Executable Counterexamples in Software Model Checking

J. Gennari[1] and A. Gurfinkel[2] and T. Kahsai[3] and
J. A. Navas[4] and E. J. Schwartz[1]

Presenter: Natarajan Shankar[4]

[1]Carnegie Mellon University
[2]University of Waterloo
[3]Amazon Web Services
[4]SRI International

VSTTE'18
July 18, 2018

Problem

- A distinguishing feature of Model-Checking is to produce a counterexample (cex) when a property is violated
- A cex is a trace through the system that shows how system gets to an error state from the initial states
- Software Model Checkers (SMC) often generate cex's as a set of assignments from logical variables to values
- In this work: how to show a SMC cex to developers?
- Most approaches use text format containing line numbers and variable values which can be understood for visualizers that relate them with the program
    - SLAM Verification Project
    - Linux Driver Verification Project
    - SV-COMP

Our solution: Executable Counterexamples

An executable cex triggers the buggy execution witnessed by the SMC

1. Generate code stubs for the environment with which the Code Under Analysis (CUA) interacts: libc, memcpy, malloc, OS system calls, user input, socket, file, etc
2. Generate an executable after linking the stubs with the CUA

- Key benefit: developer can use her traditional debugging tools such as gdb, `valgrind`, etc.
- Challenges:
    1. scalability: naive symbolic or concolic execution do not scale
    2. memory: counterexamples often dereference external memory
    3. precision: fully ignoring external memory is not often precise to replay the error

Test cases vs Executable Counterexamples

- A test case is an executable that determines whether the CUA satisfies a property or not
  - If property is violated, a test case is a proof of the existence of the error
- An executable cex is also an executable that synthesizes an environment for the CUA that is sufficient to trigger the error witnessed by the SMC
- An executable cex does not guarantee the existence of the error because it might not consider all the system assumptions
  - Human help is still needed to confirm the existence of the error
- However, executable cex's are easier to generate than test cases

```
x = input();
if (hash(0x1234) == x) __VERIFIER_error();
```

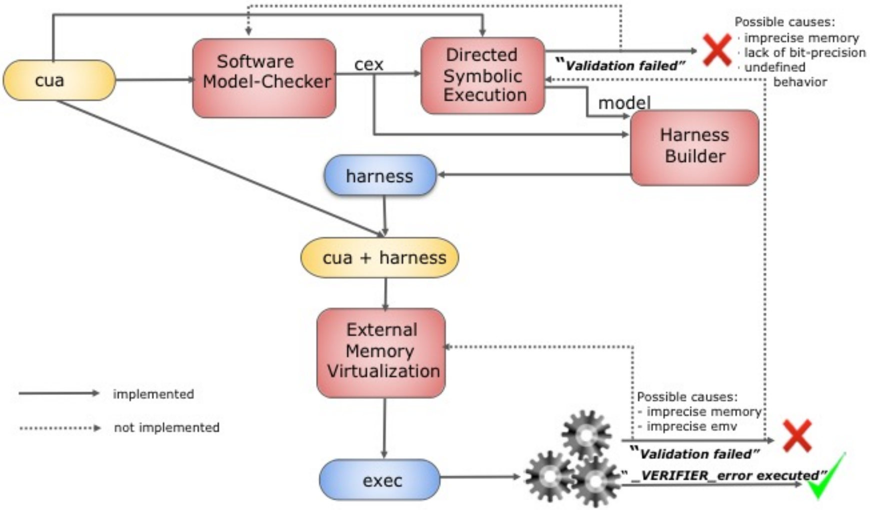
Example: read/write a field of a nondet pointer

```
struct st { int x; int y; struct st *next; };

extern struct st*  nd_st(void);
int main(int argc, char**argv) {
   struct st *p;
   p = nd_st();
   if (p > 0) {
      p->y = 43;
      if (p->x == 42)
        if (p->y == 43)
           __VERIFIER_error();
   }
   return 0;
}
```

- `nd_st()` returns non-deterministically a pointer to a new memory region
- The external memory region is both modified and read

Proposed Framework



Software Model-Checker (SMC)

IN: CUA + property
OUT: generate a cex in the form of a trace if property is violated

- Property violated if `__VERIFIER_error()` is executed
- A trace is, in its most general form, a Control-Flow Graph representation of the CUA where cut-point vertices are annotated with the number of times they are executed in the cex
- A trace can contain all blocks from the CUA
- A trace can be also a transformed/optimized version of the CUA
- SMC can over-approximate the concrete semantics or be unsound:
  - presence of undefined-behavior
  - unsound and/or too imprecise memory modeling
  - lack of bit-precise semantics of integer operations
  - . . .

Directed Symbolic Execution (DirSE)

IN: CUA + property + SMC trace
OUT: more precise cex wrt to the concrete semantics if success or abort otherwise

- DirSE aims at proving `__VERIFIER_error()` is still reachable but modeling more precisely the concrete semantics
- DirSE implemented as an SMT-based BMC problem
- A sound and more precise memory modeling:
  - `malloc` yields a pointer to a fresh allocated memory area disjoint from previously allocated regions
  - memory addresses are 4- or 8-byte aligned
  - assume program is memory safe until the first error occurs:
    - `malloc` always succeeds
    - assume all dereferenced pointers are in-bounds
- Bit-precise modeling of integer operations
- More details of the concrete semantics can be considered at the expense of increasing solving time

Harness Builder (HB)

Internalize all external functions by creating stubs for them

IN: Detailed cex produced by DirSE

OUT: code stubs for each external call and instrumented CUA

Sample from Linux Device Verification (LDV) project

```
extern int nondet_int(void);
extern void* ldv_ptr(void);
int main(...) {
 void *p = ldv_ptr();
 if (p <= (long) 2012)
   if (nondet_int() > 456)
    __VERIFIER_error();
}
```

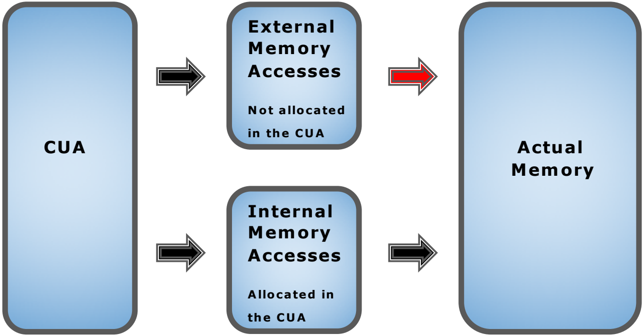
```
int nondet_int() {
  static int x=0;
  switch(x++) {
    case 0: return 457;
    case 1: ...
    default: return 0; }}
```

```
void* ldv_ptr() {
  static int x=0;
  switch(x++) {
    case 0:
    { uintptr_t p = 2011;
      return (void*) p; }
    case 1: ...
    default: return nullptr; }
```

Generating stubs for Linux Device Drivers is challenging

- Use of absolute addresses (e.g., 2012)
  - We believe address 2012 is added by the LDV team as part of the kernel modeling
  - Real code is likely to have other absolute addresses
- External functions can allocate new memory
- Generated stubs can have addresses for which no memory has been allocated in the CUA
- The HB instruments the CUA with memory read/store hooks that can control access to memory

External Memory Virtualization



Problem: map external memory accesses to actual memory

We have implemented two versions to deal with external accesses:

1. Ignore stores and return default value for loads
2. Allocate memory for external memory that is read or written

Demo

- https://www.youtube.com/watch?v=3Mx2WKFbLus
- https://www.youtube.com/watch?v=ct1X6pmnqk0&t=10s

Experiments

- We implemented DirSE, HB and EMV in SeaHorn and used `Spacer` as the model-checker
- We selected all the 356 false instances from `Systems`, `DeviceDrivers`, and `ReachSafety` categories of SV-COMP'18
- SMC solved 144, failed in 18, and ran out of resources in 194 (timeout=5m, memory limit=4GB)
- DirSE discarded 3 instances due to mismatch in bit-precise reasoning between SMC and DirSE
- We used a simple version of EMV: ignore stores and return default values for reads
- Counterexamples were validated (i.e., `__VERIFIER_error` was executed) in 24 cases (from 141)

Experimental results for validated counterexamples

Program	SMC		DirSE		HB+EMV	Harness Exec
	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_2a-drivers--staging--media--go7007--go7007-loader`	0.82	5	0.24	35	0.44	0.0
`205_9a_linux-3.16-rc1_9a-drivers--net--ppp_synctty`	44.32	6	3.46	61	0.71	0.0
`205_9a_linux-3.16-rc1_9a-drivers--net--wan--hdlc_ppp`	195.22	5	57.41	52	0.66	0.0
`43_2a_linux-3.16-rc1_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_1a-drivers--md--md-cluster`	5.84	5	12.0	23	0.68	0.0

Related Work: Executable Counterexamples from SMC

- `EZProofC` [RBCN12] and Beyer [BDLT18] replace in CUA all assignments with values from the SMC cex's:
    - they do not focus on dereferences of pointers allocated by external functions
    - unclear how to extract executables in presence of aggressive compiler optimizations
- Muller [MR11] generate C# executable cex's from Spec# programs:
    - Spec# does not have direct pointer manipulation
    - executables simulate the verification semantics of the verifier rather than the concrete semantics of the language
- CnC (Check 'n' crash) [CS05] produces test cases from cex's identified by ESC/Java

More Related Work

- Dynamic SMC (e.g., VeriSoft) and test-case generation tools such as JPF, DART, EXE, CUTE, Klee, SAGE, PEX, etc
    - they focus on producing high coverage and/or trigger shallow bugs based on dynamic symbolic execution and model checking
    - they model the concrete semantics of the program and allocate memory on-the-fly
    - we allow the SMC to use abstract semantics or even be unsound so that the process of finding deep errors can scale
- Guided symbolic execution: Directed Symbolic Execution [MKFH11] and Christakis [CMW16]
    - use of static analysis and/or heuristics to guide forward symbolic execution
    - they do not deal with memory

Conclusions

- We believe that executable counterexamples are essential for software engineers to adopt Model-Checking technology
- Executable counterexamples implement stubs for external functions that are linked to the CUA
- We have proposed a new framework and provided an implementation to generate executable counterexamples from C programs
- The framework allows the model checker to over-approximate the concrete semantics or to be unsound
- Initial results are promising but more work needs to be done, specially with complicated memory structures:

WIP implementation

New EMV that allocates actual memory for external memory regions

References

- `[CS05]`: Check 'n' Crash. Csallner and Smaragdakis in ICSE'05
- `[MR11]`: Using Debuggers to Understand Failed Verification Attempts. Müller and Ruskiewicz in FM'11
- `[MKFH11]`: Directed Symbolic Execution. Ma, Khoo, Foster, and Hicks in SAS'11
- `[RBCN12]`: Understanding Programing Bugs in ANSI-C Software using Bounded Model Checking Counter-Examples. Rocha, Barreto, Cordeiro, and Neto in IFM'12
- `[CMW16]`: Guiding dynamic symbolic execution toward unverified program executions. Christakis, Müller, and Wüstholz in ICSE'16
- `[BDLT18]`: Tests from witnesses - execution-based validation of verification results. Beyer, Dangl, Lemberger, and Tautschnig in TAP'18