length);
  assert(list_is_valid(&l));
}
```

No undefined behavior. Semantics depend on implementation of the explicit initialization function

RQ1: Does CaS empower multiple (analysis) tools?

Different tools require somewhat different specification of assumptions

- refactor to have different implementation of specs for each style of tools

SEAHORN

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

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

libFuzzer

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

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

UNIVERSITY OF WATERLOO

RQ2: Are there bugs in verified code?

Found bugs in representation invariant

- representation invariant defines basic properties of a data structure
- assumed to be true before any function under verification
- checked that it is maintained at every call

Bug was subtle enough to be preserved by each function

- could hide real bugs in real code (but did not in this case)

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

UNIVERSITY OF WATERLOO

RQ2: Are there bugs in verified code?

Found bugs in representation invariant

- representation invariant defines basic properties of a data structure
- assumed to be true before any function under verification
- checked that it is maintained at every call

Bug was subtle enough to be preserved by each function

- could hide real bugs in real code (but did not in this case)

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

UNIVERSITY OF WATERLOO

Vacuity in CaS

Vacuity is a known sanity check in temporal model checking
- also known as *antecedent failure*
- a property is satisfied *vacuously* if
  - it is true in the model
  - a much stronger property is true
- e.g., **always if p then q** is true, but also **always not p** is true

In CaS, properties are not specified in a specialized language
- properties are embedded in code, they are part of code
- What is vacuity in this case?

Our definition: **sassert** is satisfied vacuously iff it is never reachable
- e.g., `if (c) { sassert(0); }`

RQ3: Can CaS specs be improved?

Writing specifications is no different than writing code

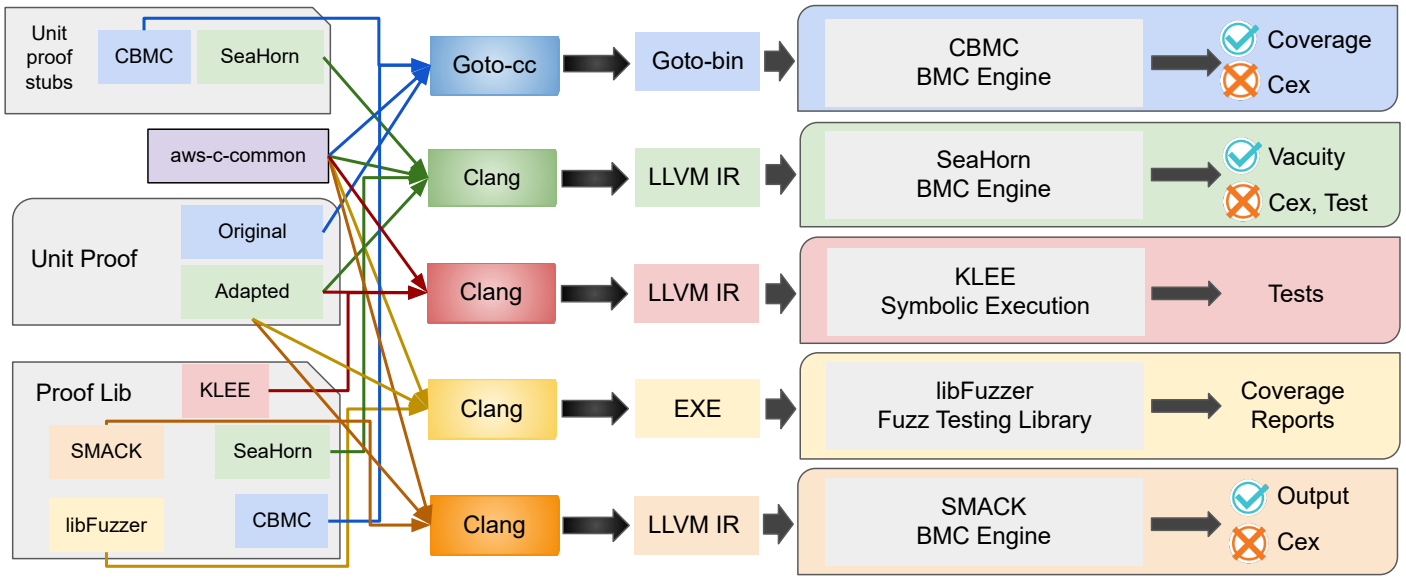
- there is good code, there is ok code, there is bad code
- sometimes, there is better code
- e.g., we improve specification of linked list data structure to verify faster while checking stronger properties

However, need additional built-in functions to communicate intention in the specifications

- `is_deref(p, sz)` – true if pointer p points to at least sz accessible bytes
- `is_mod(p)` – true if object pointed to by pointer p has been modified recently
- `is_alloc(p)` – true if object pointed to be p is (still) allocated
- …

Reusing CaS between tools requires standard for built-ins!

Case Study Architecture

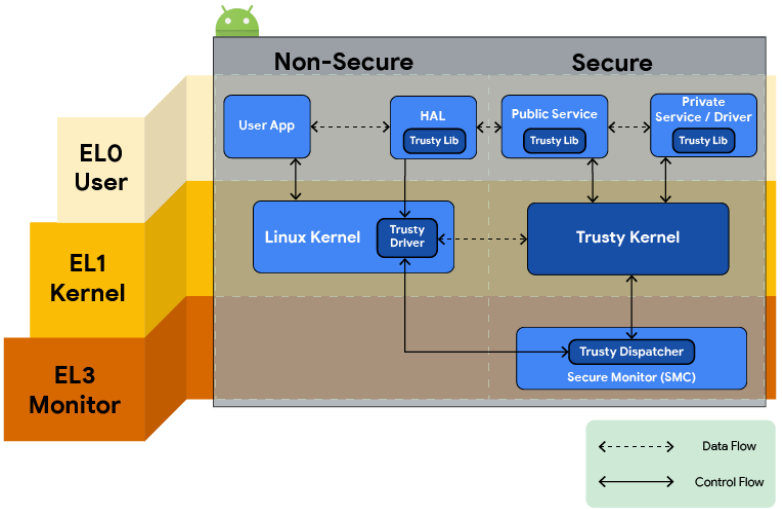


https://github.com/seahorn/verify-c-common

UNIVERSITY OF WATERLOO

Verifying Applications in Trusty TEE

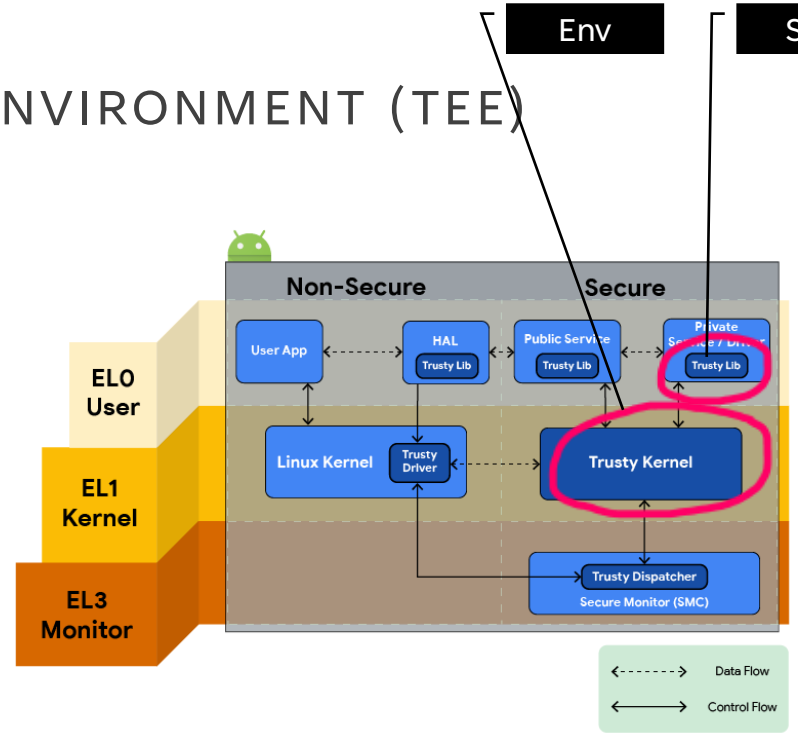
CASE STUDY



TRUSTY TRUSTED EXECUTION ENVIRONMENT (TEE)

TEE enables secure, trusted, and verified computation on an untrusted user-controlled mobile device

- Secure storage, cryptographic key management, payment services, etc.



Trusty architecture

https://source.android.com/docs/security/features/trusty

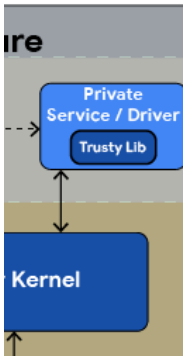
TRUSTLET VERIFICATION EXAMPLE

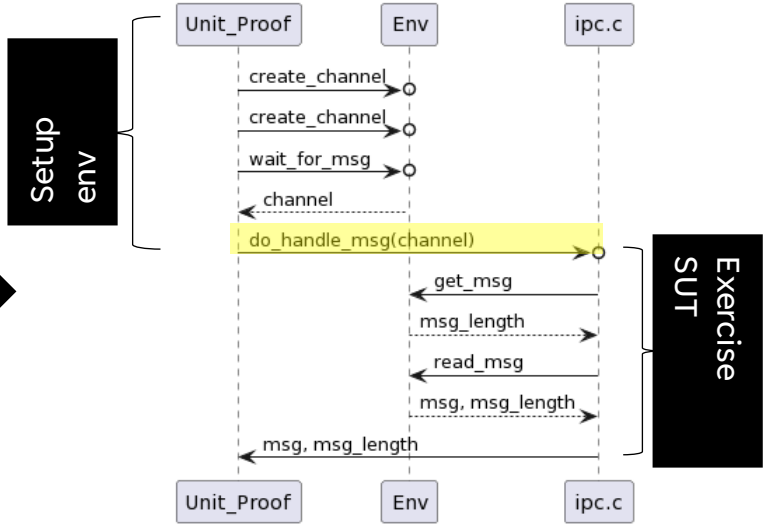
`ipc.c::do_handle_msg(channel)` is a middle layer (SUT)

It handles message transfer between kernel and trustlet (application)

Properties to verify in ipc.c

- No out of bounds buffer access
- Message buffer is not modified
- Message is not surreptitiously copied





TRUSTY LIB AND KERNEL INTERACTION

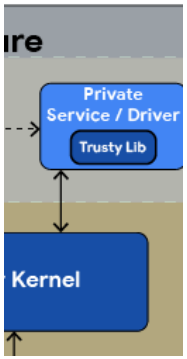
TRUSTLET VERIFICATION EXAMPLE

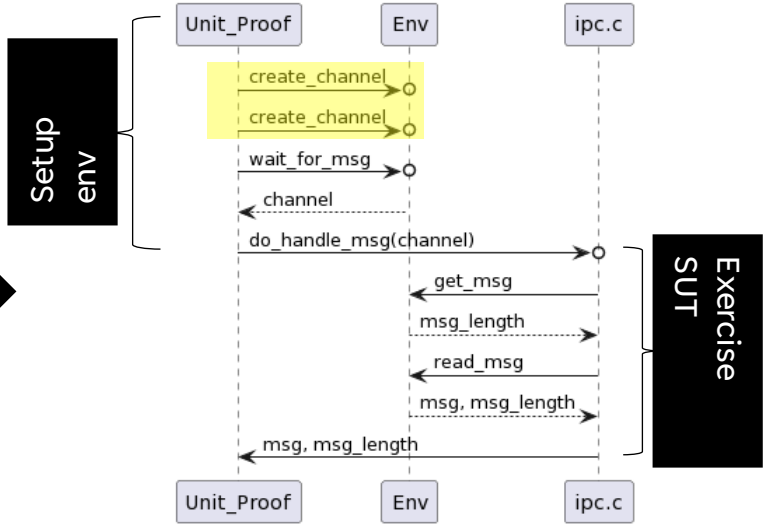
`ipc.c::do_handle_msg(channel)` is a middle layer (SUT)

It handles message transfer between kernel and trustlet (application)

Properties to verify in ipc.c

- No out of bounds buffer access
- Message buffer is not modified
- Message is not surreptitiously copied





TRUSTY LIB AND KERNEL INTERACTION

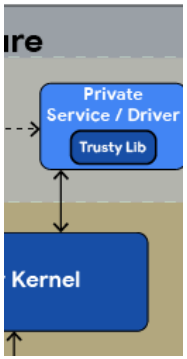
TRUSTLET VERIFICATION EXAMPLE

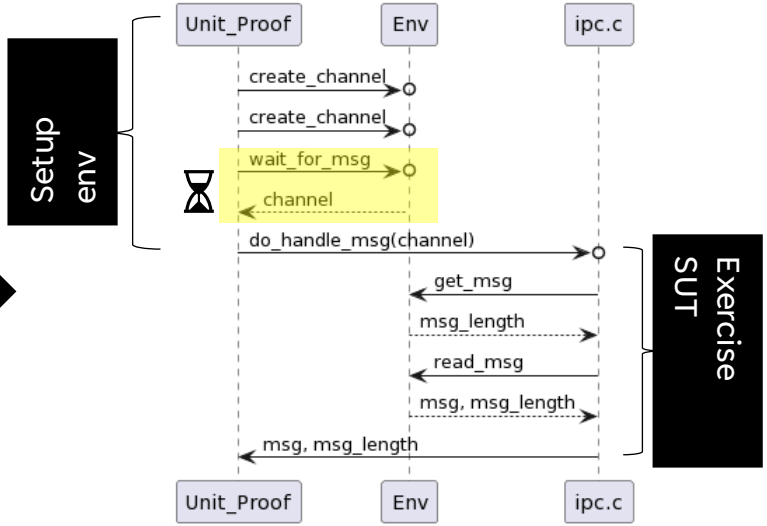
`ipc.c::do_handle_msg(channel)` is a middle layer (SUT)

It handles message transfer between kernel and trustlet (application)

Properties to verify in ipc.c

- No out of bounds buffer access
- Message buffer is not modified
- Message is not surreptitiously copied





TRUSTY LIB AND KERNEL INTERACTION

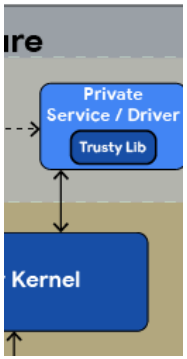
TRUSTLET VERIFICATION EXAMPLE

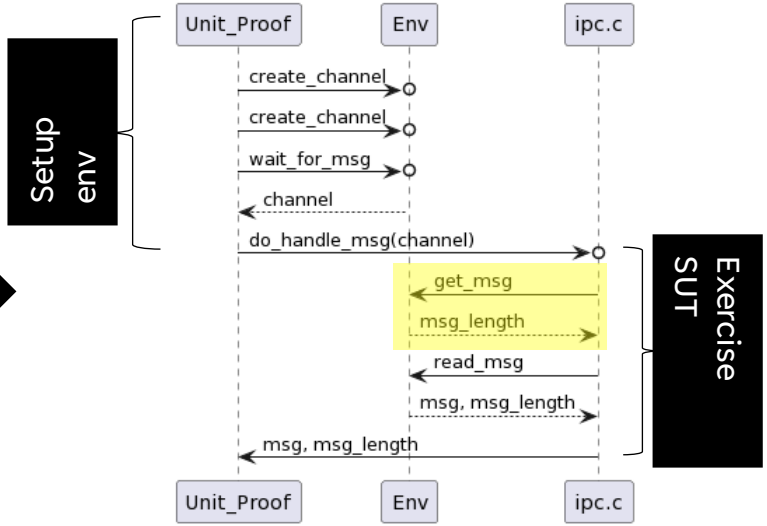
`ipc.c::do_handle_msg(channel)` is a middle layer (SUT)

It handles message transfer between kernel and trustlet (application)

Properties to verify in ipc.c

- No out of bounds buffer access
- Message buffer is not modified
- Message is not surreptitiously copied





TRUSTY LIB AND KERNEL INTERACTION

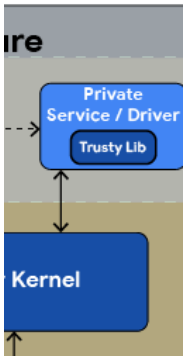
TRUSTLET VERIFICATION EXAMPLE

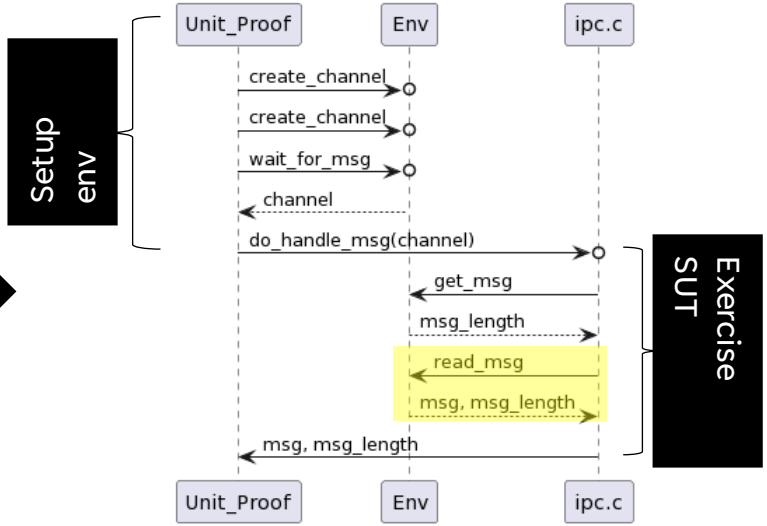
`ipc.c::do_handle_msg(channel)` is a middle layer (SUT)

It handles message transfer between kernel and trustlet (application)

Properties to verify in ipc.c

- No out of bounds buffer access
- Message buffer is not modified
- Message is not surreptitiously copied





TRUSTY LIB AND KERNEL INTERACTION

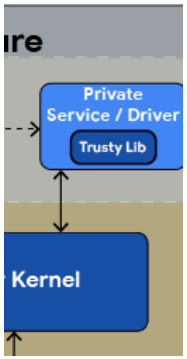
TRUSTLET VERIFICATION EXAMPLE

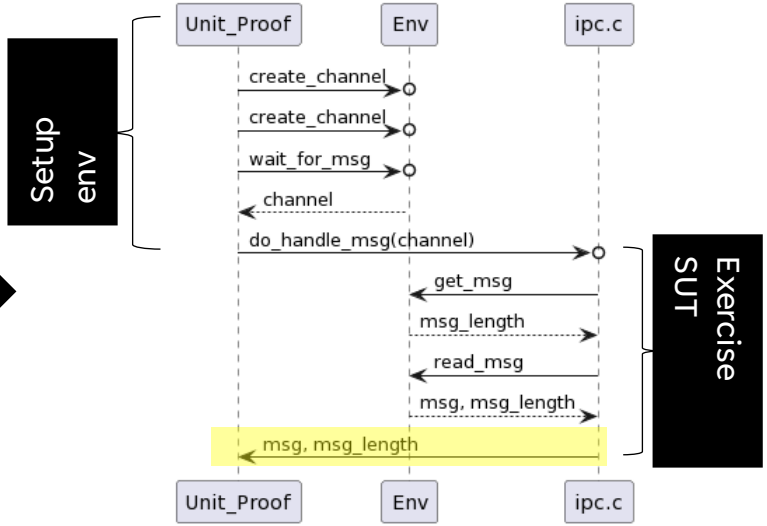
`ipc.c::do_handle_msg(channel)` is a middle layer (SUT)

It handles message transfer between kernel and trustlet (application)

Properties to verify in ipc.c

- No out of bounds buffer access
- Message buffer is not modified
- Message is not surreptitiously copied





TRUSTY LIB AND KERNEL INTERACTION

SYSTEM UNDER TEST

```
1.  #define MAX_SIZE 4096
2.  int do_handle_msg(msg_handler_t msg_handler, int chan) {
3.    char *msg = (char *)malloc(MAX_SIZE);
4.    size_t msg_len;
5.    get_msg(chan, &msg_len);
6.    int rc = read_msg(chan, msg);
7.    if (((size_t)rc) < msg_len) {
8.      free(msg);
9.      return rc;
10.   }
11.   rc = msg_handler(msg, msg_len);
12.   free(msg);
13.   return rc;
14. }
```

System Under Test (SUT)

The `do_handle_msg` function

1. Extracts the message from the kernel
2. Calls the application handler, passing in the message

SYSTEM UNDER TEST

```
1.  #define MAX_SIZE 4096
2.  int do_handle_msg(msg_handler_t msg_handler, int chan) {
3.    char *msg = (char *)malloc(MAX_SIZE);
4.    size_t msg_len;
5.    get_msg(chan, &msg_len);
6.    int rc = read_msg(chan, msg);
7.    if (((size_t)rc) < msg_len) {
8.      free(msg);
9.      return rc;
10.   }
11.   rc = msg_handler(msg, msg_len);
12.   free(msg);
13.   return rc;
14. }
```

System Under Test (SUT)

The do_handle_msg function

1. Extracts the message from the kernel
2. Calls the application handler, passing in the message

Kernel ← → trustlet channel

Application handler

SYSTEM UNDER TEST

```
1.  #define MAX_SIZE 4096
2.  int do_handle_msg(msg_handler_t msg_handler, int chan) {
3.    char *msg = (char *)malloc(MAX_SIZE);
4.    size_t msg_len;
5.    get_msg(chan, &msg_len);
6.    int rc = read_msg(chan, msg);
7.    if (((size_t)rc) < msg_len) {
8.      free(msg);
9.      return rc;
10.   }
11.   rc = msg_handler(msg, msg_len);
12.   free(msg);
13.   return rc;
14. }
```

Prepare to receive message from kernel

The do_handle_msg function

1. Extracts the message from the kernel
2. Calls the application handler, passing in the message

System Under Test (SUT)

SYSTEM UNDER TEST

```
1.  #define MAX_SIZE 4096
2.  int do_handle_msg(msg_handler_t msg_handler, int chan) {
3.    char *msg = (char *)malloc(MAX_SIZE);
4.    size_t msg_len;
5.    get_msg(chan, &msg_len);
6.    int rc = read_msg(chan, msg);
7.    if (((size_t)rc) < msg_len) {
8.      free(msg);
9.      return rc;
10.   }
11.   rc = msg_handler(msg, msg_len);
12.   free(msg);
13.   return rc;
14. }
```

Receive message

The do_handle_msg function

1. Extracts the message from the kernel
2. Calls the application handler, passing in the message

System Under Test
(SUT)

SYSTEM UNDER TEST

```
1.  #define MAX_SIZE 4096
2.  int do_handle_msg(msg_handler_t msg_handler, int chan) {
3.    char *msg = (char *)malloc(MAX_SIZE);
4.    size_t msg_len;
5.    get_msg(chan, &msg_len);
6.    int rc = read_msg(chan, msg);
7.    if (((size_t)rc) < msg_len) {
8.      free(msg);
9.      return rc;
10.   }
11.   rc = msg_handler(msg, msg_len);
12.   free(msg);
13.   return rc;
14. }
```

System Under Test (SUT)

The do_handle_msg function

1. Extracts the message from the kernel
2. Calls the application handler, passing in the message

Invariant: Length returned by read_msg <= length returned by get_msg

SYSTEM UNDER TEST

```
1.  #define MAX_SIZE 4096
2.  int do_handle_msg(msg_handler_t msg_handler, int chan) {
3.    char *msg = (char *)malloc(MAX_SIZE);
4.    size_t msg_len;
5.    get_msg(chan, &msg_len);
6.    int rc = read_msg(chan, msg);
7.    if (((size_t)rc) < msg_len) {
8.      free(msg);
9.      return rc;
10.   }
11.   rc = msg_handler(msg, msg_len);
12.   free(msg);
13.   return rc;
14. }
```

Pass message to app

The do_handle_msg function

1. Extracts the message from the kernel
2. Calls the application handler, passing in the message

System Under Test
(SUT)

UNIT PROOF

```
1.  static int test_msg_handler(char *msg, size t msg size) {
2.    sassert(!sea_is_modified((char *)msg));
3.    return nd int();
4.  }
5.  int main(void) {
6.    create_channel();
7.    create_channel();
8.    int chan = wait_for_msg();
9.    do_handle_msg(test_msg_handler, chan);
10.   return 0;
11. }
```

Seahorn unit proof

Unit proof

UNIT PROOF

```
1.  static int test_msg_handler(char *msg, size t msg size) {
2.    sassert(!sea_is_modified((char *)msg));
3.    return nd int();
4.  }
5.  int main(void) {
6.    create_channel();
7.    create_channel();
8.    int chan = wait_for_msg();
9.    do_handle_msg(test_msg_handler, chan);
10.   return 0;
11. }
```

Seahorn unit proof

Create channels in the kernel

UNIT PROOF

```
1.  static int test_msg_handler(char *msg, size t msg size) {
2.    sassert(!sea_is_modified((char *)msg));
3.    return nd int();
4.  }
5.  int main(void) {
6.    create_channel();
7.    create_channel();
8.    int chan = wait_for_msg();
9.    do_handle_msg(test_msg_handler, chan);
10.   return 0;
11. }
```

Wait for message from kernel

Seahorn unit proof

UNIT PROOF

```
1.  static int test_msg_handler(char *msg, size t msg size) {
2.    sassert(!sea_is_modified((char *)msg));
3.    return nd int();
4.  }
5.  int main(void) {
6.    create_channel();
7.    create_channel();
8.    int chan = wait_for_msg();
9.    do_handle_msg(test_msg_handler, chan);
10.   return 0;
11. }
```

Call SUT

Seahorn unit proof

UNIT PROOF

```
1.  static int test_msg_handler(char *msg, size t msg size) {
2.    sassert(!sea_is_modified((char *)msg));
3.    return nd int();
4.  }
5.  int main(void) {
6.    create_channel();
7.    create_channel();
8.    int chan = wait_for_msg();
9.    do_handle_msg(test_msg_handler, chan);
10.   return 0;
11. }
```

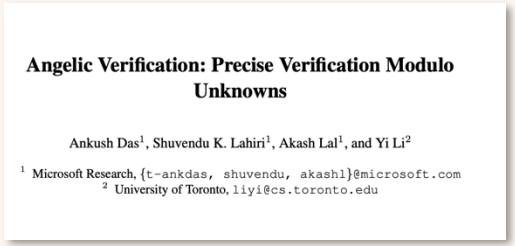
Seahorn unit proof

Check message is unmodified between kernel and application layer

HOW TO MODEL THE TRUSTY KERNEL (ENVIRONMENT)?

System under Test (SUT) requires an environment (env)
Environment can be modelled in various ways

Getting the env model right is trial-and-error

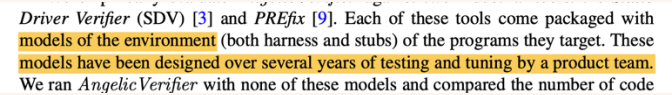


Angelic Verification: Precise Verification Modulo Unknowns

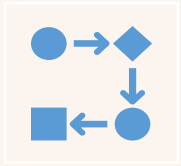
Ankush Das[1], Shuvendu K. Lahiri[1], Akash Lal[1], and Yi Li[2]

[1] Microsoft Research, {t-ankdas, shuvendu, akashl}@microsoft.com
[2] University of Toronto, liyi@cs.toronto.edu

CAV 2015



Driver Verifier (SDV) [3] and *PREfix* [9]. Each of these tools come packaged with models of the environment (both harness and stubs) of the programs they target. These models have been designed over several years of testing and tuning by a product team. We ran *Angelic Verifier* with none of these models and compared the number of code



Two ways of specifying verification environment

Stateful env using **fakes**

Stateless env using Function **summaries**

FAKE ENVIRONMENT FOR VERIFICATION

```
1.  static int g_next;
2.  typedef struct msg {
3.    char *buf;
4.    size_t len; } MSG;
5.  static MSG msgs[10];
6.  int create_channel() {
7.    size_t len = nd_size_t();
8.    char *msg = (char *)malloc(len);
9.    memhavoc(msg, len);
10.   int chan = g_next++;
11.   msgs[chan].buf = msg;
12.   msgs[chan].len = len;
13.   return chan; }
```

```
int wait_for_msg() {
  int channel = nd_int();
  assume(channel > 0 && channel < g_next);
  return channel; }

void get_msg(int chan, size t *len) {
  *len = msgs[chan].len; }

int read_msg(int chan, char *msg) {
  size_t len = nd_size_t();
  assume(len <= msgs[chan].len);
  memcpy(msg, msgs[chan].buf, msgs[chan].len);
  sea_reset_modified(msg);
  return nd_bool() ? -1 : len; }
```

A fake environment pretends to be the real thing!

FAKE ENVIRONMENT FOR VERIFICATION

```
1.  static int g_next;
2.  typedef struct msg {
3.    char *buf;
4.    size_t len; } MSG;
5.  static MSG msgs[10];
6.  int create_channel() {
7.    size_t len = nd_size_t();
8.    char *msg = (char *)malloc(len);
9.    memhavoc(msg, len);
10.   int chan = g_next++;
11.   msgs[chan].buf = msg;
12.   msgs[chan].len = len;
13.   return chan; }
```

```
int wait_for_msg() {
  int channel = nd_int();
  assume(channel > 0 && channel < g_next);
  return channel; }

void get_msg(int chan, size t *len) {
  *len = msgs[chan].len; }

int read_msg(int chan, char *msg) {
  size_t len = nd_size_t();
  assume(len <= msgs[chan].len);
  memcpy(msg, msgs[chan].buf, msgs[chan].len);
  sea_reset_modified(msg);
  return nd_bool() ? -1 : len; }
```

Environment has a queue of messages

FAKE ENVIRONMENT FOR VERIFICATION

```
1.  static int g_next;
2.  typedef struct msg {
3.    char *buf;
4.    size_t len; } MSG;
5.  static MSG msgs[10];
6.  int create_channel() {
7.    size_t len = nd_size_t();
8.    char *msg = (char *)malloc(len);
9.    memhavoc(msg, len);
10.   int chan = g_next++;
11.   msgs[chan].buf = msg;
12.   msgs[chan].len = len;
13.   return chan; }
```

```
int wait_for_msg() {
  int channel = nd_int();
  assume(channel > 0 && channel < g_next);
  return channel; }

void get_msg(int chan, size t *len) {
  *len = msgs[chan].len; }

int read_msg(int chan, char *msg) {
  size_t len = nd_size_t();
  assume(len <= msgs[chan].len);
  memcpy(msg, msgs[chan].buf, msgs[chan].len);
  sea_reset_modified(msg);
  return nd_bool() ? -1 : len; }
```

The create_channel func creates a msg with nondeterministic length and content for the next avail channel

FAKE ENVIRONMENT FOR VERIFICATION

```
1.  static int g_next;
2.  typedef struct msg {
3.    char *buf;
4.    size_t len; } MSG;
5.  static MSG msgs[10];
6.  int create_channel() {
7.    size_t len = nd_size_t();
8.    char *msg = (char *)malloc(len);
9.    memhavoc(msg, len);
10.   int chan = g_next++;
11.   msgs[chan].buf = msg;
12.   msgs[chan].len = len;
13.   return chan; }
```

```
int wait_for_msg() {
  int channel = nd_int();
  assume(channel > 0 && channel < g_next);
  return channel; }

void get_msg(int chan, size t *len) {
  *len = msgs[chan].len; }

int read_msg(int chan, char *msg) {
  size_t len = nd_size_t();
  assume(len <= msgs[chan].len);
  memcpy(msg, msgs[chan].buf, msgs[chan].len);
  sea_reset_modified(msg);
  return nd_bool() ? -1 : len; }
```

The wait_for_msg func returns any valid channel

FAKE ENVIRONMENT FOR VERIFICATION

```
1.  static int g_next;
2.  typedef struct msg {
3.    char *buf;
4.    size_t len; } MSG;
5.  static MSG msgs[10];
6.  int create_channel() {
7.    size_t len = nd_size_t();
8.    char *msg = (char *)malloc(len);
9.    memhavoc(msg, len);
10.   int chan = g_next++;
11.   msgs[chan].buf = msg;
12.   msgs[chan].len = len;
13.   return chan; }
```

```
int wait_for_msg() {
  int channel = nd_int();
  assume(channel > 0 && channel < g_next);
  return channel; }

void get_msg(int chan, size t *len) {
  *len = msgs[chan].len; }

int read_msg(int chan, char *msg) {
  size_t len = nd_size_t();
  assume(len <= msgs[chan].len);
  memcpy(msg, msgs[chan].buf, msgs[chan].len);
  sea_reset_modified(msg);
  return nd_bool() ? -1 : len; }
```

The get_msg func returns the msg length at given channel

FAKE ENVIRONMENT FOR VERIFICATION

```
1.  static int g_next;
2.  typedef struct msg {
3.    char *buf;
4.    size_t len; } MSG;
5.  static MSG msgs[10];
6.  int create_channel() {
7.    size_t len = nd_size_t();
8.    char *msg = (char *)malloc(len);
9.    memhavoc(msg, len);
10.   int chan = g_next++;
11.   msgs[chan].buf = msg;
12.   msgs[chan].len = len;
13.   return chan; }
```

```
int wait_for_msg() {
  int channel = nd_int();
  assume(channel > 0 && channel < g_next);
  return channel; }

void get_msg(int chan, size t *len) {
  *len = msgs[chan].len; }

int read_msg(int chan, char *msg) {
  size_t len = nd_size_t();
  assume(len <= msgs[chan].len);
  memcpy(msg, msgs[chan].buf, msgs[chan].len);
  sea_reset_modified(msg);
  return nd_bool() ? -1 : len; }
```

The read_msg func copies the message from queue to output parameter (msg)

FAKE ENVIRONMENT FOR VERIFICATION

```
1.  static int g_next;
2.  typedef struct msg {
3.    char *buf;
4.    size_t len; } MSG;
5.  static MSG msgs[10];
6.  int create_channel() {
7.    size_t len = nd_size_t();
8.    char *msg = (char *)malloc(len);
9.    memhavoc(msg, len);
10.   int chan = g_next++;
11.   msgs[chan].buf = msg;
12.   msgs[chan].len = len;
13.   return chan; }
```

```
int wait_for_msg() {
  int channel = nd_int();
  assume(channel > 0 && channel < g_next);
  return channel; }

void get_msg(int chan, size t *len) {
  *len = msgs[chan].len; }

int read_msg(int chan, char *msg) {
  size_t len = nd_size_t();
  assume(len <= msgs[chan].len);
  memcpy(msg, msgs[chan].buf, msgs[chan].len);
  sea_reset_modified(msg);
  return nd_bool() ? -1 : len; }
```

Invariant: Length returned by read_msg <= length returned by get_msg

FAKE ENVIRONMENTS SUMMARY

DO MAINTAIN GLOBAL INVARIANTS

ARE COMPLEX TO WRITE

CAN REQUIRE VERIFICATION

MOCKS: UNIT PROOF LOCAL ENVIRONMENT

```
1.  #define MAX_SIZE 4096

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

3.  int wait_for_msg() { /* NOP */ return nd_int(); }
4.  static size_t g_msg_size;

5.  void get_msg(int chan, size_t *len) {
6.    g_msg_size = nd_size_t();
7.    *len = g_msg_size;
8.  }

9.  int read_msg(int chan, char *msg) {
10.   char buf[MAX_SIZE];
11.   memhavoc(&buf);
12.   size_t len = nd_size_t();
13.   assume(len <= g_msg_size);
14.   memcpy(msg, buf, len);
15.   sea_reset_modified(msg);
16.   return len;
17. }
```

Idea: Mock the environment to include just enough state to capture invariant expected by SUT

Pros: no array of messages

simplifications are specification-dependent

MOCKS: UNIT PROOF LOCAL ENVIRONMENT

```
1.  #define MAX_SIZE 4096

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

3.  int wait_for_msg() { /* NOP */ return nd_int(); }
4.  static size_t g_msg_size;

5.  void get_msg(int chan, size_t *len) {
6.    g_msg_size = nd_size_t();
7.    *len = g_msg_size;
8.  }

9.  int read_msg(int chan, char *msg) {
10.   char buf[MAX_SIZE];
11.   memhavoc(&buf);
12.   size_t len = nd_size_t();
13.   assume(len <= g_msg_size);
14.   memcpy(msg, buf, len);
15.   sea_reset_modified(msg);
16.   return len;
17. }
```

Idea: Mock the environment to include just enough state to capture invariant expected by SUT

Pros: no array of messages

simplifications are specification-dependent

Invariant: Length returned by read_msg <= length returned by get_msg

MOCK OBJECTS IN UNIT TESTING

Endo-Testing: Unit Testing with Mock Objects

Tim Mackinnon, Steve Freeman, Philip Craig

(tim.mackinnon@pobox.com, steve@m3p.co.uk, philip@pobox.com)

https://www2.ccs.neu.edu/research/demeter/related-work/extreme-programming/MockObjectsFinal.PDF, Circa 2001

Mock Roles, Not Objects

Steve Freeman, Tim Mackinnon, Nat Pryce, Joe Walnes
ThoughtWorks UK
Berkshire House, 168-173 High Holborn
London WC1V 7AA
{sfreeman, tmackinn, npryce, jwalnes} @thoughtworks.com

OOPSLA 2004

*"Writing mock objects is similar to writing code stubs with **two interesting differences**:*

*We test at a **finer level of granularity** than is usual, and we use our **tests** and stubs **to drive the development** of our production code."*

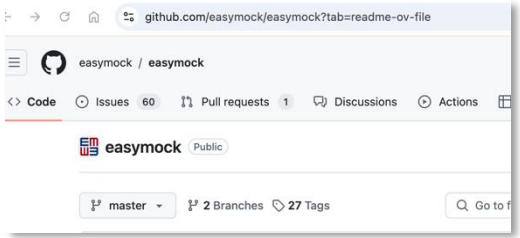
Google Testing Blog

Testing on the Toilet: Don't Mock Types You Don't Own

Thursday, July 16, 2020

This article was adapted from a Google Testing on the Toilet (ToT) episode. You can download a printer-friendly version of this ToT episode and post it in your office.

Google Testing best practices blog



Easymock framework for Java

`unittest.mock` — mock object library

Added in version 3.3.

Source code: Lib/unittest/mock.py

`unittest.mock` is a library for testing in Python. It allows you to replace parts of your system under test with mock objects and make assertions about how they have been used.

Python includes a mocking framework

MOCKS IN VERIFICATION WORKFLOW

- Mocks allow for rapid verification and prototyping
  - specify only scenarios of how environment behaves
- Mocks change the soundness criteria
  - verification is against the mock and not real environment
  - the mock might be restrictive and limit verification to slice of behaviors
  - however, this is not necessary – mocks can be sufficient as a complete verification
- Mocks for Verification require new style of Mocking Framework
  - static vs dynamic dispatch
  - non-determinism to model many related scenarios at once
  - must play well with symbolic verification

Siddharth Priya, Temesghen Kahsai, Arie Gurfinkel:
Unlocking the Power of Environment Assumptions for Unit Proofs. SEFM 2024: 366-384

SEAMOCK DSL

Expectation	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(f)	Mock function captures Nth positional argument, applies function f on it
invokeFn(f)	Mock function invokes function f with all arguments

MOCK TRUSTY ENVIRONMENT

```
1.  static size_t g_msg_size;
2.  constexpr auto invoke_get_msg =
3.      [](int chan, size_t *len) {
4.    *len = nd_size_t();
5.    g_msg_size = *len;
6.    return nd_int();
7.  };
8.  constexpr auto invoke_read_msg =
9.      [](int chan, char *msg) {
10.   char *blob = (char *)malloc(g_msg_size);
11.   memhavoc(blob, g_msg_size);
12.   sassert(msg);
13.   memcpy((size_t *)msg, (size_t *)blob, g_msg_size);
14.   sea_reset_modified(msg);
15.   return g_msg_size;
16. };
```

```
1.  constexpr auto get_msg_expectations =
2.      seamock::ExpectationBuilder()
3.          .invokeFn(invoke_get_msg)
4.          .times(seamock::Eq<1>())
5.          .build();
6.  constexpr auto read_msg_expectations =
7.      seamock::ExpectationBuilder()
8.      .invokeFn(invoke_read_msg)
9.      .times(seamock::Lt<2>())
10.     .build();
11. extern "C" {
12. MOCK_FUNCTION(
13.     get_msg, get_msg_expectations, int, (int, size_t *))
    MOCK_FUNCTION_W_ORDER(read_msg,
14.     read_msg_expectations,
15.     MAKE_PRED_FN_SET(get_msg),
        int, (int, char *))
16. SETUP_POST_CHECKS((get_msg, read_msg, put_msg))
17. }
```

MOCK TRUSTY ENVIRONMENT

```
1.  constexpr auto get_msg_expectations =
2.      seamock::ExpectationBuilder()
3.          .invokeFn(invoke_get_msg)
4.          .times(seamock::Eq<1>())
5.          .build();
6.  constexpr auto read_msg_expectations =
7.      seamock::ExpectationBuilder()
8.      .invokeFn(invoke_read_msg)
9.      .times(seamock::Lt<2>())
10.     .build();
11. extern "C" {
12. MOCK_FUNCTION(
13.     get_msg, get_msg_expectations, int, (int, size_t *))
    MOCK_FUNCTION_W_ORDER(read_msg,
14.     read_msg_expectations,
15.     MAKE_PRED_FN_SET(get_msg),
        int, (int, char *))
16. SETUP_POST_CHECKS((get_msg, read_msg, put_msg))
17. }
```

get_msg Builder pattern:

1. Delegate to invoke_get_msg
2. Make sure get_msg func is called once
3. Build the expectation

Similar to GoogleMock, SeaMock uses invokeFn to execute arbitrary code in the mock

MOCK TRUSTY ENVIRONMENT

```
1.  constexpr auto get_msg_expectations =
2.      seamock::ExpectationBuilder()
3.          .invokeFn(invoke_get_msg)
4.          .times(seamock::Eq<1>())
5.          .build();
6.  constexpr auto read_msg_expectations =
7.      seamock::ExpectationBuilder()
8.      .invokeFn(invoke_read_msg)
9.      .times(seamock::Lt<2>())
10.     .build();
11. extern "C" {
12. MOCK_FUNCTION(
13.     get_msg, get_msg_expectations, int, (int, size_t *))
    MOCK_FUNCTION_W_ORDER(read_msg,
14.     read_msg_expectations,
15.     MAKE_PRED_FN_SET(get_msg),
        int, (int, char *))
16. SETUP_POST_CHECKS((get_msg, read_msg, put_msg))
17. }
```

read_msg Builder pattern:

1. Delegate to invoke_read_msg
2. Make sure read_msg func is called less than twice
3. Build the expectation

MOCK TRUSTY ENVIRONMENT

```
1.  constexpr auto get_msg_expectations =
2.      seamock::ExpectationBuilder()
3.          .invokeFn(invoke_get_msg)
4.          .times(seamock::Eq<1>())
5.          .build();
6.  constexpr auto read_msg_expectations =
7.      seamock::ExpectationBuilder()
8.      .invokeFn(invoke_read_msg)
9.      .times(seamock::Lt<2>())
10.     .build();
11. extern "C" {
12. MOCK_FUNCTION(
13.     get_msg, get_msg_expectations, int, (int, size_t *))
    MOCK_FUNCTION_W_ORDER(read_msg,
14.     read_msg_expectations,
15.     MAKE_PRED_FN_SET(get_msg),
        int, (int, char *))
16. SETUP_POST_CHECKS((get_msg, read_msg, put_msg))
17. }
```

Mock function definitions

MOCK TRUSTY ENVIRONMENT

```
1.  constexpr auto get_msg_expectations =
2.      seamock::ExpectationBuilder()
3.          .invokeFn(invoke_get_msg)
4.          .times(seamock::Eq<1>())
5.          .build();
6.  constexpr auto read_msg_expectations =
7.      seamock::ExpectationBuilder()
8.      .invokeFn(invoke_read_msg)
9.      .times(seamock::Lt<2>())
10.     .build();
11. extern "C" {
12. MOCK_FUNCTION(
13.     get_msg, get_msg_expectations, int, (int, size_t *))
    MOCK_FUNCTION_W_ORDER(read_msg,
14.     read_msg_expectations,
15.     MAKE_PRED_FN_SET(get_msg),
        int, (int, char *))
16. SETUP_POST_CHECKS((get_msg, read_msg, put_msg))
17. }
```

MOCK TRUSTY ENVIRONMENT

```
1.  constexpr auto get_msg_expectations =
2.      seamock::ExpectationBuilder()
3.          .invokeFn(invoke_get_msg)
4.          .times(seamock::Eq<1>())
5.          .build();
6.  constexpr auto read_msg_expectations =
7.      seamock::ExpectationBuilder()
8.      .invokeFn(invoke_read_msg)
9.      .times(seamock::Lt<2>())
10.     .build();
11. extern "C" {
12. MOCK_FUNCTION(
13.     get_msg, get_msg_expectations, int, (int, size_t *))
    MOCK_FUNCTION_W_ORDER(read_msg,
14.     read_msg_expectations,
15.     MAKE_PRED_FN_SET(get_msg),
        int, (int, char *))
16. SETUP_POST_CHECKS((get_msg, read_msg, put_msg))
17. }
```

Define read_msg signature; add constraint that read_msg called before

MOCK TRUSTY ENVIRONMENT

```
1.  constexpr auto get_msg_expectations =
2.      seamock::ExpectationBuilder()
3.          .invokeFn(invoke_get_msg)
4.          .times(seamock::Eq<1>())
5.          .build();
6.  constexpr auto read_msg_expectations =
7.      seamock::ExpectationBuilder()
8.      .invokeFn(invoke_read_msg)
9.      .times(seamock::Lt<2>())
10.     .build();
11. extern "C" {
12. MOCK_FUNCTION(
13.     get_msg, get_msg_expectations, int, (int, size_t *))
    MOCK_FUNCTION_W_ORDER(read_msg,
14.     read_msg_expectations,
15.     MAKE_PRED_FN_SET(get_msg),
        int, (int, char *))
16. SETUP_POST_CHECKS((get_msg, read_msg, put_msg))
17. }
```

Post-conditions check

MOCK TRUSTY ENVIRONMENT

```
1.  static size_t g_msg_size;
2.  constexpr auto invoke_get_msg =
3.      [](int chan, size_t *len) {
4.    *len = nd_size_t();
5.    g_msg_size = *len;
6.    return nd_int();
7.  };
8.  constexpr auto invoke_read_msg =
9.      [](int chan, char *msg) {
10.   char *blob = (char *)malloc(g_msg_size);
11.   memhavoc(blob, g_msg_size);
12.   sassert(msg);
13.   memcpy((size_t *)msg, (size_t *)blob, g_msg_size);
14.   sea_reset_modified(msg);
15.   return g_msg_size;
16. };
```

```
1.  constexpr auto get_msg_expectations =
2.      seamock::ExpectationBuilder()
3.          .invokeFn(invoke_get_msg)
4.          .times(seamock::Eq<1>())
5.          .build();
6.  constexpr auto read_msg_expectations =
7.      seamock::ExpectationBuilder()
8.      .invokeFn(invoke_read_msg)
9.      .times(seamock::Lt<2>())
10.     .build();
11. extern "C" {
12. MOCK_FUNCTION(
13.     get_msg, get_msg_expectations, int, (int, size_t *))
    MOCK_FUNCTION_W_ORDER(read_msg,
14.     read_msg_expectations,
15.     MAKE_PRED_FN_SET(get_msg),
        int, (int, char *))
16. SETUP_POST_CHECKS((get_msg, read_msg, put_msg))
17. }
```

MOCK TRUSTY ENVIRONMENT

```
1.  static size_t g_msg_size;
2.  constexpr auto invoke_get_msg =
3.      [](int chan, size_t *len) {
4.    *len = nd_size_t();
5.    g_msg_size = *len;
6.    return nd_int();
7.  };
8.  constexpr auto invoke_read_msg =
9.      [](int chan, char *msg) {
10.   char *blob = (char *)malloc(g_msg_size);
11.   memhavoc(blob, g_msg_size);
12.   sassert(msg);
13.   memcpy((size_t *)msg, (size_t *)blob, g_msg_size);
14.   sea_reset_modified(msg);
15.   return g_msg_size;
16. };
```

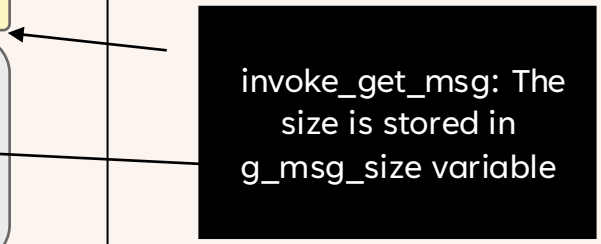
MOCK TRUSTY ENVIRONMENT

```
1.  static size_t g_msg_size;
2.  constexpr auto invoke_get_msg =
3.      [](int chan, size_t *len) {
4.    *len = nd_size_t();
5.    g_msg_size = *len;
6.    return nd_int();
7.  };
8.  constexpr auto invoke_read_msg =
9.      [](int chan, char *msg) {
10.   char *blob = (char *)malloc(g_msg_size);
11.   memhavoc(blob, g_msg_size);
12.   sassert(msg);
13.   memcpy((size_t *)msg, (size_t *)blob, g_msg_size);
14.   sea_reset_modified(msg);
15.   return g_msg_size;
16. };
```

The g_msg_size variable holds length of the message

MOCK TRUSTY ENVIRONMENT

```
1.  static size_t g_msg_size;
2.  constexpr auto invoke_get_msg =
3.      [](int chan, size_t *len) {
4.    *len = nd_size_t();
5.    g_msg_size = *len;
6.    return nd_int();
7.  };
8.  constexpr auto invoke_read_msg =
9.      [](int chan, char *msg) {
10.   char *blob = (char *)malloc(g_msg_size);
11.   memhavoc(blob, g_msg_size);
12.   sassert(msg);
13.   memcpy((size_t *)msg, (size_t *)blob, g_msg_size);
14.   sea_reset_modified(msg);
15.   return g_msg_size;
16. };
```



MOCK TRUSTY ENVIRONMENT

```
1.  static size_t g_msg_size;
2.  constexpr auto invoke_get_msg =
3.      [](int chan, size_t *len) {
4.    *len = nd_size_t();
5.    g_msg_size = *len;
6.    return nd_int();
7.  };
8.  constexpr auto invoke_read_msg =
9.      [](int chan, char *msg) {
10.   char *blob = (char *)malloc(g_msg_size);
11.   memhavoc(blob, g_msg_size);
12.   sassert(msg);
13.   memcpy((size_t *)msg, (size_t *)blob, g_msg_size);
14.   sea_reset_modified(msg);
15.   return g_msg_size;
16. };
```

The g_msg_size variable is used to return msg length in invoke_read_msg

WHY A NEW MOCKING FRAMEWORK?

return non deterministic int

Verification specific DSL features

- E.g., Express nondeterminism
  - LAZY_MOCK_FUNCTION(ssl_build_record_nonce, **int**, (unsigned char *, size_t *))

Dynamic dispatch used in unit test mock frameworks expensive in symbolic environments

- Use compile-time metaprogramming to transform DSL to C-functions!

```
constexpr auto get_msg_expectations =
    seamock::ExpectationBuilder()
        .invokeFn(invoke_get_msg)
        .times(seamock::Eq<1>())
        .build();
```

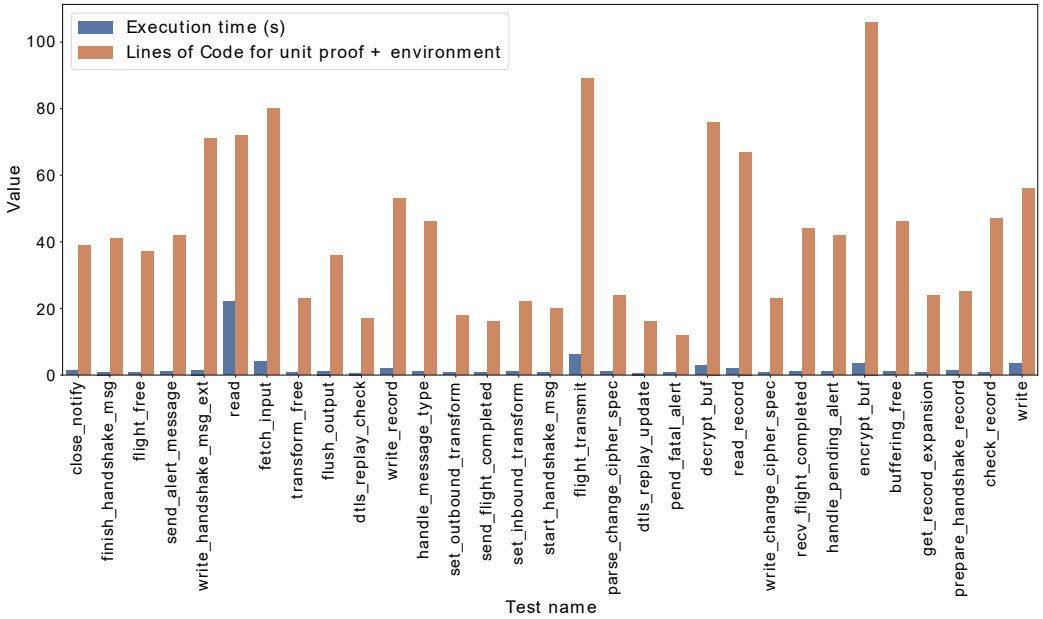
- 400 lines of C++-17 code
- Header only library

EVALUATION

MbedTLS is a C library that implements crypto primitives, SSL and DTLS protocols.

31 unit-proofs (**6kLOC** of SUT) for SeaBMC developed using SeaMock.

Improved proof development time, with caveats!



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

SEAMOCK

Mocks - New **environment** design point

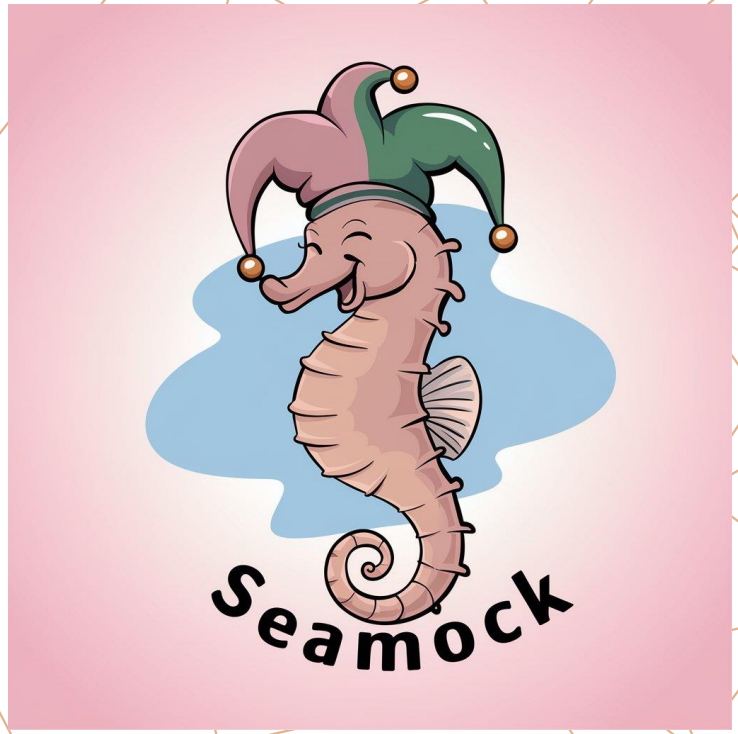
- Different from fakes, summaries
- Specify enough to maintain **function-local** invariants
- Do **not** replace fakes

Connect methodology to mocking in unit testing

- Mocking **widely used** in industry
- May **lead** to verification driven development

SeaMock mocking DSL & library

- **Optimized** for use in verification
- **Open source** at https://github.com/seahorn/seamock



References

SeaHorn

- http://seahorn.github.io/
- includes CHC (spacer), AbsInt (clam/crab), BMC (SeaBmc), Pointer Analysis (SeaDsa)

Case-Studies

- https://github.com/seahorn/verify-c-common
- https://github.com/seahorn/verifyTrusty
- https://github.com/seahorn/verify-rust

Papers

- Siddharth Priya, Xiang Zhou, Yusen Su, Yakir Vizel, Yuyan Bao, Arie Gurfinkel: Verifying Verified Code. The 19th International Symposium on Automated Technology for Verification and Analysis (ATVA 2021)
- Siddharth Priya, Xiang Zhou, Yusen Su, Yakir Vizel, Yuyan Bao, Arie Gurfinkel: Bounded Model Checking for LLVM. The 22nd International Conference on Formal Methods in Computer-Aided Design (FMCAD 2022)
- Siddharth Priya, Arie Gurfinkel: Ownership in Low-Level Intermediate Representation. FMCAD 2024: 292-300
- Siddharth Priya, Temesghen Kahsai, Arie Gurfinkel: Unlocking the Power of Environment Assumptions for Unit Proofs. SEFM 2024: 366-384
- Joseph Tafese, Siddharth Priya, Giuliano Losa, Arie Gurfinkel, Graydon Hoare: A Tale of Two Case Studies: A Unified Exploration of Rust Verification with SEABMC. FMCAD 2025

END

Bounded Model Checker for LLVM (C, C++, Rust …)

SEABMC

From C to SEA-IR

C

```
int main() {
  uint8_t x = nd_char();
  assume(x > 0 && x < 10);
  uint8_t *p = nd_bool() ? malloc(2*sizeof(uint8_t))
                         : malloc(sizeof(uint8_t));
  *p = x;
  *(p + 1) = 0;
  sassert(0 < *p && *p < 10);
  sassert(*(p + 1) == 0); // potential UB
  return 0;
```

SEA-IR

```
fun main() {
BB0:
  M0 = mem.init()
  R0 = nd_char()
  R1 = R0 > 0
  assume R1
  R2 = R0 < 10
  assume R2
  R3 = nd_bool()
  br R3, BB1, BB2
BB1:
  P1, M1 = malloc 2, M0
  br BB3
BB2:
  P2, M2 = malloc 1, M0
  br BB3
BB3:
  M3 = phi [M1,BB1],[M2,BB2]
  R4 = phi [P1,BB1],[P2,BB2]
  M4 = store R0, R4, M3
  R5 = R4 + 1
  M5 = store 0, R5, M4
  R6 = R0 > 0 && R0 < 10
  assert R6
  assert 0 == 0
  halt
}
```

Simplifying IR for VCGen

Single Assert

```
fun main() {
BB0:
  M0 = mem.init()
  R0 = nd_char()
  R1 = R0 > 0
  assume R1
  R2 = R0 < 10
  assume R2
  R3 = nd_bool()
  br R3, BB1, BB2
BB1:
  P1, M1 = malloc 2, M0
  br BB3
BB2:
  P2, M2 = malloc 1, M0
  br BB3
BB3:
  M3 = phi [M1,BB1],[M2,BB2]
  R4 = phi [P1,BB1],[P2,BB2]
  M4 = store R0, R4, M3
  R5 = R4 + 1
  M5 = store 0, R5, M4
  R6 = R0 > 0 && R0 < 10
  br R6, BB4, ERR
BB4:
  assume 0 != 0
  br ERR
ERR:
  assert false
  halt
}
```

Single Assume

```
fun main() {
BB0:
  M0 = mem.init()
  R0 = nd_char()
  R1 = R0 > 0 && R0 < 10
  R2 = nd_bool()
  br R2, BB1, BB2

BB1:
  P1, M1 = malloc 2, M0
  br BB3
BB2:
  P2, M2 = malloc 1, M0
  br BB3
BB3:
  M3 = phi [M1,BB1],[M2,BB2]
  R3 = phi [P1,BB1],[P2,BB2]
  M4 = store R0, R3, M3
  R4 = R3 + 1
  M5 = store 0, R4, M4
  R5 = R0 > 0 && R0 < 10
  br R5, BB4, ERR
BB4:
  R6 = false
  br ERR
ERR:
  A = phi [R6,BB4],[R1,BB3]
  assume A
  assert 0
  halt
}
```

Gated SSA

```
fun main() {
BB0:
  M0 = mem.init()
  R0 = nd_char()
  R1 = R0 > 0 && R0 < 10
  R2 = nd_bool()
  br R2, BB1, BB2

BB1:
  P1, M1 = malloc 2, M0
  br BB3
BB2:
  P2, M2 = malloc 1, M0
  br BB3
BB3:
  M3 = select R2, M1, M2
  R3 = select R2, P1, P2
  M4 = store R0, R3, M3
  R4 = R3 + 1
  M5 = store 0, R4, M4
  R5 = R0 > 0 && R0 < 10
  br R5, BB4, ERR
BB4:
  R6 = false
  br ERR
ERR:
  A = select R5, R6, R1
  assume A
  assert 0
  halt
}
```

Pure Data Flow and Verification Condition

SEA-IR in reduced PD form

```
fun main() {
entry:
  M0 = mem.init()
  R0 = nd_char()
  R1 = R0 > 0 && R0 < 10
  R2 = nd_bool()
  P1, M1 = malloc 2, M0
  P2, M2 = malloc 1, M0
  M3 = select R2, M1, M2
  R3 = select R2, P1, P2
  M4 = store R0, R3, M3
  R4 = R3 + 1
  M5 = store 0, R4, M4
  R5 = R0 > 0 && R0 < 10
  R6 = false
  A = select R5, R6, R1
  assume A
  assert 0
  halt
}
```

VCGen

SMT

$$
\begin{aligned}
& r_1 = (0 < r_0 \wedge r_0 < 10) \wedge \\
& \\
& p_1 = addr_0 \wedge m_1 = m_0 \wedge \\
& p_2 = addr_0 + 4 \wedge m_2 = m_0 \wedge \\
& \\
& r_3 = ite(r_2, p_1, p_2) \wedge \\
& \\
& r_4 = r_3 + 1 \wedge \\
& \\
& r_5 = (r_0 > 0 \wedge r_0 < 10) \wedge \\
& r_6 = false \wedge \\
& a = ite(r_5, r_6, r_1) \wedge \\
& a \wedge \\
& \neg false
\end{aligned}
$$

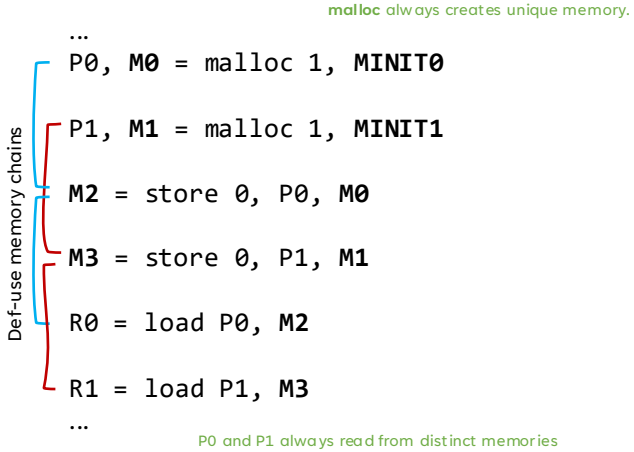
SEA-IR – purify memory operations

PR	::=	`fun main(){`BB$^+$`}`
BB	::=	L : PHI* S$^+$ (BR | `halt`)
BR	::=	`br` E, L, L | `br` L
PHI	::=	R = `phi` [R, L](, [R, L])* | M = `phi` [M, L](, [M, L])* | P = `phi` [P, L](, [P, L])*
S	::=	RDEF | MDEF | VS
RDEF	::=	R = E | P, M = `alloca` R, M | P, M = `malloc` R, M | R = `load` P, M | P = `load` P, M | M = `free` P, M
MDEF	::=	M = `store` R, P, M | M = `store` P, P, M
VS	::=	`assert` R | `assume` R

Unlimited registers: Each register has a type – scalar, pointer, or memory

All operations are pure: **SEA-IR** extends LLVM IR by making dependency information between memory operations explicit

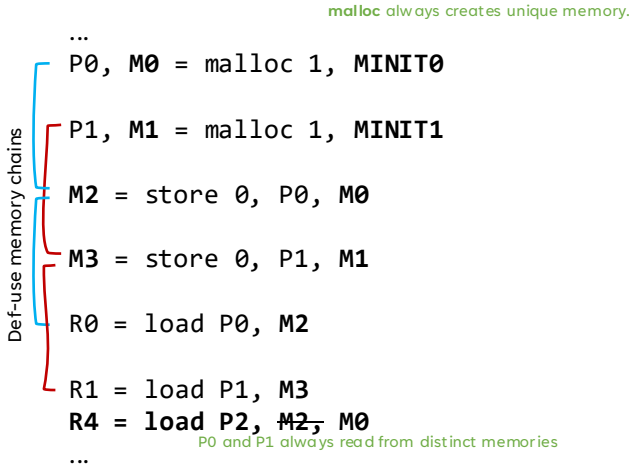
SEA-IR – purify memory operations



Example: SEA-IR program with pure memory operations

- **Blue** and **Red** are distinct def-use memory chains
- This distinction helps generate simpler VC

SEA-IR – purify memory operations



Example: SEA-IR program with pure memory operations

- **Blue** and **Red** are distinct def-use memory chains
- This distinction helps generate simpler VC

SEA-IR: Program transformation

Source prog.

```
int main() {
  int s = nd_int();
  assume(s > -5);
  if (s > 0) {
    s = s - nd_int();
  }
  assert(s > -5);
  return 0;
}
```

C program: **nd_int** returns a non-deterministic int; **assume** and **assert** have usual meanings

SA prog.

```
define main() {
BB0:
  R0 = nd_int()
  R1 = R0 > -5
  assume R1
  R2 = R0 > 0
  br R2, BB1, BB2
BB1:
  R3 = nd_int()
  R4 = R0 - R3
  br BB2
BB2:
  PHINODE = phi [R4, BB1], [R0, BB0]
  R5 = PHINODE > -5
  assume(!R5)
  assert false
  halt
}
```

SA program: SEA-IR program in control flow form with **phi** nodes. It has a single assert (SA).

GSA prog.

```
define main() {
BB0:
  R0 = nd_int()
  R1 = R0 > -5
  R2 = R0 > 0
  br R2, BB1, BB2
BB1:
  R3 = nd_int()
  R4 = R0 - R3
  br BB2
BB2:
  GAMMA = select R2, R4, R0
  R5 = GAMMA > -5
  R6 = !R5
  R7 = R1 && R6
  assume R7
  assert false
  halt
}
```

GSA program: SEA-IR program in gated SSA form (**GSA**). It has a single assume and a single assert (**SASA**).

VC

```
(r4 = r0 - r3) &&
(r2 = r0 > 0)
(gamma = ite(r2, r4, r0)) &&
(gamma > -5)
(r6 = !r5) &&
(r1 = r0 > -5) &&
(r7 = r1 && r6) &&
r7 &&
!false
```

VCGen from **GSA** program using pure dataflow analysis.

VC generation can happen from different SEA-IR forms – control flow or dataflow.

UNIVERSITY OF WATERLOO

Shadow memory and fat pointers

Shadow every byte (or word) of program memory with program state metadata

- Memcheck – addressable, initialized memory?
- Eraser – concurrent access follows locking discipline

  Recent CBMC-SSM extension has shadow memory for CBMC

  – CBMC-SSM: Bounded Model Checking of C Programs with Symbolic Shadow Memory, ASE 2022, Bernd Fischer, Salvatore La Torre, Gennaro Parlato, Peter Schrammel

Prog Memory	Metadata0	Metadata1	Metadata2
Addr0			
Addr1			
...			
AddrN			

Shadow mem representation

Address	Metadata0	Metadata1	Metadata2

Fat pointer representation

Some metadata can be "cached" at pointers instead of memory, saving memory accesses. This scheme is called Fat pointers.

Fat pointer application – detect OOB access

```
int main() {
  char *p = (char *) malloc(sizeof(char));
  *p = 255;
  *(p+8) = 255;
  return 0
}
```

OOB access;
Undefined behaviour

```
int main() {
  char *p = (char *) malloc(sizeof(char));
  sea_is_deref(p, 0);
  *p = 255;
  sea_is_deref(p, 8);
  *(p+8) = 255;
  return 0
}
```

```
sym(R1 = isderef P0 B) ==
    r1 = 0 <= p0.offset + B < p0.size
```

isderef semantics

Base Address	Offset	Size
p	0	1

Base Address	Offset	Size
p	8	1

Contrast with CBMC: CBMC overloads pointer bits to store metadata adding constraints on the addresses that can be modelled. Fat pointers have no such limitation!

UNIVERSITY OF WATERLOO

Shadow memory application – detect UAF

```
int main() {
  char *p = (char *)malloc(sizeof(char));
  *p = 0;
  free(p);
  *p = 255;
  return 0
}
```

UAF; Undefined behaviour

Intrinsic like `sea_is_alloc` operate on program metadata.

Note: This scheme relies on fat pointers that store base address.

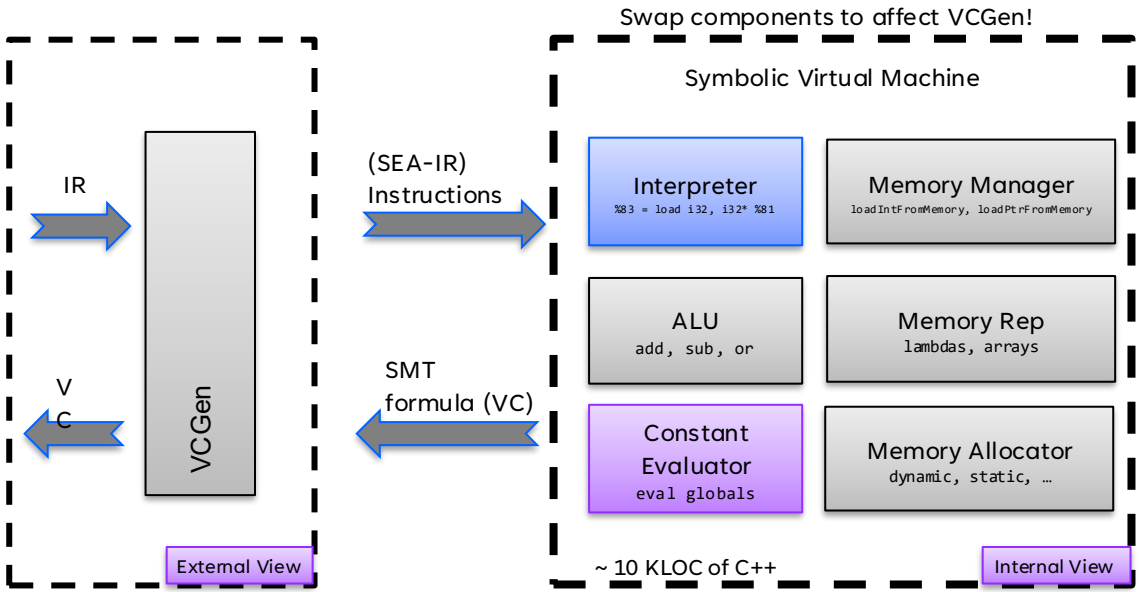
Intrinsics to track other program properties – e.g., `sea_is_mod` (RO memory integrity)

```
int main() {
  char *p = (char *) malloc(sizeof(char));
✓ sea_is_alloc(p);
  *p = 0;
  free(p);
✗ sea_is_alloc(p);
  *p = 255;
  return 0
}
```

Prog Memory	Base	Offset	Size	isAlloc
p	--	--	--	0 or 1

UNIVERSITY OF WATERLOO

BACKEND: VCGEN as a (symbolic) VM



112

aws-c-common benchmark verification time

Comparision with SeaBMC, CBMC, SMACK, SYMBIOTIC, KLEE

	SEABMC	CBMC	SMACK	SYMBIOTIC	KLEE
Total Time	**710s**	6,398s	6,370s	10,946s	5,741s

category	Statistics cnt	Statistics loc	SEABMC avg (s)	SEABMC std (s)	SEABMC time (s)	CBMC avg (s)	CBMC std (s)	CBMC time (s)	SMACK cnt	SMACK fld/to	SMACK avg (s)	SMACK std (s)	SMACK time (s)	SYMBIOTIC cnt	SYMBIOTIC fld/to	SYMBIOTIC avg (s)	SYMBIOTIC std (s)	SYMBIOTIC time (s)	KLEE cnt	KLEE avg (s)	KLEE std (s)	KLEE time (s)
arithmetic	6	202	1	0	**3**	4	0	22	6	2/0	3	1	18	6	0/0	135	281	809	6	1	0	5
array	4	390	2	1	**7**	6	0	23	4	0/1	53	98	213	4	0/0	11	4	44	4	26	2	103
array_list	24	3,150	3	4	**71**	19	33	450	24	0/0	5	1	126	23	0/0	43	68	980	24	41	38	994
byte_buf	29	2,908	1	1	**29**	9	10	252	29	0/2	27	50	788	29	0/0	40	162	1,168	27	59	96	1,592
total	169	20,790			**710**			6,398	4/20				6,370	10/5				10,946				5,741

TABLE II: Verification results for SEABMC, CBMC, SMACK, SYMBIOTIC, and KLEE. Timeout for SMACK and SEABMC is 200s, and 5,000s for SYMBIOTIC. **cnt**, **fld**, **to**, **avg**, **std** and **time**, are the number of verification tasks, failed cases, timeout cases, average run-time, standard deviation, and total run-time in seconds, per category.

Read only memory proof using shadow memory (rewrite 70 proofs)

SEABMC config	Total time
Shadow	**90s**
No shadow	143s

https://github.com/seahorn/verify-c-common

UNIVERSITY OF WATERLOO