Executable Counterexamples in Software Model Checking[⋆]

Jeffrey Gennari[1], Arie Gurfinkel[2], Temesghen Kahsai[3], Jorge A. Navas[4], and Edward J. Schwartz[1]

Carnegie Mellon University[1], University of Waterloo[2],
University of Iowa[3], and SRI International[4]

Abstract. Counterexamples — execution traces of the system that illustrate how an error state can be reached from the initial state — are essential for understanding verification failures. They are one of the most salient features of Model Checkers, which distinguish them from Abstract Interpretation and other Static Analysis techniques by providing a user with information on how to debug their system and/or the specification. While in Hardware and Protocol verification, the counterexamples can be replayed in the system, in Software Model Checking (SMC) counterexamples take the form of a textual or semi-structured report. This is problematic since it complicates the debugging process by preventing developers from using existing processes and tools such as debuggers, fault localization, and fault minimization.

In this paper, we argue that for SMC the most useful form of a counterexample is an *executable mock environment* that can be linked with the code under analysis (CUA) to produce an executable that exhibits the fault witnessed by the counterexample. A mock environment is different from a unit test since it can interface with the CUA at the function level, potentially allowing it to bypass complex logic that interprets program inputs. This makes mock environments easier to construct than unit tests. In this paper, we describe the automatic environment generation process that we have developed in the SeaHorn verification framework. We identify key challenges for generating mock environments from SMC counterexamples of complex memory manipulating programs that use many external libraries and function calls. We validate our prototype on the verification benchmarks from Linux Device Drivers in SV-COMP. Finally, we discuss open challenges and suggests avenues for future work.

1 Introduction

Software testing is the most widely used technique for assuring quality of a software system. Automated testing tools, such as fuzzers, generate a *test input* (or a test-case), which are concrete values for program inputs that are fed to the

[⋆] This material is based upon work supported by the Office of Naval Research under contract no. N68335-17-C-0558 and by the Defense Advanced Research Projects Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under contract no. N66001-18-C-4011. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Office of Naval Research, DARPA, or SSC Pacific. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), RGPAS-2017-507912.

Code Under Analysis (CUA). If the execution raises an exception, crashes, or produces unexpected output, then that test-case triggers a bug. Developers are familiar with such test-cases and can use them to help understand the nature of the bug and develop a fix.

Although testing can be very effective at finding bugs, it cannot uncover all bugs because exhaustively enumerating all program inputs is not possible. A complementary approach to testing is Software Model Checking (SMC) [1]. SMC has several advantages over testing. First, it can (symbolically) explore all program executions, and as a result, it can prove the *absence* of bugs in addition to finding them. Unlike some forms of testing (e.g., mutational fuzzing), SMC is completely automated and does not require user-provided test-cases or inputs.

One of the most important features of SMC (and Model Checking in general) is its ability to produce a counterexample when the property of interest is violated. A counterexample is a trace through the system that shows how the system reaches an error state from the initial state. The current state-of-the-art is for SMC tools to generate counterexamples as a machine readable document describing a set of assignments of variables to their corresponding values, or traces through an abstract transition system. For example, the SLAM verification project uses a special text format and a special visualizer for its counterexamples, and the Linux Driver Verification project uses an XML-based format indicating the line numbers and function calls that were executed. Most recently, the Software Verification Competition (SV-COMP) has adopted an XML-based format for its counterexamples.

These counterexample formats are often enhanced by a variety of visualizers to illustrate the relationship between a counterexample and a program. Most commonly, a visualizer simulates a debug session, by showing a counterexample as an execution over the program text. A recent study [25] has argued that a textual report from an analysis tool does not fit well into the usual development cycle, and that this is one of the leading reasons why developers do not adopt static program analysis tools.

In this paper, we argue that the most useful representation of a counterexample that a SMC can output for the developer is an *executable mock environment*. An executable mock environment E is a code module that implements the external environment used by the CUA C such that linking C and E together produces an executable that triggers the buggy execution witnessed by the counterexample. In other words, a mock environment lifts the counterexample into an executable code.

As an example, we show in Fig. 1 a simplified C snippet from the Linux Driver Verification Project (LDV) [26] and a conceptual C implementation of a mock environment. LDV programs are Linux kernel modules annotated with assertions that check for proper API usage (e.g., every lock is eventually unlocked and no lock is taken twice in a row). The C snippet shown on the left of Fig. 1 allocates

[1] Some authors make the distinction between static and dynamic SMC. The former analyzes statically all possible program executions while the latter is an adaptation for testing. Unless otherwise stated, we always refer to static SMC.

```c
extern int __VERIFIER_nondet_int(void);
extern int __VERIFIER_error(void);
extern void* ldv_ptr(void);

int main(int argc, char* argv[]) {
  void *p = ldv_ptr();

  if (p > (long) 2012) {
    if (__VERIFIER_nondet_int() > 456) {
      ...
      __VERIFIER_error();
    }
  }
  return 0;
}
```

```c
void* ldv_ptr(void) {
  static int ctr = 0;
  switch (ctr++) {
    case 0: return 2013;
    default: abort();
  }
}

int __VERIFIER_nondet_int(void) {
  static int ctr = 0;
  switch (ctr++) {
    case 0: return 457;
    default: abort();
  }
}
```

Fig. 1: C snippet (left) and an example of a mock environment implementation (right)

external memory by calling `ldv_ptr`, a special LDV function that represents a memory interaction between the device driver and the kernel. To represent an error during this interaction, `ldv_ptr` can return a pointer value greater than a predefined absolute address[2]. The mock environment on the right of Fig. 1 triggers the error function (`__VERIFIER_error`) by returning an invalid pointer (2013) from `ldv_ptr` and yielding 457 when `__VERIFIER_nondet_int` is called.

Mock environments are natural to software developers. They are analogous to traditional test doubles, such as mock objects[3], which are often used to simulate complex behaviors or external services in testing. Mock objects tend to be limited in their implementation; they must be manually configured, and perhaps involve recompilation or specific program design strategies to achieve desired behaviors. Furthermore, the mocks themselves become additional dependencies that must be maintained with test-cases. Conversely, mock environments are automatically generated from counterexamples and capture all the conditions necessary to replay error traces through the CUA. Developers need not worry about configuration or environmental dependencies; they can simply run the executable counterexample using their traditional tools such as a debugger.

The main challenge in generating mocks is to synthesize an environment that is sufficient to trigger a bug in the CUA while being a realistic enough representation of the real environment to be of interest to the developer. In principle, mock generation can be reduced to symbolic execution, which would guarantee that the mock is consistent with the operational semantics of the program. Unfortunately, in practice, state-of-the-art symbolic execution engines do not scale to this task. Existing symbolic execution engines are good at exploring many shallow executions, or opportunistically finding bugs at an end of a long concrete execution. None are good at finding a targeted non-trivial execution that satisfies some constraints found by an SMC [6].

In summary, we make the following contributions: (1) formally define a concrete semantics for executable counterexamples, (2) describe a general framework

[2] The constant 2012 is added by the LDV team as part of kernel modeling.

[3] `http://www.mockobjects.com/2009/09/brief-history-of-mock-objects.html`

$$\begin{array}{l}
a \ ::= n \ \mid \ v_i \ \mid \ a_1 \ op_a \ a_n \\
p \ ::= \mathsf{null} \mid \ v_p + a \\
b \ ::= \mathbf{true} \ \mid \ \mathbf{false} \ \mid \ \mathbf{not}\ b \ \mid \ b_1 \ op_b \ b_2 \ \mid \ a_1 \ op_r \ a_2 \ \mid \ p_1 \ op_p \ p_2 \\
S ::= \mathbf{skip} \ \mid \ \mathbf{error} \ \mid \ S_1; S_2 \ \mid \ \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 \ \mid \ \mathbf{while}\ b\ \mathrm{do}\ S\ \mathbf{end} \\
\qquad v_p := \mathbf{alloc}\ (sz) \ \mid \ v_i := \mathbf{load}\ (p) \ \mid \ v_p := \mathbf{load}\ (v_p{}') \\
\qquad \mathbf{store}\ (v_p, v_p{}') \ \mid \ \mathbf{store}\ (v_i, v_p{}') \ \mid \ v_i := a \ \mid \ v_p := p
\end{array}$$

Fig. 2: A simple imperative language

for building executable counterexamples, (3) describe an instance of the framework as implemented in SeaHorn [22], a state of the art software analysis tool, and (4) present a preliminary experimental evaluation of our framework implementation in SeaHorn by benchmarking it on the Linux Driver Verification set of benchmarks from SV-COMP.

2 Concrete Semantics for Executable Counterexamples

In this section, we formally define what we mean by a *counterexample* and a *mock environment*. To do so, we first define a simple imperative language that has an explicit error state and a corresponding concrete semantics. We then show how this language can be extended to represent *external* functions and memory allocations.

2.1 A Simple Imperative Language

To simplify the presentation, we first define a simple language restricted to integers and pointers and without function calls. The syntax is described in Figure 2. The set of program variables is $\mathcal{V} = \mathcal{V}_\mathcal{P} \cup \mathcal{V}_\mathcal{I}$, where $\mathcal{V}_\mathcal{P}$ and $\mathcal{V}_\mathcal{I}$ are the set of pointer and integer variables, respectively. We assume that the integer and pointer variables are disjoint, $\mathcal{V}_\mathcal{P} \cap \mathcal{V}_\mathcal{I} = \emptyset$. Integer and pointer variables are denoted with symbols $v_i \in \mathcal{V}_\mathcal{I}$ and $v_p \in \mathcal{V}_\mathcal{P}$, respectively. The symbol $v \in \mathcal{V}$ denotes a variable of either integer or pointer type. Boolean and arithmetic expressions are described by $b \in \mathsf{BExp}$ and $a \in \mathsf{AExp}$, respectively. We assume they are equipped with the standard boolean (op_b) and arithmetic (op_a) operators. Similarly, we define pointer expressions $p \in \mathsf{PExp}$, which are equipped with pointer equality and inequality operators (op_p).

We assume a classical structural operational semantics with a standard memory model for C programs. A pointer is a pair $\langle \mathit{Loc}, \mathit{Offset} \rangle$, where *Loc* is a unique identifier of a memory object of size $Sz(\mathit{Loc})$ and *Offset* is the byte offset in *Loc*, where $0 \leq \mathit{Offset} < Sz(\mathit{Loc})$. The number of possible memory objects is infinite. The special constant null is denoted by the pointer $(0, 0)$. We assume a function $Sz : \mathit{Loc} \mapsto \mathbb{N}$ that maps each memory object to its size.

$\langle \_, e_i, e_p, h, \omega \rangle \Rightarrow \langle e_i, e_p, h, \omega \rangle$	if $\omega = \text{true}$
$\langle \textbf{error}, e_i, e_p, h, \omega \rangle \Rightarrow \langle e_i, e_p, h, \text{true} \rangle$	
$\langle v_p := \textbf{alloc}(sz), e_i, e_p, h, \omega \rangle \Rightarrow$ $\langle e_i, e_p[v_p \mapsto c], h[c \equiv \langle Loc, 0 \rangle \mapsto \epsilon], \omega \rangle$	if $\omega = \text{false}$ and $Loc \notin \mathsf{Dom}(h)$ and $Sz[Loc \mapsto sz]$
$\langle \textbf{store}(v_p, v_p'), e_i, e_p, h, \omega \rangle \Rightarrow \langle e_i, e_p, h[e_p({v_p}') \mapsto h(e_p(v_p))], \omega \rangle$	if $\omega = \text{false}$
$\langle \textbf{store}(v_i, v_p'), e_i, e_p, h, \omega \rangle \Rightarrow \langle e_i, e_p, h[e_p({v_p}') \mapsto e_i(v_i)], \omega \rangle$	if $\omega = \text{false}$
$\langle v_p := \textbf{load}(v_p'), e_i, e_p, h, \omega \rangle \Rightarrow \langle e_i, e_p[v_p \mapsto h(v_p')], h, \omega \rangle$	if $\omega = \text{false}$
$\langle v_i := \textbf{load}(v_p), e_i, e_p, h, \omega \rangle \Rightarrow \langle e_i[v_i \mapsto h(v_p)], e_p, h, \omega \rangle$	if $\omega = \text{false}$
$\langle v_p := p, e_i, e_p, h, \omega \rangle \Rightarrow \langle e_i, e_p[v_p \mapsto \mathcal{P}[\![p]\!](e_i, e_p)], h, \omega \rangle$	if $\omega = \text{false}$
$\langle v_i := a, e_i, e_p, h, \omega \rangle \Rightarrow \langle e_i[v_i \mapsto \mathcal{A}[\![a]\!](e_i)], e_p, h, \omega \rangle$	if $\omega = \text{false}$
$\langle \textbf{skip}, e_i, e_p, h, \omega \rangle \Rightarrow \langle e_i, e_p, h, \omega \rangle$	
$\dfrac{\langle S_1, e_i, e_p, h, \omega \rangle \Rightarrow \langle e_i', e_p', h', \omega' \rangle}{\langle S_1; S_2, e_i, e_p, h, \omega \rangle \Rightarrow \langle S_2, e_i', e_p', h', \omega' \rangle}$	
$\dfrac{\langle S_1, e_i, e_p, h, \omega \rangle \Rightarrow \langle S_1', e_i', e_p', h', \omega' \rangle}{\langle S_1; S_2, e_i, e_p, h, \omega \rangle \Rightarrow \langle S_1'; S_2, e_i', e_p', h', \omega' \rangle}$	
$\langle \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2, e_i, e_p, h, \omega \rangle \Rightarrow \langle S_1, e_i, e_p, h, \omega \rangle$	if $\mathcal{B}[\![b]\!](e_i, e_p) = \text{true}$
$\langle \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2, e_i, e_p, h, \omega \rangle \Rightarrow \langle S_2, e_i, e_p, h, \omega \rangle$	if $\mathcal{B}[\![b]\!](e_i, e_p) = \text{false}$
$\langle \textbf{while}\ b\ \textbf{do}\ S\ \textbf{end}, e_i, e_p, h, \omega \rangle \Rightarrow$ $\langle S; \textbf{while}\ b\ \textbf{do}\ S\ \textbf{end}, e_i, e_p, h, \omega \rangle$	if $\mathcal{B}[\![b]\!](e_i, e_p) = \text{true}$
$\langle \textbf{while}\ b\ \textbf{do}\ S\ \textbf{end}, e_i, e_p, h, \omega \rangle \Rightarrow \langle e_i, e_p, h, \omega \rangle$	if $\mathcal{B}[\![b]\!](e_i, e_p) = \text{false}$

Fig. 3: Operational semantics for language described in Figure 2

To define a program state, we need environments that map both program variables and pointers to values, and a store that represents memory contents:

$$e_i \in Env_{\mathcal{I}} = \mathcal{V}_{\mathcal{I}} \mapsto \mathbb{Z} \qquad e_p \in Env_{\mathcal{P}} = \mathcal{V}_{\mathcal{P}} \mapsto \langle Loc, \text{Offset} \rangle$$
$$h \in \text{Store} = \langle \text{Loc}, \text{Offset} \rangle \mapsto \langle \text{Loc}, \text{Offset} \rangle \ \cup\ \mathbb{Z}\ \cup\ \epsilon$$

An integer environment $e_i \in Env_{\mathcal{I}}$ maps integer variables to integer values. A pointer environment $e_p \in Env_{\mathcal{P}}$ maps pointer variables to pointers. A store $h \in$ Store is a mapping from pointers to either pointers or integer values. The symbol ϵ denotes that the pointer points to uninitialized memory. We use functions $\mathcal{P}$, $\mathcal{B}$, and $\mathcal{A}$ to express the semantics of pointer, boolean and arithmetic expressions:

$$\begin{aligned} \mathcal{P} &: \ \mathsf{PExp} \to (Env_{\mathcal{I}} \times Env_{\mathcal{P}}) \to \langle Loc, \text{Offset} \rangle \\ \mathcal{B} &: \ \mathsf{BExp} \to (Env_{\mathcal{I}} \times Env_{\mathcal{P}}) \to \mathbb{B} \\ \mathcal{A} &: \ \mathsf{AExp} \to Env_{\mathcal{I}} \to \mathbb{Z} \end{aligned}$$

The semantics of boolean and arithmetic expressions is standard and the details are omitted for brevity. However, we describe here the semantics for pointers:

$$\mathcal{P}[\![p]\!](e_i, e_p) = \begin{cases} (0, 0) & \text{if } p \equiv \ \mathsf{null} \text{ or } (0, 0) = e_p(p) \\ (Loc, o + \ \mathcal{A}[\![a]\!] e_i) & \text{if } p \equiv \ v_p + a \text{ and } (Loc, o) = e_p(v_p) \end{cases}$$

The structural operational semantics for our language is given in Figure 3. A configuration $\langle S, e_i, e_p, h, \omega \rangle$ consists of a statement S, an integer environment e_i, a pointer environment e_p, a store h and a flag ω that indicates whether `error` has been executed. Given two configurations c_1 and c_2, the notation $c_1 \Rightarrow c_2$ means that c_2 is reachable from c_1 in one execution step according to the semantics. $\Rightarrow^*$ is the transitive closure of the $\Rightarrow$ relation. Our semantics tracks whether `error` is reached and sets ω to true if that is the case. The statement $p := \mathbf{alloc}(sz)$ allocates a fresh memory object of size sz and returns a pointer to it. The statement $v_p := p$ performs pointer arithmetic but does not read from memory. The statements $v_p := \mathbf{load}(v_p{}')$, $v_i := \mathbf{load}(v_p)$, $\mathbf{store}(v_i, v_p{}')$, and $\mathbf{store}(v_p, v_p{}')$ read and write memory. We assume that memory operations abort execution when the pointer operand cannot be resolved to a legal offset of an allocated memory object. For simplicity, we do not keep track of such runtime error states in our semantics. The rest is standard so we omit the details.

2.2 Extending with External Functions and Memory

One key feature of SMC is that the environment of the CUA does not need to fully defined. For example, external functions, which are called by the CUA but whose implementations are not in the CUA, and memory regions allocated by external functions are both permitted by SMC. This is vital, for instance, when model checking of Linux device drivers, as the whole system is not available. However, partially defined programs cannot be represented by the operational semantics presented so far. To represent external functions and memory regions, we first extend our syntax with a new statement:

$$v := \mathbf{extern\_alloc}(v_1, \ldots, v_n)$$

where the variables $v, v_1, \ldots, v_n$ can be either integers or pointers. Note that with some syntactic sugar this statement is enough to model both external function calls with parameters $v_1, \ldots, v_n$ and externally allocated memory.

We then extend our definition of a configuration as follows. In addition to $\langle S, e_i, e_p, h, \omega \rangle$, we need a global counter $\lambda \in \mathbb{N}$ used for a time-stamp. The counter is needed to distinguish external memory allocations across loop iterations. We also define two external environments e_i^{ext}, e_p^{ext} for integers and pointers whose values and memory are allocated externally, respectively:

$$e_i^{ext} \in \mathit{ExternalEnv}_{\mathcal{I}} = \mathbb{N} \times \overline{\mathcal{V}_{\mathcal{P}}} \times \mathit{Env}_{\mathcal{I}} \times \mathit{Env}_{\mathcal{P}} \times \mathit{Store} \mapsto \mathbb{Z}$$
$$e_p^{ext} \in \mathit{ExternalEnv}_{\mathcal{P}} = \mathbb{N} \times \overline{\mathcal{V}_{\mathcal{P}}} \times \mathit{Env}_{\mathcal{I}} \times \mathit{Env}_{\mathcal{P}} \times \mathit{Store} \mapsto (\mathrm{Loc}, \mathrm{Offset})$$

The environment e_i^{ext} (e_p^{ext}) is a mapping from a tuple consisting of: a time-stamp, a vector of program variables representing the arguments to the function, and the standard environments (i.e., integer and pointer environments and the store). We are now ready to define the semantics of our new statement:

$$\langle v := \textbf{extern\_alloc}(v_1, \ldots, v_n), e_i, e_p, h, \omega, \lambda, e_i^{ext}, e_p^{ext}\rangle =$$

$$\begin{cases} \langle e_i[v \mapsto n], e_p, h, \omega, \lambda + 1, e_i^{ext}, e_p^{ext}\rangle & \text{if } v \in \mathcal{V}_\mathcal{I} \text{ and} \\ & n = e_i^{ext}(\lambda, v_1, \ldots, v_n, e_i, e_p, h) \\ \langle e_i, e_p[v \mapsto c], h[c \equiv \langle Loc, O\rangle \mapsto \epsilon], & \\ \quad \omega, \lambda + 1, e_i^{ext}, e_p^{ext}\rangle & \text{if } v \in \mathcal{V}_\mathcal{P} \text{ and} \\ & \langle Loc, O\rangle = e_p^{ext}(\lambda, v_1, \ldots, v_n, e_i, e_p, h) \end{cases}$$

2.3 Counterexamples and Mock Environments

We can now formally define both *counterexamples* and *mock environments*:

Definition 1 (Counterexample and Mock Environment). *Given a program S_{entry}, a* counterexample *is defined as*

$$\langle S_{entry}, \emptyset, \emptyset, \emptyset, \mathit{false}, 0, e_i^{ext}, e_p^{ext}\rangle \Rightarrow^* \langle e_i, e_p, h, \mathit{true}, \lambda, e_i^{ext}, e_p^{ext}\rangle.$$

and a mock environment $\mathcal{E}$ *is defined as the pair of external environments,* $\langle e_i^{ext}, e_p^{ext}\rangle$.

The rest of this paper describes how to synthesize the external environments e_i^{ext} and e_p^{ext} from a SMC counterexample and how to combine it with the CUA in order to exercise the error location.

3 A Framework for Constructing Executable Counterexamples

In this section, we present our framework for generating mocks from counterexamples produced by a Software Model-Checker (SMC), and the process of generating an executable that links the code under analysis (CUA) (which may be partially defined) with the mock to form a fully defined executable. The framework is illustrated in Figure 4. Rectangular boxes denote the main components, and the labeled arrows between these components denote the inputs and outputs of these components.

The main components of the framework are: (a) a Software Model Checker, (b) Directed Symbolic Execution, (c) a Mock Environment Builder, and (d) an External Memory Virtualization. The input to the framework is the Code Under Analysis (CUA), which contains an embedded safety property. An output, if possible, is a fully-defined executable that takes no inputs and references no external functions, and that violates the safety property when it is executed. We summarize each component and corresponding assumptions in the rest of this section. An instance of this framework using the Software Model Checker SeaHorn is presented in Section 4.

Software Model Checker. In the first step, an SMC is used to identify a potential buggy behavior of the CUA. We assume that the SMC finds a counterexample,

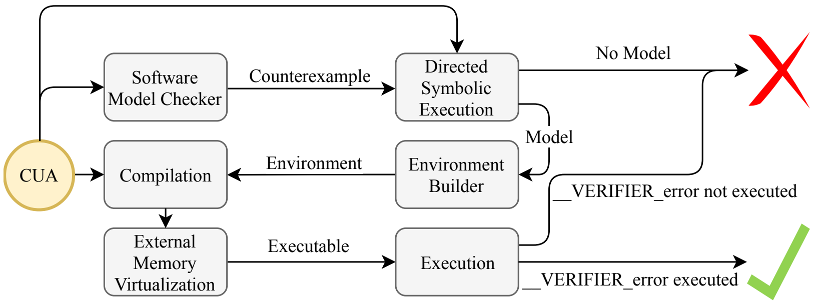

Fig. 4: Executable Counterexample Generation

since otherwise nothing needs to be generated. We make minimal assumptions about SMC. First, we assume that the safety property is already combined with the CUA. This, for example, can be done via a common convention of reducing safety verification to checking reachability of a designated error function, such as the `__VERIFIER_error` function used by SV-COMP. Second, we assume that the SMC can produce a trace indicating the loops executed by the counterexample and their corresponding number of iterations. However, we do *not* require the SMC to produce a detailed trace. This is necessary to allow the SMC to use simplification and optimization techniques during verification, some of which might make it difficult to extract a detailed trace after the analysis. Third, we do not require the SMC to be sound with respect to the C operational semantics. This is a necessary assumption because most current SMC techniques sacrifice soundness for scalability. Common soundness issues are related to undefined behavior, bit-precise semantics of integer operations, or memory modeling. Although we do not make any assumption about the SMC's soundness, we hope that the SMC is sound with respect to *some* useful subset of the language's operational semantics.

Directed Symbolic Execution (DirSE). In this step, a counterexample produced by SMC is analyzed by Directed Symbolic Execution. The main purpose of DirSE is to reproduce the counterexample found by the SMC and produce a more precise counterexample with respect to the concrete semantics. First, DirSE produces a Control Flow Graph (CFG) of the program which is sliced with respect to the counterexample trace. A key observation is that the sliced CFG is acyclic since each loop is unrolled using the information from the trace. Recall that the SMC must produce counterexamples given as traces indicating the number of times loops are executed. However, since we do not require the SMC to produce a detailed trace, the sliced CFG can still contain a large number of paths. Next, DirSE tries to prove that `__VERIFIER_error` is still reachable. This search process can be quite challenging because of the need for bit-precise semantics of integer operations, potential undefined behavior, and the presence of external memory allocation and functions which are not defined in the CUA. A successful output of DirSE is a counterexample generated by a SMT solver using bit-level precision. If a counterexample cannot be obtained because DirSE determines that no buggy execution exists in the sliced CFG, the SMC's counterexample is deemed unsound and the counterexample generation process is aborted.

Mock Environment Builder (MB). This component takes the detailed trace produced by DirSE and produces a mock environment in the form of object code. Essentially, it "internalizes" all external functions by creating mocks for them. Thus, the main task for the mock builder is to produce all values and memory addresses for all the memory that is allocated outside of the CUA. This is quite challenging. While the addresses for return values can be extracted from the counterexample, the mock builder is not aware whether an external call is allocating memory or what happens to a pointer that is passed as a parameter.

External Memory Virtualization (EMV). The last component of the framework takes the executable binary produced by linking the CUA and the mock together, and ensures that, when executed, it violates the safety property. The main challenge is to ensure that all memory addresses generated by the MB are *valid* so that each memory access can be resolved to a legal offset of an allocated memory object, while at the same time the complete execution triggers a property violation. In our framework, the MB does not allocate memory and therefore, it cannot map the (abstract) memory addresses generated by DirSE to valid (i.e., allocated) memory. This is the main task of the EMV which translates between the two types of addresses. The EMV provides a virtual external memory to the executable, ensuring that no memory access ever triggers a program failure during the execution. More precisely, the EMV traps each memory access of the program, and, whenever the access appears to reference an unallocated memory region, it either redirects the access to a special memory region, or, simulates a valid memory access by providing a default value back to the program. Having multiple choices in how to map unallocated memory regions to valid ones is the reason why MB is decoupled from EMV.

Finally, we believe that our framework is general enough so that almost any verification tool can be plugged in. However, this does not mean that all tools can clearly benefit from all our components or the framework itself. For instance, Bounded Model Checking (BMC) can usually produce bit-precise counterexamples, and as a result would not benefit from DirSE. Some Test Case Generation (TCG) tools (see Section 6) model the concrete semantics of the program and allocate memory on-the-fly. Thus, they might not benefit from our framework at all. The main advantage of our framework is, however, that it separates the problem of model-checking a program from the generation of an executable counterexample. This is vital for scalability since it allows the SMC to perform abstractions which would be difficult, if not impossible, to apply on BMC and TCG tools.

```
DirSE(cfg, cex)
1  Build a sliced acyclic cfg based on the cex
2  Let $\phi$ be the verification conditions of cfg
3  Let $\phi_{alloc}$ be the encoding of memory allocation constraints
4  if $\phi \wedge \phi_{alloc}$ is UNSAT (or timeout) then
5      print "no concrete CEX found"
6      return $\emptyset$
7  else
8      return model $M$ of $\phi \wedge \phi_{alloc}$
```

Fig. 5: DirSE in SeaHorn implemented as a BMC problem

4 Executable Counterexample Generation in SeaHorn

In this section, we present an instance of our framework using SeaHorn [22], a publicly available Software Model Checker. We organize the section following the same structure as Section 3 describing the implementation details of each of the four main components of the framework: Software Model Checker, Directed Symbolic Execution, Mock Environment Builder, and External Memory Virtualization.

The SeaHorn *Software Model Checker.* SeaHorn is a SMC for C/C++ programs based on the LLVM framework. It uses `clang` to compile programs to the LLVM intermediate representation, applies many of the LLVM optimizations to pre-process the code before analysis, and then uses a custom analysis engine based on Abstract Interpretation and Constrained Horn Clauses for verification. SeaHorn is sound with respect to its specialized semantics of C. In particular, it assumes that all integers are of arbitrary precision (i.e., unbounded or mathematical), and assumes a C-like memory model [23]. Furthermore, because SeaHorn relies on multiple LLVM components, which aggressively optimize undefined computations, the presence of undefined behavior (e.g., signed integer overflow, out-of-bound array access, and reads from uninitialized memory) in a program may significantly affect its interpretation.

Directed Symbolic Execution in SeaHorn*.* A high-level description for DirSE implemented in SeaHorn is shown in Fig. 5. The input to DirSE is a counterexample CEX produced by SeaHorn. The counterexample CEX only indicates which loop heads must be executed and for how many iterations, and represents many potential execution paths. From CEX, DirSE first constructs a sliced acyclic CFG that contains all executions witnessed by CEX (line 1). Symbolic execution over the CFG is reduced to a Bounded Model Checking (BMC) problem. The verification condition of the CFG is encoded into a SMT formula ϕ, and the satisfiability of ϕ is checked by an SMT solver. The BMC is specialized for handling dynamically allocated memory. We use points-to analysis [23] to partition the memory used by the CFG into disjoint regions, represented by a points-to graph G_{Mem}. Next, all behaviors of the CFG are encoded into verification conditions ϕ using bit-precise semantics of all of the LLVM instructions (line 2). In the formula ϕ, we represent each memory region in G_{Mem} by an

```
MockEnvironmentBuilder(cex, model)
1   let $m$ be a map $String \times (\tau_1 \times \ldots \times \tau_k \to \tau_{ret}) \to Vec(int)$
2   foreach $s \in$ cex
3       if $s$ is external callsite $\mathtt{v} := \mathtt{f}(\overline{\mathtt{v}})$ then
4           $key = (\mathtt{nameof}(\mathtt{f}), \mathtt{typeof}(\mathtt{f}))$
5           $m[key] := \mathtt{add}(m[key], \textsc{model}(\mathtt{v}))$
6   endfor
7   foreach $((f, T), \textsc{Vals}) \in m$
8       EmitCode($f, T, \textsc{Vals}$)
9   endfor

EmitCode($\textsc{f}, \tau_1 \times \ldots \times \tau_k \to \tau_{ret}, \textsc{Vals}$)
10  add global counter for $\lambda_{\textsc{f}}$ initially to 0
11  add function declaration for $\textsc{f}$ with type $\tau_1 \times \ldots \times \tau_k \to \tau_{ret}$
12  add function body:
13      if ($\lambda_{\textsc{f}}$ == 0) then $\textsc{Vals}[0]$
14      else if ($\lambda_{\textsc{f}}$ == 1) then $\textsc{Vals}[1]$
15      ...
16      else $\textsc{Vals}[\mathtt{len}(\textsc{Vals}) - 1]$
17      $\lambda_{\textsc{f}} := \lambda_{\textsc{f}} + 1$
```

Fig. 6: High-level description of the Mock Environment Builder in SeaHorn

array, and each memory access is mapped to an array select or store operation respectively, by associating a pointer to its corresponding memory region. Finally, extra constraints ϕ_{alloc} are generated to map regions to memory addresses (line 3) which are consistent with the C memory model. These constraints ensure, for instance, that allocated pointers are not `NULL`, that they are disjoint, and that no two allocated segments intersect. More precise modeling of memory allocation is possible (e.g., all memory addresses are 4-byte or 8-byte aligned) but at the expense of increasing the solving time. Finally, an SMT solver checks for satisfiability of $\phi \wedge \phi_{alloc}$. If the solver returns UNSAT (or times out) the process is aborted. Otherwise, the model corresponding to the concrete counterexample is returned. The model is extended to contain meta-data information so that each variable in the model can be mapped back to its corresponding LLVM variable.

Mock Environment Builder in SeaHorn. A description of the MB is shown in Fig. 6. The MB produces an LLVM bitcode file that provides definitions for all of the external functions in the CUA. The MB proceeds in two phases. In the first phase (lines 2–6), the MB walks the concrete counterexample produced by DirSE and collects all external calls of the form $\mathtt{v} := \mathtt{f}(\overline{\mathtt{v}})$. It then uses the model M from DirSE to find and record the return value v of the call-site. This represents the only possible side-effect since we assume external functions do not modify global state or any of their arguments. In the second phase (lines 7–9), the MB emits LLVM bitcode (function EmitCode) defining each external function. For each external function f, it constructs a body B_f that tracks the number of times it is called and returns an appropriate value based on the order

```
 1  @lambda_1 = private global i32 0
 2  @lambda_2 = private global i32 0
 3  @int_vals = private constant [1 x i32] [i32 457]
 4  @ptr_vals = private constant [1 x i8*] [i8* inttoptr (i32 2013 to i8*)] ;
 5
 6  define i32 @__VERIFIER_nondet_int() {
 7  entry:
 8    %0 = load i32, i32* @lambda_1
 9    %1 = add i32 %0, 1
10    store i32 %1, i32* @lambda_1
11    %2 = i32* getelementptr inbounds ([1 x i32], [1 x i32]* @int_vals, i32 0, i32 0)
12    %3 = call i32 @__seahorn_get_value_i32(i32 %0, i32* %2, i32 1)
13    ret i32 %3
14  }
15
16  define i8* @ldv_ptr() {
17  entry:
18    %0 = load i32, i32* @lambda_2
19    %1 = add i32 %0, 1
20    store i32 %1, i32* @lambda_2
21    %2 = i8* bitcast ([1 x i8*]* @ptr_vals to i8*)
22    %3 = call i8* @__seahorn_get_value_ptr(i32 %0, i8* %2, i32 1)
23    ret i8* %3
24  }
25
26  declare i32 @__seahorn_get_value_i32(i32, i32*, i32)
27  declare i8* @__seahorn_get_value_ptr(i32, i8*, i32)
```

Fig. 7: Example of mock environment in LLVM bitcode corresponding to the C program in Fig. 1

of the call. That is, in the first call to f, B_f returns the first value that f returned in the counterexample, in the second call it returns the second value, etc.

As an example, Fig. 7 shows the mocks (in LLVM bitcode) that MB generated for the Linux Driver Verification (LDV) program introduced earlier in Fig. 1. Lines 6–14 and 16–24 provide definitions for the two external functions in that code: `__VERIFIER_nondet_int` and `ldv_ptr`, respectively. Lines 1–4 define the global variables used by the two functions. Since the code of the two functions follows exactly the same structure, we focus on the definition of `ldv_ptr`. The function is assigned its own global counter, `lambda_2`, to track the number of calls. Lines 18–20 increment the counter each time the function is called. The function is also assigned a global array `ptr_vals` containing the values that will be returned by each call to `ldv_ptr`; these values are extracted from the concrete counterexample. (This array is called Vals in EmitCode in Fig. 6.) Finally, a call to our run-time library function `__seahorn_get_value_ptr` is used to retrieve an appropriate value from `ptr_vals` using the current value of `lambda_2`. We postpone the definitions of the two external functions, `__seahorn_get_value_i32` and `__seahorn_get_-value_ptr`, until we describe our next component.

External Memory Virtualization in SeaHorn. The EMV instruments the CUA with memory load and store hooks that control access to memory. This is achieved by replacing each load or store instruction in the CUA with a function call to the special functions `__seahorn_mem_load` and `__seahorn_mem_store`, respectively. Note that it is sufficient to instrument only instructions whose cor-

```
const int MEM_REGION_SIZE_GUESS = 4196;
const int TYPE_GUESS = sizeof(int);
std::map<intptr_t, intptr_t, std::greater<intptr_t>>
absptrmap;

void __VERIFIER_error() {
  printf("[sea]␣__VERIFIER_error␣was␣executed\n");
  exit(1);
}

int32_t __seahorn_get_value_i32(int ctr, intptr_t *g_arr, int g_arr_sz) {
  if (ctr >= g_arr_sz)
    return 0;
  else
    return g_arr[ctr];
}

intptr_t __seahorn_get_value_ptr(int ctr, intptr_t *g_arr, int g_arr_sz) {
  if (ctr >= g_arr_sz) return 0;
  intptr_t absptr = g_arr[ctr];
  size_t sz = MEM_REGION_SIZE_GUESS * TYPE_GUESS;
  absptrmap[absptr] = absptr + sz;
  return absptr;
}

bool is_external_address (void *addr) {
  intptr_t ip = intptr_t (addr);
  auto it = absptrmap.lower_bound (ip+1);
  if (it == absptrmap.end()) return false;
  intptr_t lb = it->first;
  intptr_t ub = it->second;
  return (ip >= lb && ip < ub);
}

bool is_valid_address (void *addr) {
  return !is_external_address (addr);
}

void __seahorn_mem_load (void *dst, void *src, size_t sz) {
  if (is_valid_address (src)) {
    memcpy (dst, src, sz);
  } else {
    // ignore read from an illegal memory address
    bzero(dst, sz);
  }
}

void __seahorn_mem_store (void *src, void *dst, size_t sz) {
  if (is_valid_address (dst)) {
    memcpy (dst, src, sz);
  } else { // ignore write to illegal memory address
  }
}
```

Fig. 8: External Virtualization implemented in SeaHorn

responding memory object might alias with an external object. The goal of these hooks is to map external "virtual" memory to real memory. This is vital because if a pointer that is externally allocated (e.g., `p` at line 6 on the left of Fig. 1) does not refer to a real memory address, the executable counterexample will probably crash when the pointer is dereferenced.

We have implemented EMV as an LLVM pass that replaces every load and store instructions with calls to the corresponding functions in our run-time library. A simplified version of the source code of these functions is shown in Fig. 8. We also present the implementation of the functions `__seahorn_get_value_i32` and `__seahorn_get_value_ptr` discussed earlier. These functions check for a given value in a given global array and return it to the caller. The case of `__seahorn_get_value_ptr` is a bit more involved. Whenever a pointer to an external memory object is returned, we need to guess the size of the corresponding allocated object. Unfortunately, it is not possible in general. Instead, we guess the size based on the type. We assume that all addresses within the guessed regions are externally allocated.

The definition of `__seahorn_mem_load` and `__seahorn_mem_store` are shown on the right of Fig. 8. These functions decide whether a pointer being dereferenced is allocated by the CUA or not (`is_valid_address`). For this, we use the map `absptrmap`. If the dereferenced pointer is within the bounds of any of the memory objects externally allocated then the address is considered *invalid*, otherwise it is considered to be allocated by the CUA, and, therefore, *valid*. If the pointer is valid then both `__seahorn_mem_load` and `__seahorn_mem_store` implement the original semantics of load and store. Otherwise, a load returns a pointer pointing to a region with all its contents written by zeroes and store is ignored. Note that although simple, this solution is sufficient for many of our benchmarks.

5 Experimental Evaluation

In this section, we report on the evaluation of our framework as implemented in SeaHorn. Our goal is to show that the generation of executable counterexamples is feasible on a set of non-trivial benchmarks. In the future, it would be interesting to evaluate the effectiveness of executable counterexamples compared to other outputs from an SMC, such as textual reports. All experiments were done on a 16 core, 3.5GHz Intel Xeon CPU and 64GB of RAM. Each component of our framework was restricted to 5 minutes CPU and 4GB memory limits.

For the evaluation, we took all benchmarks in the `Systems`, `DeviceDrivers`, and `ReachSafety` categories of SV-COMP 2018. These categories are representative of real code. In total, this yielded 356 unsafe benchmarks. From those, our SMC solved 144, failed in 18, and ran out of resources in 194. DirSE successfully concretized 141 counterexamples (out of 144). The three failures are due to an abstraction mismatch between SMC and DirSE: SMC is not bit-precise but DirSE is bit-precise. MB and EMV were successful on all 141 concretized counterexamples. Finally, we ran all the binaries witnessing a counterexample. We observed three outcomes: (a) the executable found the dedicated error function `__VERIFIER_error` in 24 cases, (b) the executable terminated but it did not execute `__VERIFIER_error` in 44 cases, and (c) the executable ran out of resources in 73 cases.

The detailed results for the successful 24 cases are shown in Table 1. The table reports on the time in seconds taken by the SMC, DirSE, MB, and counterexample execution, respectively. We also show the number of CFG cut-points (#CP)

	SMC		DirSE		MB+EMV	Exec
Program	T(s)	#CP	T(s)	#BB	T(s)	T(s)
module_get_put-drivers-net-wan-farsync	8.72	3	12.66	11	0.7	0.0
32_7_linux-32_1-drivers-staging-keucr-keucr	2.38	3	0.88	11	2.17	0.0
32_7_single_drivers-usb-image-microtek	0.76	3	0.02	6	0.78	0.0
linux-3.12-rc1-144_2a-drivers-net-wireless-mwifiex-mwifiex_usb	23.39	3	13.82	15	0.74	0.0
32_7_cilled_linux-32_1-drivers-usb-image-microtek	0.64	3	0.01	6	0.79	0.0
32_7_cilled_linux-32_1-drivers-media-dvb-dvb-usb-dvb-usb-dib0700	2.19	3	0.48	11	2.76	0.0
32_7_cilled_linux-32_1-drivers-isdn-capi-kernelcapi	0.92	3	6.37	11	1.51	0.0
32_7_cilled_linux-32_1-drivers-media-video-mem2mem_testdev	5.28	3	3.5	16	0.8	0.0
32_7_cilled_linux-32_1-drivers-usb-storage-usb-storage	30.59	3	124.27	11	1.68	0.0
32_7_single_drivers-staging-media-dt3155v4l-dt3155v4l	2.63	3	5.47	12	0.93	0.0
43_1a_cilled_linux-43_1a-drivers-misc-sgi-xp-xpc	105.8	5	2.64	31	2.0	0.0
m0_drivers-usb-gadget-g_printer-ko-106_1a-2b9ec6c-1	8.35	2	0.41	16	0.65	0.0
linux-3.12-rc1.tar.xz-144_2a-drivers-staging-media-go7007-go7007-loader	0.82	5	0.24	35	0.44	0.0
205_9a_linux-3.16-rc1.tar.xz-205_9a-drivers-net-ppp-ppp_synctty	44.32	6	3.46	61	0.71	0.0
205_9a_linux-3.16-rc1.tar.xz-205_9a-drivers-net-wan-hdlc_ppp	195.22	5	57.41	52	0.66	0.0
43_2a_linux-3.16-rc1.tar.xz-43_2a-drivers-usb-host-max3421-hcd	2.3	4	5.28	36	0.82	0.0
linux-stable-9ec4f65-1-110_1a-drivers-rtc-rtc-tegra	0.78	6	0.2	35	0.52	0.0
linux-stable-39a1d13-1-101_1a-drivers-block-virtio_blk	1.71	5	7.04	37	0.52	0.0
linux-stable-42f9f8d-1-111_1a-sound-oss-opl3	6.03	4	14.08	22	0.61	0.0
linux-stable-2b9ec6c-1-106_1a-drivers-usb-gadget-g_printer	51.12	4	28.46	37	0.67	0.0
linux-stable-39a1d13-1-101_1a-drivers-block-virtio_blk	1.63	5	0.84	33	0.66	0.0
linux-stable-2b9ec6c-1-106_1a-drivers-usb-gadget-g_printer	43.1	4	17.29	26	0.69	0.0
linux-stable-d47b389-1-32_7a-drivers-media-video-cx88-cx88-blackbird	39.48	4	27.18	96	0.75	0.0
linux-4.2-rc1.tar.xz-08_1a-drivers-md-md-cluster	5.84	5	12.0	23	0.68	0.0

Table 1: Experimental results for validated counterexamples in SeaHorn

of the counterexample returned by SMC and the number of basic blocks (#BB) that DirSE considered based on the counterexample. Note that sometimes DirSE takes significantly longer than SMC. This is expected because DirSE uses more complex semantics. Mock construction and execution take a negligible amount of time.

Analysis of Results and Current Limitations. Results show that constructing executable counterexamples is possible using current Software Model Checking techniques. The main challenge is improving techniques for extracting the memory model assumed by the SMC. Manually inspecting the failing cases shows that the pointers extracted from external allocation sites are often dereferenced further. Our current strategy traps such dereferences and replaces them with some default values. While this is sometimes sufficient, it does not always work. We have tried replacing such addresses by symbolic memory and using a symbolic execution engine, but this did not scale.

One manual solution we found is to replace external dereferences by external functions calls. For example, dereferencing an external field `foo->f` is replaced by a call to an external function `get_foo_f(foo)` that returns the value of the field. Such external calls are trapped by the mock to produce the required value. Selectively applying this manual technique, we converted several failing cases to successful executable counterexamples. For example, Fig. 9 shows the changes for `usb_urb-drivers-input-misc-keyspan_remote.ko_false-unreach-call.cil.out.i.pp.i`. Three `get` functions are added to wrap around memory references (the original code is in comments). These functions allow the MB to inject the right values to guide the program toward the counterexample. While currently this is a manual process, we believe it can be significantly automated in the future.

```
extern __u8 get_bEndpointAddress(struct usb_endpoint_descriptor const *e) ;
extern __u8 get_bmAttributes(struct usb_endpoint_descriptor const *e);
extern struct device* get_dev(struct usb_interface *iface);

__inline static int usb_endpoint_dir_in(struct usb_endpoint_descriptor const *ed) {
  /* return (((int const )ed->bEndpointAddress & 128) == 128); */
  return (((int const) get_bEndpointAddress(ed) & 128) == 128);
}
__inline static int usb_endpoint_xfer_int(struct usb_endpoint_descriptor const *ed) {
  /* return (((int const )ed->bmAttributes & 3) == 3);  */
  return (((int const )get_bmAttributes(ed) & 3) == 3);
}
__inline static void *usb_get_intfdata(struct usb_interface *intf ) {
  void *tmp___7 ;
  /* struct device const* dev = (struct device const *)(& intf->dev); */
  struct device const* dev = get_dev(intf);
  tmp___7 = dev_get_drvdata(dev);
  return (tmp___7);
}

int main(void) {
  /* struct usb_interface *var_group1; */
  struct usb_interface *var_group1 = ldv_undefined_pointer();
  ...
}
```

Fig. 9: Example of manual modifications to generate executable counterexample

6 Related Work

Generating executable tests from Software Model Checking counterexamples is not a new idea. One of the earliest approaches was proposed by Beyer et al. [3]. However, they do not consider programs that manipulate memory or use external functions.

Executable Counterexamples from SMC. Rocha et al. [29] propose EZProofC, a tool to extract information about program variables from counterexamples produced by ESBMC [12] and generate executable programs that reproduce the error. First, EZProofC extracts the name, value, and line number for each variable assignment in the counterexample. Second, the code is instrumented so that the original assignment statements are replaced with assignments of the corresponding values in the counterexample. This approach is closely related to ours, but there are some important differences. First, EZProofC assumes that it is easy to match assignments in ESBMC counterexamples to the original source code. This assumption does not hold if verification is combined with aggressive optimization or transformations. In our experience, such optimizations are essential for scalability. In contrast, we make no such assumptions. More importantly, EZProofC does not deal with dereferences of pointers allocated by external functions. We found this to be prevalent in benchmarks, difficult to address, and is a primary focus of our work.

Muller and Ruskiewicz [28] produce .NET executable from a Spec# program and a symbolic counterexample. Counterexamples may include complex types including classes, object creation, and initialization of their fields. There are again some key differences. First, they target Spec#, a language without direct pointer manipulation, while we target C. As a result, our memory models differ significantly since Spec# is type-safe while C is not. Second, their executables simulate the verification semantics as defined by the verifier rather than the concrete semantics as defined by the language. Instead, our executables simulate the concrete semantics of C programs. As a result, their executables cannot guarantee the existence of an error even when an error is exercised since it might be ruled out by the concrete semantics. In contrast, in our approach an error is always consistent with the concrete semantics when the executable triggers it. A downside to our approach, however, is that our approach might fail to generate a successful executable counterexample when the verification semantics differ significantly from the concrete semantics. Third, their executables do not contain the original CUA but instead an abstraction of it where loops are modeled with loop invariants and methods with contracts.

Csallner and Smaragdakis [14] propose CnC (Check 'n' crash), a tool that uses counterexamples identified by ESC/Java [15], to create concrete test-cases (set of program inputs) that exercise the identified violation. Test-cases are then fed to the testing tool JCrasher [13]. When ESC/Java identifies a violation, CnC turns the counterexample into a set of constraints, which are solved to yield a program that exercises the violation. CnC is able to produce programs that contain numeric, reference, and array values. As in our framework, the executables produced by CnC simulate the concrete semantics of the underlying language (Java for CnC). Apart from using different memory models, the main distinction is that CnC aims at generating test-cases, while we focus on generating mocks that synthesize the external environment of the program. Test-cases are, in general, harder to produce because of the difficulty of ensuring that library calls produce the outputs needed to exercise the error. Instead, we try to generate the coarsest mocks for those library calls that can still exercise the error. Therefore, we believe our methodology can scale better for larger applications.

Recently, Beyer et al. [4] proposed an approach similar to Rocha et al. [29]. Given a counterexample in the SV-COMP [2] *witness automaton* format, original source is instrumented by assigning values from the counterexample. The approach is supported by CPAChecker [5] and Ultimate Automizer [24]. Similar to Rocha et al. [29], they do not deal with externally allocated pointers.

Test Case Generation. Dynamic Model Checking (DMC) adapts Model Checking to perform testing. One of the earliest DMC tools is VeriSoft [16] which has been very successful at finding bugs in concurrent software. The tool provides a simulator that can replay the counterexample but it does not generate executables. Test-case generation tools such as Java PathFinder [32], DART [18], EXE [8], CUTE [30], Klee [7], SAGE [19], and PEX [31] generate test-cases that can produce high coverage and/or trigger shallow bugs based on dynamic symbolic execution (DSE). The Yogi project [21,1,20] combines SMC with testing to improve scalability of the verification process. They compute both over- and under-approximations of the program semantics so that they can both prove absence of bugs and finding errors in a scalable way. Yogi tools have been integrated in the Microsoft's Static Driver Verifier. Christakis and Godefroid [9] combine SAGE [19] and MicroX [17] to prove memory safety of the ANI Windows Image

Parser. SAGE starts from a random test case and performs DSE while computing procedure summaries. MicroX computes sets of inputs and outputs for ANI functions without any provided information while allocating memory on-the-fly for each uninitialized memory address. They model precisely the concrete semantics of the program: "symbolic execution of an individual path has *perfect precision*: path constraint generation and solving is then *sound* and *complete*".

These tools model the concrete semantics of the program and allocate memory on-the-fly while we deliberately allow the SMC to use abstract semantics or even be unsound. By doing so, the verification process can scale. The challenge for us is to synthesize an environment that can exercise the error in the presence of uninitialized memory, while for these tools, the process of lifting an error execution to a test case is relatively simpler.

Guided Symbolic Execution. Hicks et al. [27] propose two heuristics to guide symbolic execution (SE) to reach a particular location. The first uses a distance metric to guide SE while the second uses the callgraph to run SE in a forward manner while climbing up through the call chain. Christakis et al. [11] introduce a program instrumentation to express which parts of the program have been verified by a static analysis tool, and under which assumptions. They use PEX [31] to exercise only those unverified parts. The same authors [10] instrument the code of a static analysis tool to check for all known unsound cases and provide a detailed evaluation about it. In all these cases, symbolic execution is guided in an intelligent manner to reach certain locations of interest. However, none of these techniques focus on dealing with memory.

7 Conclusion

We presented a new framework to generate mock environments for the Code Under Analysis (CUA). A mock environment can be seen as actual binary code that implements the external functions that are referenced by the CUA so that the CUA execution mirrors the counterexample identified by the Model Checker. We believe that having executable counterexamples is essential for software engineers to adopt Model Checking technology since they would be able to use their existing toolchain. Moreover, we described formally the concrete semantics of executable counterexample based on a simple extension to the standard operational semantics for C programs. This significantly differs from the textual-based counterexample representation used by SV-COMP tools. Finally, we have implemented an instance of the framework in SeaHorn, and tested it on benchmarks from SV-COMP 2018. Although the initial results are promising, more work remains to be done, especially to handle counterexamples with more complicated memory structures.

Acknowledgments

Authors would like to thank Natarajan Shankar for his invaluable comments to improve the quality of this paper.

References

1. N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, pages 3–14, 2008.
2. D. Beyer. Software verification with validation of results - (report on SV-COMP 2017). In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, pages 331–349, 2017.
3. D. Beyer, A. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, pages 326–335, 2004.
4. D. Beyer, M. Dangl, T. Lemberger, and M. Tautschnig. Tests from witnesses - execution-based validation of verification results. In *Tests and Proofs - 12th International Conference, TAP 2018, Held as Part of STAF 2018, Toulouse, France, June 27-29, 2018, Proceedings*, pages 3–23, 2018.
5. D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 184–190, 2011.
6. D. Beyer and T. Lemberger. Software verification: Testing vs. model checking - A comparative evaluation of the state of the art. In *Hardware and Software: Verification and Testing - 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings*, pages 99–114, 2017.
7. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224, 2008.
8. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2):10:1–10:38, 2008.
9. M. Christakis and P. Godefroid. Proving memory safety of the ANI windows image parser using compositional exhaustive testing. In *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings*, pages 373–392, 2015.
10. M. Christakis, P. Müller, and V. Wüstholz. An experimental evaluation of deliberate unsoundness in a static program analyzer. In *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings*, pages 336–354, 2015.
11. M. Christakis, P. Müller, and V. Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 144–155, 2016.
12. L. C. Cordeiro, B. Fischer, and J. Marques-Silva. Smt-based bounded model checking for embedded ANSI-C software. *IEEE Trans. Software Eng.*, 38(4):957–974, 2012.
13. C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robustness tester for java. *Softw., Pract. Exper.*, 34(11):1025–1050, 2004.

14. C. Csallner and Y. Smaragdakis. Check 'n' crash. In *Proceedings of the 27th international conference on Software engineering - ICSE '05*, page 422, New York, New York, USA, 2005. ACM Press.
15. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 234–245, 2002.
16. P. Godefroid. Verisoft: A tool for the automatic analysis of concurrent reactive software. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, pages 476–479, 1997.
17. P. Godefroid. Micro execution. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 539–549, 2014.
18. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223, 2005.
19. P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*, 2008.
20. P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 43–56, 2010.
21. B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: a new algorithm for property checking. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*, pages 117–127, 2006.
22. A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn verification framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 343–361, 2015.
23. A. Gurfinkel and J. A. Navas. A context-sensitive memory model for verification of C/C++ programs. In *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*, pages 148–168, 2017.
24. M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling, and A. Podelski. Ultimate automizer with smtinterpol - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 641–643, 2013.
25. B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 672–681, 2013.
26. LDV: Linux Driver Verification. `http://linuxtesting.org/ldv`.
27. K. Ma, Y. P. Khoo, J. S. Foster, and M. Hicks. Directed symbolic execution. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, pages 95–111, 2011.
28. P. Müller and J. N. Ruskiewicz. Using Debuggers to Understand Failed Verification Attempts. pages 73–87. Springer, Berlin, Heidelberg, 2011.

29. H. Rocha, R. Barreto, L. Cordeiro, and A. D. Neto. Understanding Programming Bugs in ANSI-C Software Using Bounded Model Checking Counter-Examples. pages 128–142. Springer, Berlin, Heidelberg, 2012.
30. K. Sen and G. Agha. CUTE and jcute: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, pages 419–423, 2006.
31. N. Tillmann and J. de Halleux. Pex-white box test generation for .net. In *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, pages 134–153, 2008.
32. W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with java pathfinder. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004*, pages 97–107, 2004.