Memory Management Test-Case Generation of C Programs using Bounded Model Checking

Herbert Rocha, Raimundo Barreto, and Lucas Cordeiro

Federal University of Amazonas
{herberthb12,lucasccordeiro}@gmail.com
{rbarreto}@icomp.ufam.edu.br

Abstract. We describe a novel method to automatically generate and verify memory management test cases for unit tests, which are based on assertions extracted from safety properties typically generated by bounded model checking (BMC) tools. In particular, the proposed method checks for properties related to pointer safety, memory leaks, and invalid deallocation. To investigate our method's effectiveness, we developed a tool called Map2Check that adopts the ESBMC model checker and the CUnit testing framework. Additionally, Map2Check provides an integration of BMC tools with unit testing frameworks, which helps developers not very familiar with formal methods to verify large C programs. We use Map2Check to perform an empirical evaluation over publicly available benchmarks and compare the results to recognized tools, e.g., Valgrind's Memcheck, CBMC, LLBMC, CPAChecker, Predator, and ESBMC. Experimental results show that our proposed method detects at least as many memory management defects as existing tools; and it does not report any false positive and negative. We compared Map2Check with tools on the Competition on Software Verification 2014 (SVCOMP), in the *MemorySafety* category. Map2Check would have the same score than the 1st place and it would win the 2st place when ranking the evaluated tools on memory consumption.

1 Introduction

Nowadays, software applications need to be developed quickly, mainly due to the short time-to-market. However, programmers make mistakes, e.g., writing a given system requirement incorrectly. In this sense, the application of verification and testing are indispensable techniques to the development of high-quality software. Integrating formal program verification and testing has been adopted as a widely recognized solution to improve the software quality. This integration aims to alleviate the weaknesses from these strategies [8, 13, 15], e.g., in software testing a significant human effort is required to generate effective test cases and as a result, subtle bugs are difficult to detect by testing and that can cause significant overhead after the target software is deployed. According to Kebrt and Sery [17], the adoption of software model checking technologies in the industrial development process is still very slow. This is caused by two main reasons: limited scalability to large software and missing tool-supported integration into the development process.

In the last few years, we have observed a trend towards the application of formal verification techniques to the implementation level. Bounded Model Checking (BMC) is going into this direction since it has been successfully applied to reason about low-level ANSI-C/C++ programs [5, 9, 19]. The main challenge in model checking is how to deal with both the state space explosion problem and the lacking of integration with other test environments more familiar to practitioners [6]. One possible solution to tackle these problems is to explore features already provided by the model checking community (e.g., identification of safety properties) for test case generation. According to Baier and Katoen [1], safety properties are often characterized as "*nothing bad should happen*". In particular, the violation of a safety property can be detected by monitoring the run-time system execution.

The verification of memory management is an important task to avoid unexpected behavior of the programs, e.g., pointer safety violation results in an invalid address, which might produce an incorrect result of the program and not necessarily a crash; a memory leak does not immediately produce an easily visible symptom, i.e., a crash or the output of a wrong value. However, memory leaks typically remain unobserved until they consume a large portion of the memory available in a system; and these might lead to a negative impact in other application running on the same system [7]. Due to the serious consequences and common occurrence of memory management errors, there are still open research fields to improve the error detection.

In this study, we describe a novel method to automatically integrate formal verification techniques with testing environments. The proposed method generates automatically memory management test cases for structural unit tests, which are based on assertions from safety properties generated by BMC tools. As a consequence, our proposed method aims to improve the unit testing environment, adopting features from (bounded) model checkers. Additionally, the proposed method adopts source code instrumentation to monitor and gather data from the program's executions, aiming to verify the generated test cases, and thus, detecting violations of safety properties from the analyzed program. Note that this method checks the program out of the BMC tools flow, given that they do not handle well pointers and pointers arithmetic [2].

The BMC is adopted as verification condition (VC) generator that translates a program fragment and its correctness property into logical formula. The VC has the property that if it is valid, then the program fragment satisfies its correctness property [12]. In this study, we use the Efficient SMT-Based Context-Bounded Model Checker (ESBMC) [9], which derives VCs using two recursive functions that compute assumptions or constraints (i.e., variable assignments) and properties (i.e., safety conditions and user-defined assertions). Both functions accumulate the control flow predicates to each program point and use that to guard both the constraints and the properties, so that they properly reflect the program's semantics. It is worth noting that ESBMC does not require the user to annotate the programs with pre/post-conditions to generate the VCs, but allows the user to state additional properties using assert-statements.

The proposed method is a complementary technique for the verification performed by state-of-the-art BMC tools. Our method aims to check for properties related to pointer safety, memory leaks, and invalid free. Additionally, the proposed method provides trace of memory addresses, which has already been executed at the current point of the program, in case of property violation. This trace of memory guides developers directly to the locations, where the memory management errors are identified. Most existing initiatives have been proposed to verify the memory management of C programs ([7, 19, 22]). However, those initiatives do not 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. The proposed method also provides an API library of functions, which helps developers to extend the tests generated by our proposed method, e.g., by using functions from API to write new assertions (i.e., test cases) in a specific point of the analyzed program to validate pointers operations. In this study, we adopted the C programming language since it is the standard language to implement different kinds of software, including critical software [21].

To evaluate the effectiveness of our proposed method, we adopted ESBMC [9] and the CUnit testing framework [18], and we implemented the method in a prototype tool called Map2Check. Note that any other BMC and unit testing tool could be used together with our approach. We performed an empirical evaluation on publicly available benchmarks from the Competition on Software Verification (SV-COMP 2014) [2], in particular the *MemorySafety* category. We also compare our proposed method with other tools, such as: Valgrind's Memcheck [22], CBMC [5], LLBMC [19], CPAChecker [4], Predator [11], and ESBMC [9]. The experimental results of the proposed method have shown to be effective, detecting 95.08% of the correct results, i.e., if a property satisfy its specification or is violated. Our method, in comparison to the results of the SV-COMP 2014, would have the same score than the 1st place.

2 Preliminaries

This section presents the ESBMC, discuss about safety properties, and software testing using CUnit.

2.1 Efficient SMT-Based Bounded Model Checking (ESBMC)

ESBMC is a Context-Bounded Model Checker based on Satisfiability Modulo Theories (SMT) solvers, which is used for ANSI-C/C++ programs [9]. ESBMC verifies single- and multi-threaded programs and checks for properties related to arithmetic under- and overflow, division by zero, out-of-bounds index, pointer safety, deadlocks, and data races. In ESBMC, the verification process is completely automated and does not require the user to annotate programs with pre- or post-conditions. ESBMC converts ANSI-C/C++ programs into equivalent *GOTO-programs*, which simplify statement representations (*e.g.*, replacement of *while* by *if* and *goto* statements). The *GOTO-program* is symbolically executed by the *GOTO-symex*, which generates a single static assignment form that is later converted into a first-order logic formula and then checked by an SMT solver. If a property violation is found, a counterexample is provided by ESBMC, which assigns values to the program variables for reproducing the respective error.

2.2 Safety Properties

Informally, a property in linear time specifies the allowable (or desired) behavior of a system [1]. If a system fails to satisfy a safety property, then there exists a finite execution that exhibits this failure. Consequently, checking the correctness of the system related to the safety properties is a means to validate the system's behavior.

Definition 1 *(Safety Property) Given a transitions system $TS = (S, S_0, E)$, let $B \subset S$ be a set of bad states such that $S_0 \cap B = \emptyset$, we may say that TS is safe in relation to B, denoted by $TS \models \boldsymbol{AG}\ \neg B$ if there is no path in the transition system from the initial state S_0 up to bad state B. Otherwise we say TS is not safety, denoted by $TS \not\models \boldsymbol{AG}\neg B$ [1].*

ESBMC is able to automatically infer safety properties from the C programming language such as arithmetic under- and overflow, memory safety, array bounds, atomicity and order violations, deadlock and data race. In this study, however, we use ESBMC VCs generator to check for memory safety as follows: VCs to check for safety pointers, i.e., checking if the pointer does reference to a correct object (represented by *SAME_OBJECT*) and also checks if a pointer is `NULL` or invalid object (represented by *INVALID_POINTER*); and VCs for dynamic memory allocation, ESBMC checks if the arguments to *malloc, free* functions, or dereferencing operations are a dynamic object (represented by *IS_DYNAMIC_OBJECT*) and if the argument to any *free*, or dereferencing operation is still a valid object (*VALID_OBJECT*).

2.3 Software Testing with CUnit

Software testing is the process of executing a program to find faults [20]. A successful test is the one that can determine the test cases for which the program under test fails. 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. In this study, we create unit tests to analyze the software specification together with their test data. In particular, we adopt the CUnit framework to develop unit tests. CUnit allows software engineers to create unit tests in a more efficient way by favoring better organization and code reuse. CUnit supports full C and provides a set of assertions for testing logical conditions (e.g., CU_ASSERT_PTR_EQUAL for pointers). The success or failure of these assertions is tracked by the framework, and can be viewed when a test run is complete. The typical sequence of steps for using CUnit is as follows[1]: (1) Write functions for tests (and `init/cleanup`, if necessary); (2) Initialize the test registry - `CU_initialize_registry()`; (3) Add suites to the test registry - `CU_add_suite()`; (4) Add tests to the suites - `CU_add_test()`; (5) Run tests using an appropriate interface, e.g. `CU_console_run_tests`; (6) Cleanup the test registry - `CU_cleanup_registry`.

3 Map2Check Method

In this section, we present the Map2Check method for memory management test case generation for C programs. Map2Check is an improvement from the FORTES (*FORmal unit TESt generation*) method [23], which explores the safety properties generated by BMC tools to create test cases. However, FORTES does not generate memory management test case. The Map2Check tool[2] is available to freely download under GPL license. Figure 1 shows an overview of our proposed method, where the boxes and the arrows with dashed lines represent, respectively, the components updated or inserted by Map2Check and its execution flow.

To explain the main steps of our proposed method, we use the program 960521 – 1_false-valid-free.c from the SV-COMP'14; this program belongs to the category *MemorySafety* [2] (see Figure 2). We use this program as a running example since 55.6% of the tools in the *MemorySafety* category are not able to find the property violation.

[1] http://cunit.sourceforge.net/doc/introduction.html#usage

[2] https://sites.google.com/site/map2check/

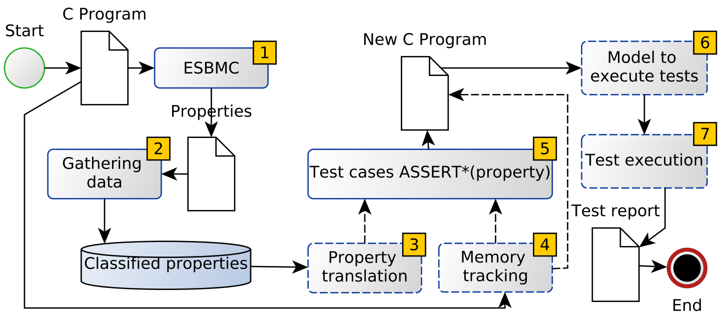


Fig. 1: Flow structure of the proposed method.

```c
#include <stdlib.h>

int *a, *b;
int n;

#define BLOCK_SIZE 128

void foo ()
{
  int i;
  for (i = 0; i < n; i++)
    a[i] = -1;
  for (i = 0; i < BLOCK_SIZE - 1; i++)
    b[i] = -1;
}

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;
}
```

Fig. 2: C program 960521 – 1_false-valid-free.c from SVCOMP’14

3.1 Step 1: Identification of safety properties

Map2Check adopts ESBMC for identification of safety properties. ESBMC receives a C program as an input parameter and an option `--show-claims`, which shows all safety properties that ESBMC automatically generates from the original C program. In the ESBMC context, a `claim` is the same as a safety property. Claims generated automatically by ESBMC do not necessarily correspond to errors, but they are just potential flaws in the program. One needs to determine, through further analysis, if some claim actually corresponds to an error. Figure 3 shows an example of a claim automatically generated. In this example, claim 1 states a potential lower bound of the dynamic object "a" in the line 12 of the function foo. All claims are stored to be used in the next steps.

```
$ 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))
```

Fig. 3: Step 1 - Identification of safety properties.

3.2 Step 2: Extract information from safety properties

The second step checks the result produced in step 1 as follows: *(i)* identification of the claim; *(ii)* comments about the claim (e.g., dereference failure: dynamic object upper bound); *(iii)* the code line number where the claim occurred (e.g., Line = 26); and *(iv)* the property identified by that claim (e.g., !(POINTER_OFFSET((void *)b) < 0) || !(IS_DYNAMIC_OBJECT(b))). The proposed method then classifies data provided in the claims via regular expressions to find all relevant information.

3.3 Step 3: Translation of safety properties

This step aims to translate claims provided by ESBMC to assertions into the C program; these claims have specific functions that are only executed by ESBMC (e.g., INVALID-POINTER). This function checks if a pointer is NULL or an invalid object. Thus, the proposed method translates the claims to check them without ESBMC intervention. The translator translates each ESBMC function using a grammar parse for the claims. The identification of each claim, and its respective components, are passed as input to the translator, which applies the appropriate rules (e.g., rewrite the function return according to source code) to convert the claims into functions that can be executed by the C program that is being analyzed, without ESBMC intervention. Aiming the function execution, Map2Check provides a library to the C program that provides the support to execute the functions generated by the translator.

3.4 Step 4: Memory tracking

The memory tracking aims to extend the FORTES verification in the sense of pointer safety. The identification of pointers and invalid objects allow us to analyze invalid free and memory leak. The memory tracking consists of two phases: (1) identify and track variables in the analyzed source code, as well as, the variable operations and assignments, and (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.

Algorithm 1 shows how to identify and track variables. The runtime complexity of this algorithm is $O(n^2)$, where n is the number of the nodes in an `Abstract Syntax Tree (AST)` of the analyzed program. We use the following terms to explain our algorithm: *Object*, which means that the analyzed variable is a pointer or dynamically allocated variable; *Simple Variable*, which are variables that are not pointers; and *Mapping*, which means that the variable is being identified and their characteristics and operations (declaration and assignments) have been extracted and saved.

```
   Input: Abstract Syntax Tree (AST)
   Output: The map of the variables
 1 begin
 2     compound_func = Not specified
 3     foreach node IN the AST do
 4         if type(node) == FuncDef then
 5             compound_func = get the sub tree from node
 6             foreach subNo FROM compound_func == Decl do getDataFromVar(subNo, 0) ;
 7         end
 8         else if type(node) == Decl then getDataFromVar(node, 1) ;
 9     end
10     Function getDataFromVar(node, enableGlobalSearch)
11         if type(node) is a pointer then
12             if node has an Assigment then Mapping the data from getNodeData (node) ;
13             if enableGlobalSearch then
14                 searchVarAssigInAllFunctions (node)
15             else
16                 searchAssigIn (compound_func, node)
17             end
18         else
19             Mapping the data from getNodeData (node)
20         end
21     end
22 end
```

Algorithm 1: Gather the variables to memory track

The input of the Algorithm 1 is an `Abstract Syntax Tree (AST)`, which is generated from the analyzed C program. Map2Check adopts Pycparser[3] that parses the C code into an AST. The algorithm runs each node in the AST and for each node, it is identified the local scope (i.e., the program functions (line 4)) and global scope (line 8).

Line 8 of the algorithm starts the mapping of the program global variables. This mapping identifies if the AST node refers to a declaration of a variable, which is indicated by the type `Decl` in the AST. Line 8 calls the function `getDataFromVar` which takes two parameters, `node` and `enableGlobalSearch`. The parameter `node` is the current AST node that contains the declaration of a variable to be mapped and `enableGlobalSearch` is a Boolean value. In this particular case, **True** indicates that a

[3] Available at https://github.com/eliben/pycparser

search is performed in all functions of the program to track variable assignments identified in the node. In the same sense, Line 4 of the Algorithm 1 identifies if the current AST node refers to a program function to perform a mapping of the AST node refers to a declaration of a variable, but in this case only in that function.

The function getDataFromVar from the algorithm (line 10) consists of identifying whether or not the variable being mapped is a pointer and then it extracts the variable data. If the variable is not a pointer, it just executes the mapping of the variable. The mapping is performed by gathering and listing data provided by the function getNodeData (in line 19). The function getNodeData receives as input the node being analyzed and it obtains the following data: (i) the line number in the source code where the variable is located; (ii) the variable name; (iii) the scope/function name where the variable is located; and (iv) if the object is dynamic.

In line 11 if the variable is a pointer, then it performs the object mapping (using the function getNodeData) only if the statement identified in the analyzed node also includes an assignment. In other case, the mapping is performed only after the first assignment. Thus, the method avoids mapping uninitialized pointers, which may contain garbage memory. Additionally, a search is performed to track pointer assignments (operations, allocation, and deallocation of memory) according to its scope (line 13). If the object is in the global scope, then a search is performed in all program functions (line 14); otherwise, the search is performed only in the scope where the object is located (line 16).

The second phase is to instrument the source code with functions that will monitor the memory addresses and the addresses pointed by the variables according to the program execution. For each line identified in the mapping (of the previous phase) for the analyzed program, the proposed method inserts, after the identified line, the function mark_map_MF, where this function receives as input the mapped data for that line. The function mark_map_MF manages a list (called of LIST_LOG) of variables, which contains: the memory address; the memory address that points to; the identifier of its scope; an identifier if it is dynamic; the identifier if was executed the *free* function; and the line number of the source code. The list LIST_LOG has the trace of memory addresses already executed at the current point of the program. In Map2Check, we developed a C library that contains specific functions, which allow the execution of the function mark_map_MF as well as the functions previously mentioned in Section 3.3.

The verification of the analyzed properties is performed by applying the functions from the Map2Check library, as shown in the following list. The functions in items 3 and 4 are generated as test case by Map2Check and are not provided from ESBMC claims, as well as, Map2Check provides test cases for union operation to check for dynamic memory address overwriting.

1. **IS_VALID_DYN_OBJ_MF**. This function identifies if a dynamic object is valid. In this case, the method searches in the list LIST_LOG by the memory address pointed to by the variable that is being traced. If the memory address is found, the method adopts these checks: (1) the method searches in the list to identify if the memory address pointed was previously traced; and (2) the method searches in the list by the attribute that identifies if the variable is still a dynamic object.
2. **IS_VALID_POINTER_MF**. This function searches in the list LIST_LOG only by the memory address pointed to by the analyzed variable to identify if the variable is pointing to a valid address. If the memory points to a dynamic object, then it verifies if it is a valid object using the function IS_VALID_DYN_OBJ_MF.

3. **INVALID_FREE**. This function identifies whether a given dynamic object can be released/deallocated from the memory properly, for instance, using the *free* function from the C programming language. The library calls the function `IS_VALID_DYN_OBJ_MF` to identify if the dynamic object is valid.
4. **CHECK_MEMORY_LEAK**. The function identifies if, in the end of the program, some allocated memory was not released. This function searches in the list `LIST_LOG` the memory addresses that are still dynamic, checking the attribute that identifies whether a given object is dynamic. If it is identified in that point of the program that there is some dynamic object, then the functions identify this as a memory leak.

Table 1 shows an example of the tracking memory execution of the analyzed program (see Figure 2). In this execution of the proposed method, we identified that the analyzed program has an invalid free in line 28. This happens because in line 22, the variable `b` was iterated, as shown in ID = 4 and Points to = 0xb44034 of the Table 1. Thus, the invalid free has been presented in line 28 and showing in ID = 260 of the table since the memory address that the pointer points to is not a valid request from a block of memory from the heap, as shown in ID = 4 and Is Dynamic = 0 in Table 1.

ID	Memory Address	Memory Address Points to	Scope	Is Dynamic	Is Free	Line Number
260	0x601050	0xb44034	global	0	1	28
259	0x601060	0xb44010	global	0	1	28
...	...	...	...	...	...	...
133	0xb44034	(nil)	global	0	0	14
...	...	...	...	...	...	...
6	0xb44010	(nil)	global	0	0	12
5	0x7fff39f18a2c	(nil)	foo	0	0	10
4	0x601050	0xb44034	global	0	0	22
3	0x601050	0xb44030	global	1	0	21
2	0x601060	0xb44010	global	1	0	20
1	0x601058	(nil)	global	0	0	4

Table 1: The result to apply the tracking memory in the analyzed program.

3.5 Step 5: Code instrumentation with assertions

This step aims to create test cases, based on assertions, which are included in the source code with their respective safety property/claim generated by ESBMC and also by Map2Check. This step adds an assertion to verify the safety property, which is identified in **Step** 2 (see Section 3.2) and **Step** 4 (see Section 3.4). This assertion can be a simple assertive provided by the C language or an assertion of a unit testing framework. This step identifies the source code line from each identified property, in order to add an assertion in a previous line, which is identified by the property in the source code being verified. For instance, in the program of Figure 2, the following assertion is added to line 28: `ASSERT(INVALID_FREE(LIST_LOG, (void *)(intptr_t)(b), 28))`.

3.6 Step 6: Implementation of the tests

This step applies the model to the analyzed program for tests execution. The method has two models: Using only C assertions, the method inserts the `include` to Map2Check library in the new instance of the analyzed program. This model is very simple and useful while debugging a program to check a property violation; and the model for CUnit, this model is useful when one needs more options/statements for unit testing. In this CUnit model, we apply a template provided by the method in the analyzed program that has the following items: (i) includes for CUnit and Map2Check library in the analyzed C program; (ii) the setup CUnit functions; (iii) functions that contain test cases, which will be tested; and (iv) the new function *main* that will be executed by CUnit.

The CUnit libraries are extracted from the template as well as the Map2Check library. The `includes` from the analyzed C program are copied from its original C code. The setup CUnit functions are used from the template. The proposed method renames the function *main* to *testClaims*, because the new function *main* and its content is taken from the template. This new function (*main* to CUnit) calls the setup CUnit functions and the function *testClaims* (old function *main*). The result is a new instance of the analyzed program, which is ready to be tested and executed by the CUnit framework. Note that our method can also be applied to other unit testing frameworks; however, one needs to create a template for code generation.

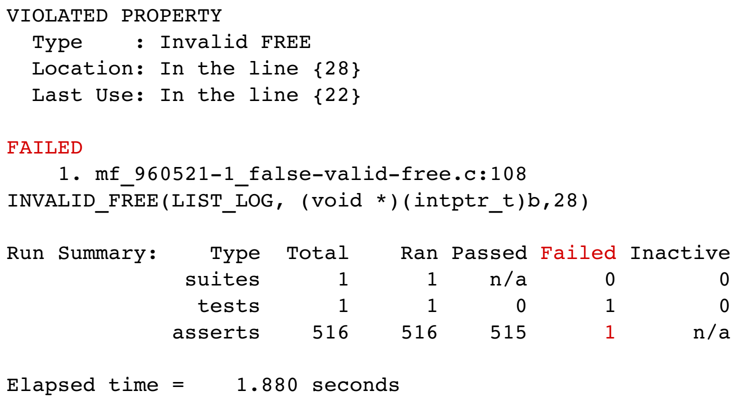
3.7 Step 7: Execution of the tests

In this last step, Map2Check provides two options: (1) executing the test cases using assertions from the C programming language or (2) executing the test cases using assertions from a unit test framework. To explain the result of this step, we adopt here the second option. Thus, CUnit runs the tests in the new program that has test cases generated from ESBMC and Map2Check safety properties, thus validating each assertion. Basically, the test cases are analyzed over the execution of the new instance of the analyzed program, where each test case generated by the proposed method can pass or fail. Each test failure is reported by the framework in the end of the new program execution. Figure 4 shows the result of the Map2Check.

It is worth noting that the test cases are analyzed over program execution, thus it is possible to improve the program coverage adopting different test inputs to the program. For instance, adopting the PathCrawler tool [25] that automatically generates test inputs for functions written in ANSI-C. PathCrawler is based on dynamic analysis and uses constraint logic programming to solve a (partial) path predicate and find test inputs.

4 Experimental Evaluation

This section describes the planning, design, execution, and the analysis of the results of an empirical study to evaluate the proposed method, when applied to the verification of standard ANSI-C benchmarks and, additionally, a comparison to the tools: Valgrind's Memcheck [22], CBMC [5], LLBMC [19], CPAChecker [4], Predator [11], and ESBMC [9]. The experiments are conducted on an Intel Core i7-2670QM CPU, 2.20GHz, 32GB RAM with Linux OS. The proposed method is implemented in a tool called Map2Check using ESBMC.



```
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
```

Fig. 4: The result of the use of Map2Check.

4.1 Planning and Designing the Experiments

This empirical evaluation checks the ability of Map2Check to generate and verify test cases related to memory management. We investigate the following research questions:

RQ1: Are the test cases generated by Map2Check enough to identify a given defect in the analyzed program?
RQ2: How was the ability (the execution of instrumented functions) of Map2Check to verify the test cases?
RQ3: How is the Map2Check's ability to detect memory management defects compared to existing tools?

To answer these three research questions, we consider 61 ANSI-C programs from the *MemorySafety* category of the SV-COMP'14 benchmark [2]. In this case, we only consider programs related to the memory safety category. In this category, the properties to be verified are: (i) **p_valid-free** - All memory deallocations are valid; (ii) **p_valid-deref** - All pointer dereferences are valid; and (iii) **p_valid-memtrack** - All allocated memory is tracked, i.e., pointed to or deallocated.

In SV-COMP benchmarks, some programs adopt specific functions, e.g., the *MemorySafety* category has the `__VERIFIER_nondet_int()` function that models nondeterministic integer values. In Map2Check, we implement a function to simulate the nondeterministic integer values; our implementation returns a random number (0 or 1) from an array according to the following distribution: 30% to 0 and 65% to 1. One could argue that this approach depends on luck to have a correct program coverage to validate the assertions. This could be true, but we adopt this simulation of non-determinism since in our preliminary tests, it was enough to detect 70% of the properties violations.

We conducted the evaluation as follows: (1) Application of Map2Check (see Section 3), adopting the model with only C assertions to identify the first property violation in the analyzed program. (2) Application of the Valgrind/MemCheck with the following options: `--leak-check = yes --undef-value-errors = yes`. (3) The results of the application of the tools: CBMC, LLBMC, CPAChecker, Predator, and ESBMC are taken literally from [2], because the options adopted to execute all tools in this experiment are the same and the hardware used is similar. It is worth noting that it is necessary to compile the program to run Valgrind/MemCheck; therefore we adopt the nondeterministic function implemented in the Map2Check library.

For the Map2Check and Valgrind/MemCheck, each program in the category is executed 3 times, because of the nondeterministic behaviour. It is important to note that from these 3 executions, we always consider the execution classified as FAILED (if any), i.e., an execution that the tool has identified a property violation.

4.2 Experiment's Execution and Results Analysis

After executing the benchmarks, we obtained the results shown in Table 2, where each row of this table means: (1) name of the tool (Tool); (2) total number of programs that satisfies the specification identified by the tool (Correct Results); (3) total number of programs that the tool identified an error for a program that fulfills the specification (False Negatives); (4) total number of programs that the tool did not identify the error (False Positives); (5) total number of programs that the tool failed to compute verification result, without resources, program crash or the tool exceeded runtime verification of 15 min (Unknown and TO); (6) the execution time in minutes of the verification for all programs in the category (Time).

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	23.33min	190.98min	151.57min	200min	76.66min	416.66min	139.06min

Table 2: Result of tools evaluation using SVCOMP'14 benchmark.

To answer research question **RQ3** (see Section 4.1), Table 2 shows that Map2Check has found 95.08% of correct results, while CPAChecker has found 95.72%, Valgrind has found 93.44%, and the other tools could detect only less than 76% of the correct results. Note that Map2Check did not generate any false positive and false negative results. Map2Check has generated 3 unknown and timeout results. We believe that this is, in part, because of the concrete execution of the program. In future, we plan to adopt a static verification based on abstract domain [24] to improve verification time.

With respect to research questions **RQ1** and **RQ2**, we can infer that Map2Check has generated and verified successful test cases. Taking into account **RQ1**, Map2Check was able to generate correct test cases to identify a given defect in the analyzed program and not generated incorrect assertions in the test cases that could result a false alarm in the test execution. We also identified for **RQ2** that the execution of the instrumented functions worked properly, since the instrumented functions supported the execution of the test cases without incorrect results.

The results presented in Table 2 shows that Map2Check can be adopted as a complementary technique for the verification performed by BMC tools. Map2Check can provide support for the program analysis, mainly when BMC tools cannot, usually because of time-out; or when there are false negative or false positive. If we compare the results of ESBMC to Map2Check, ESBMC identified 7 correct results while Map2Check identified 58 where, in this case, Map2Check may be seen as a complement to ESBMC.

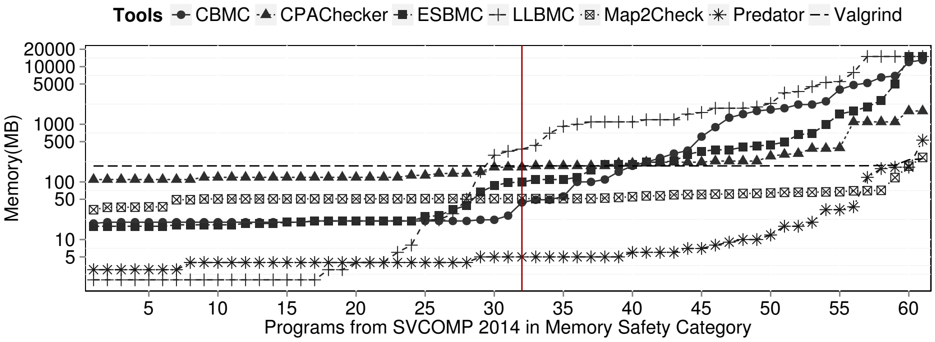


Fig. 5: Memory consumed by the tools in the programs.

In the same way, ESBMC had 18 Unknown and Time-out results, but Map2Check was able to analyze 15 of those programs without Unknown or Time-out results.

Analyzing the memory consumption by the tools in each program of the SVCOMP 2014 benchmark. We identified that Map2Check is the 2nd tool that consumes less memory (total of 3680.69 MB); the 1st is Predator tool (total of 1600 MB), as shown in Figure 5. Analyzing this figure, we identified that from 32th program (the vertical line in the Figure 5) there was an increase of the memory consumption to more than 50 MB from 5 of the 7 analyzed tools. However, Map2Check in 95% of the programs has consumed about 50 MB. Thus, Map2Check did not have considerable variation w.r.t memory consumption, which is different from other tools, e.g., LLBMC consumed more than 10.000 MB for specific programs.

Note that Map2Check consumes less memory than ESBMC since it adopts ESBMC only to generate the claims, which consumes about 20 MB. In 53% of the programs, Map2Check consumed less memory, except for Predator. However, Map2Check identified 25% more correct results than Predator. We believe that the Map2Check memory consumption can be improved because only for the test cases generation was used 78.98% (i.e., 2907.16 MB) of total memory. Therefore, optimizing the translation of claims would have significant impact in reducing the memory consumption.

We observe that the runtime verification of Map2Check was 54.16% faster than LLBMC and 4.5% than CBMC, as shown in Table 2. Note that the time to generate the claims is about 1s, which is included in Table 2. Importantly, even though the verification time of Map2Check was higher than the other tools. Map2Check only not identified less correct results, and generated less Unknown and TO than CPAChecker tool. We believe that Map2Check total verification time, in turns, could be explained by the concrete execution of the nondeterministic programs.

One could argue that concrete execution should be much faster than the symbolic execution performed by the tools adopted in this experiment. In part this could be explained by the strategy adopted to unwind loops and their respective loop exit condition, where benchmarks use the function `__VERIFIER_nondet_int()` in loop structures. The Map2Check implementation returns a random number (0 or 1) from an array according to the following distribution: 30% to 0 and 65% to 1. Thus, a BMC could complete the program verification faster than the Map2Check that depends on a random function to determine the stopping condition of a loop.

To analyze the evaluation of Map2Check in the context of the SVCOMP'14 [2] in the *MemorySafety* category, we need to take into account the same rules adopted in the SVCOMP'14. For instance, the scores that could be ranked with negative points, e.g., an incorrect TRUE is equal to −8 points. For more details, see Beyer [2]. Therefore, Map2Check could achieve the 1st place of the SVCOMP'14 in the *MemorySafety* category with a score of 95 points, where actually in the SVCOMP'14 the 1st place was CPAChecker with the same score of 95 points; 2th place was LLBMC with a score of 38 points; and 3th place was Predator with a score of 14 points.

Recently, we had participated of SV-COMP 2015 with Map2Check tool in the *MemorySafety* category (see the competition report in [3]). The main differences were: in SV-COMP 2014 the total file was 61 and in SV-COMP 2015 was 205; and the scores was updated to penalize incorrect results, which thus rules out testing and BMC tools. Map2Check won the 6th from 9 tools. Map2Check overcame tools as Forester [14], Seahorn [16] and CBMC [5]. Analyzing the Map2Check results in SV-COMP 2015, we identify that Map2Check is the 4nd tool that consume less time (total of 8.400s) and memory (total of 70.000 MB). Map2Check generated 0 false positives and 15 false negative. These incorrect answers produced by our tool in the competition are due to bugs in the implementation. Since the tool submission, we have fixed some bugs, and considerably improved the implementation. Taking into account only the correct results (the programs that satisfies the specification identified by the tool). Map2Check would win the 2nd place, where the total number of correct programs was 165 from 205; the total time of the verification was 2.100s; and the memory consumption was 9.100 MB.

These results, albeit preliminary in nature, strongly suggest that our method can be effective in generating and checking test cases of memory management for C programs. Additionally, Map2Check reports traces that guide developers to the locations where the memory management errors are. We thus argue that Map2Check integrates test and verification. The test is based on dynamic analysis and assertion verification. The assertions contain a set of specifications (for the validation of memory blocks). This verification is similar to the one performed by Delahaye et al. [10], where Pre-Post conditions based on formal program specification are translated into executable C code.

5 Conclusions and Future Work

In this study we presented a novel method to generate and verify automatically memory management test cases for structural unit tests, which are based on assertions from safety properties generated by BMC tools of C programs. The proposed method checks properties such as: pointer safety, memory leaks, and invalid free. The main purpose of this study is to integrate unit testing with model checkers, focusing on memory management defects; therefore, disseminating the application of formal methods and helping developers not very familiar with this subject to verify large C programs.

We also presented Map2Check, a prototype tool that implements our method. The experimental results have shown to be very effective. Map2Check has found 95.08% of correct results, while CPAChecker has found 95.72%, Valgrind has found 93.44%, and the other tools could detect only less than 76% of the correct results. For future work, we intend to improve the verification runtime and precision of the proposed method by adopting program invariants and static verification based on abstract domain [24].

References

1. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
2. Beyer, D.: Status report on software verification (competition summary SV-COMP 2014). In: TACAS. pp. 373–388,. Springer (2014)
3. Beyer, D.: Competition on Software Verification (SV-COMP) - Results of the Competition. Available at http://sv-comp.sosy-lab.org/2015/results/MemorySafety.table.html (2015)
4. Beyer, D., Keremoglu, M.E.: CPAchecker: A Tool for Configurable Software Verification. In: CAV. pp. 184–190. Springer-Verlag (2011)
5. Clarke, E., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: TACAS. pp. 168–176. Springer (2004)
6. Clarke, E.M.: 25 years of model checking. chap. The Birth of Model Checking, pp. 1–26. Springer-Verlag (2008)
7. Clause, J., Orso, A.: LEAKPOINT: pinpointing the causes of memory leaks. In: ICSE. pp. 515–524. ACM (2010)
8. Comar, C., Kanig, J., Moy, Y.: Integrating formal program verification with testing. In: ERTS (2012)
9. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-Based Bounded Model Checking for Embedded ANSI-C Software. In: TSE. pp. 957–974. IEEE (2012)
10. Delahaye, M., Kosmatov, N., Signoles, J.: Common Specification Language for Static and Dynamic Analysis of C Programs. In: SAC. pp. 1230–1235. ACM (2013)
11. Dudka, K., Peringer, P., Vojnar, T.: Predator: A shape analyzer based on symbolic memory graphs. In: TACAS. pp. 412–414. Springer Berlin Heidelberg (2014)
12. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: Generating compact verification conditions. In: POPL. pp. 193–205. ACM (2001)
13. Groce, A., Joshi, R.: Extending model checking with dynamic analysis. In: VMCAI. pp. 142–156. Springer (2008)
14. Holik, L., Hruska, M., Lengal, O., Rogalewicz, A., Simacek, J., Vojnar, T.: Forester. Available at http://www.fit.vutbr.cz/research/groups/verifit/tools/forester/ (2015)
15. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement. In: CAV. pp. 209–213. Springer (2008)
16. Kahsai, T., Gurfinkel, A., Navas, J.A.: SeaHorn - A software verification tool. Available at https://bitbucket.org/lememta/seahorn/wiki/Home (2015)
17. Kebrt, M., Sery, O.: Unitcheck: Unit testing and model checking combined. In: ATVA. pp. 97–103. Springer (2009)
18. Kumar, A.: CUnit. Available at: http://cunit.sourceforge.net/ (2014)
19. Merz, F., Falke, S., Sinz, C.: LLBMC: bounded model checking of C and C++; programs using a compiler IR. In: VSTTE. pp. 146–161. Springer-Verlag (2012)
20. Myers, G.J., Sandler, C.: The Art of Software Testing. John Wiley & Sons (2004)
21. Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S.: CETS: compiler enforced temporal safety for C. In: ISMM. pp. 31–40. ACM (2010)
22. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: PLDI. pp. 89–100. ACM (2007)
23. Rocha, H., Cordeiro, L., Barreto, R., Netto, J.: Exploiting Safety Properties in Bounded Model Checking for Test Cases Generation of C Programs. In: SAST. pp. 121–130. SBC (2010)
24. Ströder, T., Giesl, J., Brockschmidt, M., Frohn, F., Fuhs, C., Hensel, J., Schneider-Kamp, P.: Proving termination and memory safety for programs with pointer arithmetic. In: IJCAR. pp. 208–223. Springer (2014)
25. Williams, N., Marre, B., Mouy, P., Roger, M.: Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In: EDCC. pp. 281–292. Springer (2005)