FEDERAL UNIVERSITY OF AMAZONAS
INSTITUTE OF COMPUTING
GRADUATE PROGRAM IN COMPUTER SCIENCE

MEMORY MANAGEMENT TEST-CASE GENERATION OF C PROGRAMS USING BOUNDED MODEL CHECKING

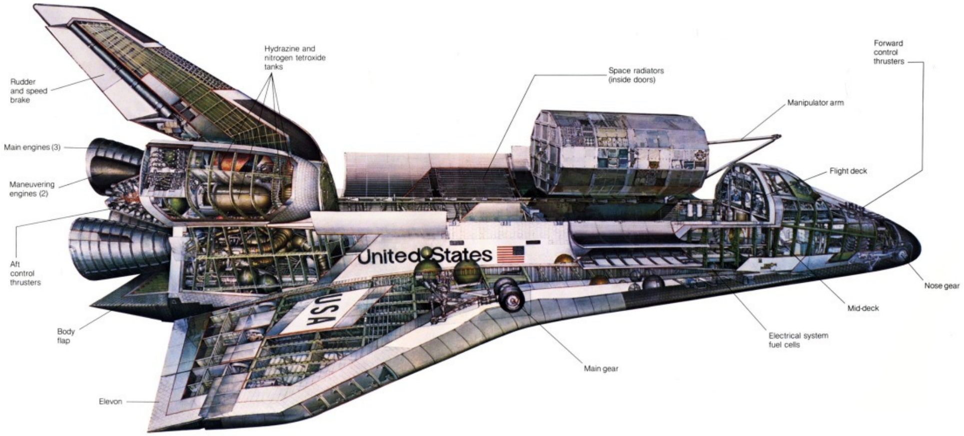
Herbert Rocha, Raimundo Barreto,
and Lucas Cordeiro

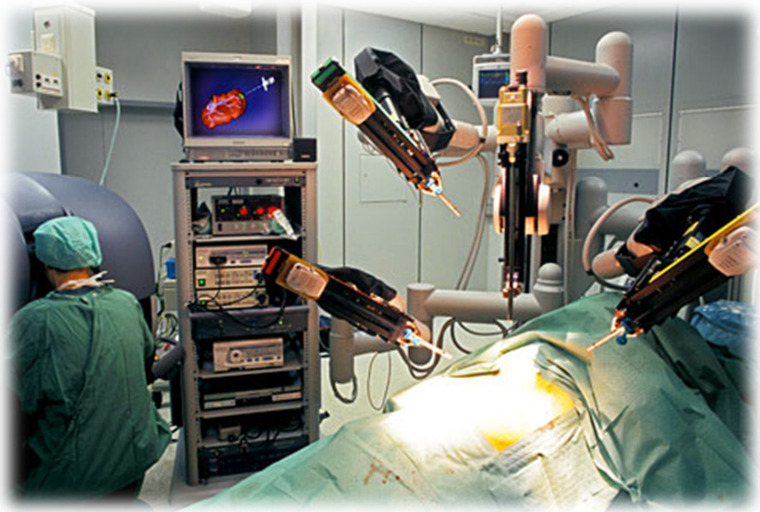
SEFM'2015

Agenda

1. Introduction
2. Background
3. Proposed Method
4. Experimental Evaluation
5. Conclusions and Future Work

Software Applications





Verification and Testing Software

In **software testing:**

- ✓ a significant human effort is required to generate effective test cases
- ✓ subtle bugs are difficult to detect

In **software model checking**:

- ✓ limited scalability to large software
- ✓ missing tool-supported integration into the development process

Verification and Testing Software

In **software testing:**

- ✓ a significant human effort is required to generate effective test cases
- ✓ subtle bugs are difficult to detect

In **software model checking**:

- ✓ limited scalability to large software
- ✓ missing tool-supported integration into the development process

The integration aims to alleviate the weaknesses from those strategies

What do you need to check?

- ✓ Analyzing **memory management** is an important task to avoid **unexpected behavior of the program**

- ✓ Pointer safety violation results in an invalid address
  - Produce an **incorrect result** of the program and not necessarily a crash

- ✓ Memory leaks have a negative impact in other application running on the same system
  - they typically **remain unobserved until** they consume a large portion of the memory

And what are we proposing? **The Map2Check Method**

- ✓ Map2Check generates automatically:
  - **memory management test cases** for structural unit tests for C programs
  - assertions from **safety properties** generated by **BMC tools**

- ✓ Map2Check aims to improve the unit testing environment, adopting features from (bounded) model checkers

- ✓ Map2Check adopts **source code instrumentation to:**
  - monitor the program's executions
  - validate assertions with **safety properties**

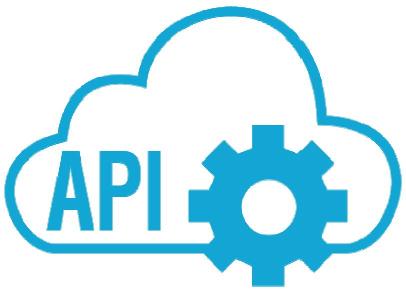
And what are we proposing? **The Map2Check Method**

Map2Check method **checks** the program **out of the BMC tools flow**

- ✓ It is based on dynamic analysis and assertion verification
- ✓ The assertions contain a set of specifications
- ✓ The **BMC is adopted as verification condition** (VC) generator that translates a program fragment and its correctness property into logical formula

The motivation of this work - **Map2Check**

- ✓ Aims to check for properties related to **pointer safety, memory leaks, and invalid free**
- ✓ Provides trace of memory addresses, in case of property violation
- ✓ Support the **integration between testing and verification** in an environment, where a software engineer can extend the analysis of the program through **APIs** and include new BMC and unit testing tools



#include <map2check.h>

Agenda

1. Introduction
2. **Background**
3. Proposed Method
4. Experimental Evaluation
5. Conclusions and Future Work

Efficient SMT-Based Bounded Model Checking - ESBMC

ESBMC is a bounded model checker for embedded ANSI-C software based on SMT (Satisfiability Modulo Theories) solvers, which allows:

- ✓ Out-of-bounds array indexing;
- ✓ Division by zero;
- ✓ Pointers safety
- ✓ Dynamic memory allocation;
- ✓ Data races;
- ✓ Deadlocks;
- ✓ Underflow e Overflow;

Safety Property

- ✓ Informally, a property in linear-time specifies the allowable (or desired) behavior of a system

- ✓ In this study, we use ESBMC VCs generator to check for memory safety as follows:
  - checking for safety pointers - **SAME OBJECT**
  - if a pointer is NULL or invalid object - **INVALID POINTER**
  - VCs for dynamic memory allocation - **IS DYNAMIC OBJECT**
  - if the argument to any free, or dereferencing operation is still a valid object - **VALID OBJECT**

Software Testing

- ✓ A **test case** consists of a test data analysis associated with an expected result of the software specification

- ✓ **Unit tests** are typically written based on a set of **test cases** to ensure that the program meets its design and behaves as expected

- ✓ We create **unit tests to analyze the software specification** together with their test data

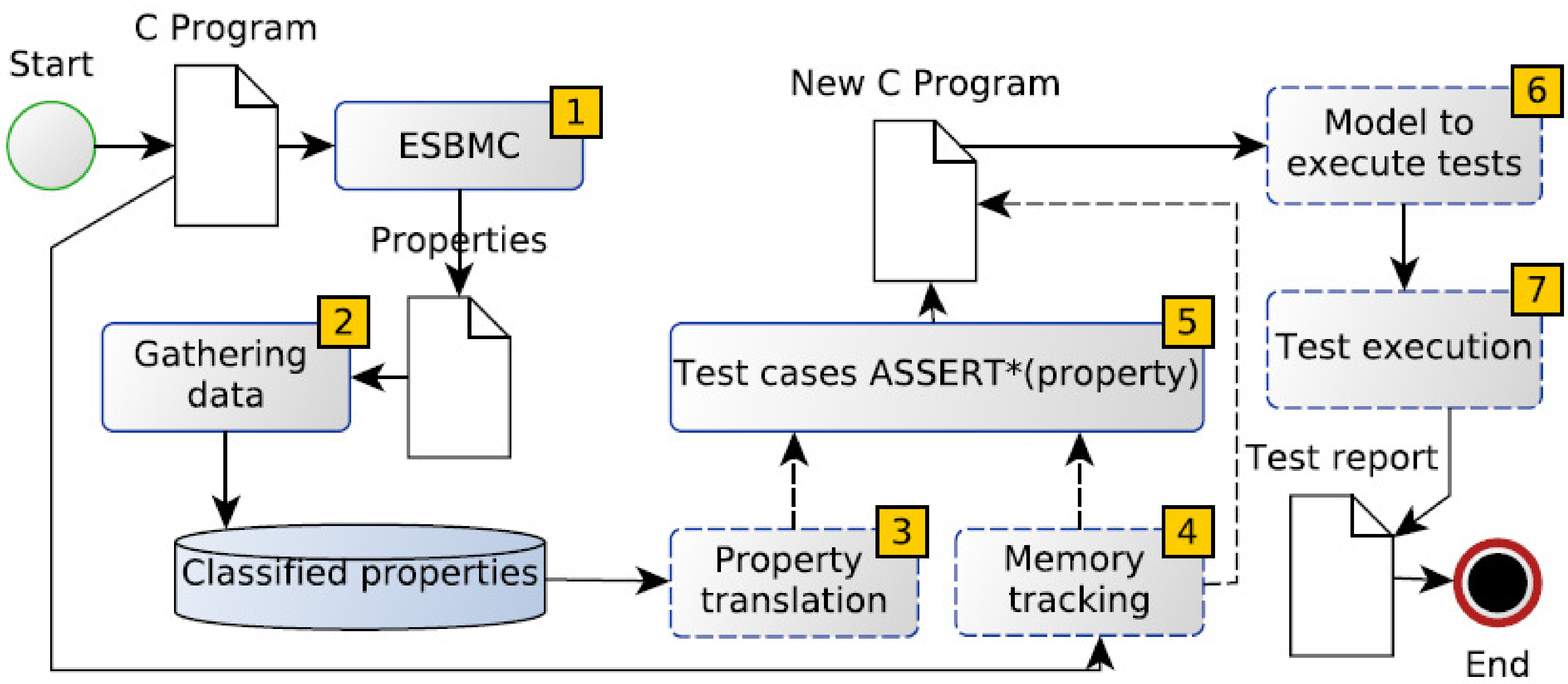
- ✓ We adopt the **CUnit framework** to develop unit tests

Available at: http://cunit.sourceforge.net

Agenda

1. Introduction
2. Background
3. **Proposed Method**
4. Experimental Evaluation
5. Conclusions and Future Work

Memory Management Test Case Generation for C Programs - Map2Check



Map2Check tool2 is available at https://sites.google.com/site/map2check/

Memory Management Test Case Generation for C Programs - Map2Check

```c
int *a, *b;
int n;

#define BLOCK_SIZE 128

void foo (){ ... }

int main ()
{
  n = BLOCK_SIZE;
  a = malloc (n * sizeof(*a));
  b = malloc (n * sizeof(*b));
  *b++ = 0;
  foo ();
  if (b[-1])
  { /* invalid free (b was iterated) */
  free(a); free(b); }
  else
  { free(a); free(b); } /* ditto */

  return 0;
}
```

960521 – 1_false-valid-free.c

SV-COMP 2014: 55.6% of the tools in the *MemorySafety* category **are not able to find** the property violation

Step 1: Identification of safety properties

```
$ esbmc --64 --no-library --show-claims
960521-1_false-valid-free.c
file 960521-1_false-valid-free.c: Parsing
Converting
Type-checking 960521-1_false-valid-free
Generating GOTO Program
Pointer Analysis
Adding Pointer Checks
Claim 1:
file 960521-1_false-valid-free.c line 12 function foo
dereference failure: dynamic object lower bound
!(POINTER_OFFSET(a) + i < 0) || !(IS_DYNAMIC_OBJECT(a))
```

Step 1: Identification of safety properties

```
$ esbmc --64 --no-library --show-claims
960521-1_false-valid-free.c
file 960521-1_false-valid-free.c: Parsing
Converting
Type-checking 960521-1_false-valid-free
Generating GOTO Program
Pointer Analysis
Adding Pointer Checks
Claim 1:
file 960521-1_false-valid-free.c line 12 function foo
dereference failure: dynamic object lower bound
!(POINTER_OFFSET(a) + i < 0) || !(IS_DYNAMIC_OBJECT(a))
```

Claims generated automatically by ESBMC do not necessarily correspond to errors

Step 2: Extract information from safety properties

Claims	Comments	Line	Property
Claim 1	dereference failure: dynamic object lower bound	12	!(POINTER_OFFSET(a) + i < 0) || !(IS_DYNAMIC_OBJECT(a))
Claim 2	dereference failure: dynamic object upper bound	12	!(POINTER_OFFSET(a) + i >= DYNAMIC_SIZE(a)) || !(IS_DYNAMIC_OBJECT(a))
Claim 3	dereference failure: dynamic object lower bound	14	!(POINTER_OFFSET(b) + i < 0) || !(IS_DYNAMIC_OBJECT(b))
Claim 4	File sum_array line 14 function main array `a' upper bound	14	!(POINTER_OFFSET(b) + i >= DYNAMIC_SIZE(b)) || !(IS_DYNAMIC_OBJECT(b))
...	...	...	...

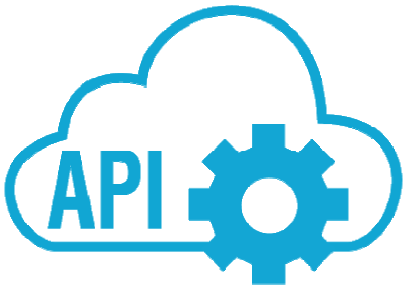
Step 3: Translation of safety properties

Translate claims provided by ESBMC to assertions into the C program:

- ✓ **INVALID-POINTER.**

$$INVALID - POINTER(i + pat) \ \boldsymbol{to}$$

IS _VALID_POINTER_MF (*LIST_LOG*, (*void∗*)&(*i+pat*), (*void∗*)(*intptr_t*)(*i+pat*))



#include <map2check.h>

Map2Check provides a library to the C program, which offers support to execute the functions generated by the translator.

Step 4: Memory tracking

Consists of two phases:

1) **identify and track variables** in the analyzed source code, as well as, the variable operations and assignments

2) **instrument the source code** with specific functions for **monitoring the memory addresses** and the addresses pointing by these variables according to the program execution

Step 4: Memory tracking

```
int *a, *b;
int n;

#define BLOCK_SIZE 128

void foo (){ ...

int main ()
{
  n = BLOCK_SIZE;
  a = malloc (n * sizeof(*a));
  b = malloc (n * sizeof(*b));
  *b++ = 0;
  foo ();
```

Phase 1: identify and track variables

Input: Abstract Syntax Tree (AST)
Output: Variables Tracking (Map)

Analyzing the program scope

foreach *node IN the AST* **do**
  if *type(node) == FuncDef* **then**
    compound_func = get the sub tree from node
    foreach *subNo FROM compound_func == Decl* **do** getDataFromVar(*subNo, 0*) ;
  end
  else if *type(node) == Decl* **then** getDataFromVar(*node, 1*) ;
end
Function getDataFromVar(*node, enableGlobalSearch*)

Step 4: Memory tracking

```
int *a, *b;
int n;

#define BLOCK_SIZE 128

void foo (){ ... }

int main ()
{
  n = BLOCK_SIZE;
  a = malloc (n * sizeof(*a));
  b = malloc (n * sizeof(*b));
  *b++ = 0;
  foo ();
  if (b[-1])
  { /* invalid free (b was iterated) */
  free(a); free(b); }
  else
  { free(a); free(b); } /* ditto */

  return 0;
}
```

Tracking of the variables

Pointer variable assignments

Step 4: Memory tracking

Phase 2: Instrumentation of the source code

- ✓ **mark_map_MF**. This function trackes the memory addresses (**`LIST_LOG`**) of the variables according to the program execution;
- ✓ **IS_VALID_DYN_OBJ_MF**. This function identifies if a dynamic object is valid;
- ✓ **INVALID_FREE**. This function identifies whether a given dynamic object can be released/deallocated from the memory properly;
- ✓ **CHECK_MEMORY_LEAK**. Identifies if, in the end of the program, some allocated memory is not released.

Step 4: Memory tracking

Tracking memory execution

```
3.  int *a, *b;
4.  int n;
5.
6.  #define BLOCK_SIZE 128
7.
8.  void foo
16.
17. int
18. {
19.   n
20.   a
21.   b
22.   *
23.   f
24.   if (  [-1])
25.   {
26.   f
27.   else
28.   { free(a); free(b); } /* ditto */
29.
30.   return 0;
31. }
```

Invalid free

Line	Address	Points to	Escope	Is Dynamic	Is Free
28	0x601050	0xb44034	global	0	1
28	0x601060	0xb44010	global	0	1
...	...	...	...	...	...
10	0x7fff39f18a2c	(nil)	foo	0	0
22	0x601050	0xb44034	global	0	0
21	0x601050	0xb44030	global	1	0
...	...	...	...	...	...

variable b was iterated

Step 5: Code instrumentation with assertions

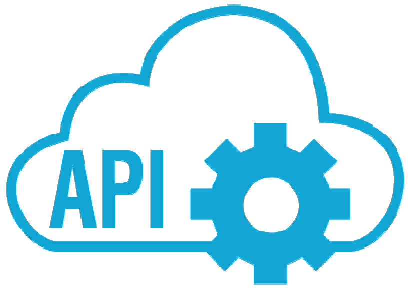
```
...

int main ()
{
  n = BLOCK_SIZE;
  a = malloc (n * sizeof(*a));
  b = malloc (n * sizeof(*b));
  *b++ = 0;
  foo ();
  if (b[-1])
  {
   ...
  }
  else
  {
    ASSERT(INVALID_FREE(LIST_LOG, (void *)(intptr_t)(a), 28));
    free(a);
ASSERT(INVALID_FREE(LIST_LOG, (void *)(intptr_t)(b), 28));
free(b);
  }
  return 0;
}
```

Step 6: Implementation of the tests

Model using C assertions

debugging



#include <map2check.h>

Model for CUnit

options/statements for unit testing

Unit Testing Framework

Step 6: Implementation of the tests

- #includes for CUnit
- #includes for Map2Check library
- #includes from analyzed C code
- The setup CUnit functions
- - Functions
  - - Test cases
  - - Global variables
  - - main = testClaims
- - New function main for Cunit
  - - It calls the setup to CUnit

```c
#include "CUnit/Basic.h"
#include "check_memory_safety_Map2Check.h"
#include <stdlib.h>
int init_suite1(void){...}
int clean_suite1(void){...}
int *a, *b;
int n;

#define BLOCK_SIZE 128

void foo ()
{ ... }

int testClaims ()
{
  ...
  foo ();
  if (b[-1])
  { /* invalid free (b was iterated) */
  free(a); free(b); }
  else
  { free(a); free(b); } /* ditto */

  return 0;
}

int main(){
  CU_pSuite pSuite = NULL;
  pSuite = CU_add_suite("check_code", init_suite1, clean_suite1);
  if(NULL==pSuite){...}
  if(CUE_SUCCESS != CU_initialize_registry()) return CU_get_error();
  if(NULL==CU_add_test(pSuite, "testClaims", testClaims))){...}
  CU_basic_set_mode(CU_BRM_VERBOSE);
  CU_basic_run_tests(); CU_cleanup_resgistry();
  return CU_get_error();
}
```

Step 7: Execution of the tests

```
VIOLATED PROPERTY
  Type     : Invalid FREE
  Location: In the line {28}
  Last Use: In the line {22}

FAILED
    1. mf_960521-1_false-valid-free.c:108
INVALID_FREE(LIST_LOG, (void *)(intptr_t)b,28)

Run Summary:     Type  Total    Ran Passed Failed Inactive
               suites      1      1    n/a      0        0
                tests      1      1      0      1        0
              asserts    516    516    515      1      n/a

Elapsed time =    1.880 seconds
```

Agenda

1. Introduction
2. Background
3. Proposed Method
4. **Experimental Evaluation**
5. Conclusions and Future Work

Planning and Designing the Experiments

Goal: Analyzing the ability of Map2Check to **generate and verify** test cases related to **memory management.**

- ✓ The experiments are conducted on an Intel Core i7-2670QMCPU, 2.20GHz, 32GB RAM com Linux OS
- ✓ The time limit to the verification is 15 min

Disponível em https://github.com/hbgit/Map2Check

Planning and Designing the Experiments

- ✓ We consider 61 ANSI-C programs from the *MemorySafety* category of the SV-COMP'14 benchmark

- ✓ Comparison to the tools:
  - Valgrind's Memcheck (Nethercote e Seward, 2007)
  - CBMC (Clarke et al., 2004)
  - LLBMC (Merz et al., 2012)
  - CPAChecker (Beyer e Keremoglu, 2011)
  - Predator (Dudka et al., 2014)
  - ESBMC (Cordeiro et al., 2012).

Experiment's Execution and Results Analysis

> 76%

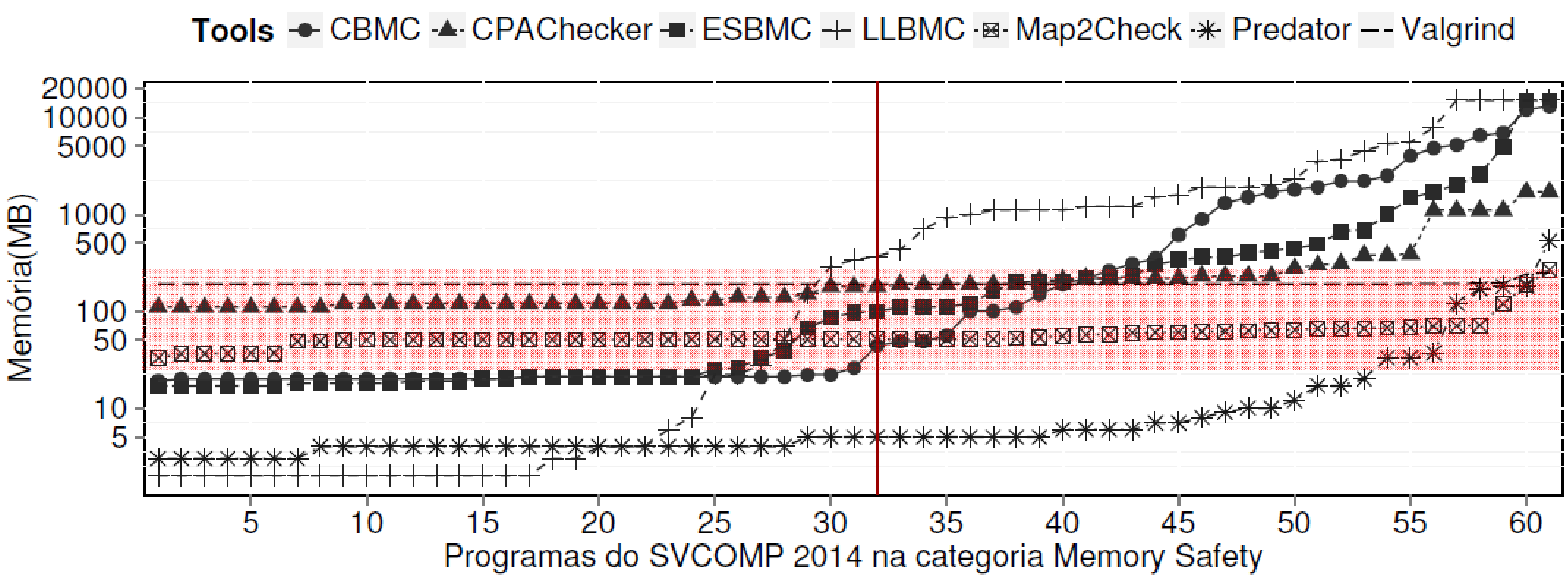
95.72% 95.08% 93.44%

Tool	CPAChecker	Map2Check	Valgrind	CBMC	Predator	LLBMC	ESBMC
Correct Results	59	58	57	46	43	31	7
False Negatives	0	0	0	8	0	0	0
False Positives	0	0	0	2	12	0	36
Unknown and TO	2	3	4	5	6	30	18
Time (min)	23.33	190.98	151.57	200	76.66	416.66	139.06

Experiment's Execution and Results Analysis

Memory consumed by the tools in the programs

- ✓ Map2Check is the **2nd tool** that consumes less memory
- ✓ Map2Check in **95%** of the programs has consumed about 50 MB



Experiment's Execution and Results Analysis

The runtime verification:

- ✓ Map2Check is **54.16%** faster than LLBMC and **4.5%** than CBMC

- ✓ Map2Check does not identify more correct results only, but also generates less Unknown and TO than CPAChecker

- ✓ Map2Check time: the concrete execution of the nondeterministic programs
  - The function `__VERIFIER_nondet_int()` in loop structures
  - Map2Check depends on a **random function** to determine the **halting condition of a loop**

Experiment's Execution and Results Analysis

Analyzing Map2Check in the context of the SVCOMP'14 in the *MemorySafety* category.

The scores could be ranked with negative points

Scores:

1st place: CPAChecker = 95 e Map2Check = 95

2th place: LLBMC = 38

3th place: Predator = 14

Experiment's Execution and Results Analysis

We had participated in **SV-COMP 2015** with **Map2Check tool** in the *MemorySafety* category

- ✓ Updates in SV-COMP:
  - In SV-COMP 2014 the total file **was 61** and in SV-COMP 2015 **was 205**
  - ✓ The scores were updated to penalize incorrect results

- ✓ Map2Check **won the 6th** from 9 tools (number of correct programs **was 165 from 205**)
  - Forester (Holik et al., 2015)
  - Seahorn (Kahsai et al., 2015)
  - CBMC (Clarke et al., 2004a)

BRNO UNIVERSITY OF TECHNOLOGY

Software Engineering Institute

Carnegie Mellon

UNIVERSITY OF OXFORD

Agenda

1. Introduction
2. Background
3. Proposed Method
4. Experimental Evaluation
5. **Conclusions and Future Work**

Conclusions and Future Work

- ✓ We presented a method to:
  - **integrate unit testing with model checkers**, focusing on memory management defects
  - disseminate the application of formal methods and **helping developers not very familiar** with this subject to verify large C programs
- ✓ Map2Check can be adopted as a **complementary technique** for the verification performed by **BMC tools**
  - Mainly when BMC tools cannot, usually **because of time-out**; or when there are **false negative or false positive**

Conclusions and Future Work

- ✓ The experimental results have shown to be very effective
- ✓ The Map2Check method has detected at least as many memory management defects as the state-of-the-art tools

For future work

- ✓ To improve the verification runtime and precision of Map2Check:
  - adopting program invariants
  - static verification based on abstract domain
- ✓ Adopting a witness checker

Questions ?

Thank you for your attention!

herberthb12@gmail.com