C-SMC: A Hybrid Statistical Model Checking and Concrete Runtime Engine for Analyzing C Programs*

Antoine Chenoy, Fabien Duchene(✉), Thomas Given-Wilson, and Axel Legay

UCLouvain, Louvain-La-Neuve, Belgium
antoinechenoy@outlook.be
{fabien.duchene,thomas.given-wilson,axel.legay}@uclouvain.be

Abstract. Finding programming errors is one of the major challenges in software development. Formal methods such as model checking have become a popular approach to address this problem because of their guarantees about error status. However, one of the greatest challenges is to have correct information about complex internal details such as memory, registers, and system state. In this paper we describe the C-SMC tool and methodology developed to find programming errors in C programs by leveraging statistical model checking and runtime information. Our prototype shows that our approach can complement many existing software verification tools.

1 Introduction and Motivation

The advantages of formal methods over traditional approaches, such as testing, are formal methods' capability to provide guarantees about results. Another advantage is in using expressive temporal logic to elegantly express and verify complex temporal properties. Approaches such as *model checking* (MC) [13] are able to produce examples of errors (falsification of properties) or guarantee the absence of errors, albeit at the cost of building a model and checking every possible state.

Over past years, formal methods were restricted to the verification of system's abstractions such as transition systems that would directly be provided by the user. The situation has changed with the arrival of a series of new tools that allows us to verify C code (among others) directly. There are many tools for C code analysis that use a variety of approaches including: heuristics [1,2,8]; model checking [17, 18, 21, 25, 34] and variations [29, 32]; other formal methods [38, 39]; symbolic verification [19], and runtime analysis [11, 36]. All those tools either suffer from the state-space explosion problem or from difficulties in handling the memory model of the system under verification.

Advances on MC such as *statistical model checking* (SMC) [12,30,31] provide an efficient balance by resolving many complexity and state-space problems from MC, albeit at the cost of certainty [15, 24].

* This work has been partially supported by a Cisco grant.

SMC simulates multiple executions of a system and monitors them in regards to properties. Then, SMC uses statistical algorithms to extrapolate to the global system. While SMC can efficiently find errors and avoid complexity problems, the trade-off is that SMC provides only a statistical likelihood of the absence of errors and a confidence in this result, not a formal guarantee. So far, SMC has been applied to verify safety/liveness requirements of (stochastic) models of systems, but not security.

The application of SMC to verify specifications of high-level code is limited to very specific reachability properties and restricted fragments of languages [36]. Consequently the verification of security properties that depends on space and time memory failures have not yet been explored. Such failures are prominent in popular languages such as C.

This is because one great challenge for many formal methods is how to accurately model true possible values within the program and also the complex internal information related to the program being analyzed. The true possible values of a program execution can be difficult to determine due to abstract approaches having extremely large domains to consider (e.g. all 64-bit Integers) or complex runtime relationships between values that are very hard to track. Similarly, memory organization may extremely complex to accurately model in an abstract manner. The choices of the compiler or of the operating system may be opaque to a verification engine. This can lead to inaccuracies and hence to security vulnerabilities that are difficult to define or analyze.

In this paper we develop the *C Statistical Model Checker* (C-SMC) approach and tool that is able to address these challenges by combining the strengths of SMC with a runtime engine to connect the model with the real execution of the program. The SMC aspects allow for many executions of the program to be handled independently and their checking results to be effectively combined using statistics on the overall likelihood to satisfy properties. The runtime engine allows for the model and states to gain real information that is apparent from the execution without having to simulate and model the entire hardware and operating system's behaviour in the model. The advantage of the approach used by C-SMC is that formal results are obtained for understanding the coverage and limitations of the analysis, i.e. the advantages of SMC. Also by using a real runtime debugging engine (here GDB [5]) to inform the analysis, the true instantiations of memory and internal system hardware can be exploited to determine concrete values and verify complex temporal properties.

The implementation of C-SMC is achieved by combining aspects of both static and dynamic analysis. A static model of the program is constructed and given to an SMC engine (Plasma Lab [16,33]), with information about the source code added as tags in the model. Then when Plasma Lab is analyzing the model, a link is made from the checker in Plasma Lab to an instance of the program being executed (by GDB) so that Plasma Lab can have access to a dynamic execution with real information based on areas that are difficult to model (e.g. memory layout, registers, true variable values, etc.). To validate the effectiveness of C-SMC we compare to other well regarded tools that are dedicated to the same

challenge such as CBMC [29], Coverity [1], CPPCheck [2] and PVS Studio [8] Our results indicate that C-SMC performs extremely well at detecting errors, particularly those that rely upon runtime information and that it is able to detect bugs that no other evaluated tool can. These results also show that although the SMC and runtime approach comes at a cost in terms of execution time and resource consumption, the trade-off is the reduction of significant abstraction both in finding good executions and avoiding complexity problems. A VirtualBox [7] Virtual Machine (VM) containing C-SMC, its source code and all the examples used in this paper is available at `https://c-smc.csvl.eu/C-SMC.ova`.

2 Systems: From Specification to Verification

This paper offers a *Statistical Model Checking* (SMC) approach to verifying safety and security properties of complex C programs. The technique works by inferring statistic properties from the monitoring of successive executions of a program written in C source code.

SMC applies to stochastic systems, while execution of C programs are inherently sequential except for two sources. The choice of initial arguments that may depends on users; and the governance of process interleaving. These two sources are inherently non-deterministic to the program. This paper is restricted to non-concurrent systems. Consequently the only source of non-determinism is the choice of initial software parameters. In order to account for this source of non-determinism, this work assumes the existence of a stochastic oracle that generates initial parameters in a uniform manner. Here the *uniform distribution* is the distribution with the maximal entropy.

Here a *state* specifies all the current information related to a C program. In particular a state not only includes current values of program's variables, but also the value of others elements used by the program. This includes, e.g., the current program counter, memory (stack and heap), registers, etc. A given program has one (or more) entry points which are here defined to be the *initial state[s]*. Such states are generated by uniformly selecting an initial value for the input parameters. As usual, a C program moves from one state to another via *transitions* that represents program's instructions. An *execution trace* is a sequence of states generated from one initial state by following a sequence of transitions of the software. Observe that such transitions not only modify program's values, but also other information like the CPU flags or the instruction pointer.

From the above, a C program can entirely be defined by the set of all of its executions. Moreover, by definition of the stochastic oracle and the restriction to non concurrent systems, one can define a unique probability distribution on such set. Consequently, SMC applies to our setting.

2.1 Trace Execution Properties

Expressing properties formally is key in software verification. Based on definitions from above, the properties about a program can be defined in terms of

properties over states, over individual traces, and over sets of traces. We first focus on the first two types of properties. The last one, which corresponds to the verification of the whole program, will be handled in the next (sub-)section.

A propositional logic can define properties about each state individually. To be able to consider all the states of an execution trace, this propositional logic needs to be extended. One popular extension is *Linear Temporal Logic* LTL [37]. LTL allows us to make hypothesis of unbounded traces via temporal operators. As SMC restricts to finite trace executions, we consider a bounded version of LTL, where each temporal operator is bounded the number of states to which it applies.

Bounded Linear Temporal Logic (BLTL) is an enhancement of LTL that adds bounds expressed in step or time units. The syntax of BLTL is as follows and adapted from LTL:

$$\phi, \psi \quad ::= \quad p \mid \phi \vee \psi \mid \neg\phi \mid \phi\, U_{\leq t}\, \psi \mid X_{\leq t}\, \phi\ . \tag{1}$$

The p is propositional variables, disjunction $\phi \vee \psi$ and negation $\neg\phi$ are all as in LTL. The formula $X\phi$ is true if ϕ is true in the next state from the current state. The formula $\phi\, U_{\leq t}\, \psi$ is true if both: ψ becomes true before t in the sequence from the current state; and ϕ remains true in every state before the state where ψ becomes true. For a formal definition of BLTL semantics, see [41]. A BLTL formula is expressed with respect to a trace. It is also helpful to have conjunction ($\phi \wedge \psi$) and implication ($\phi \Rightarrow \psi$) that are defined in the usual manner. Similarly the *always* (G) and *eventually* (F) operators can be defined using the BLTL syntax above as follows. Eventually is defined as $F_{\leq t}\phi = \texttt{true}\, U_{\leq t}\, \phi$ and means that the formula ϕ should become true before t. Always is defined as $G_{\leq t}\phi = \neg F_{\leq t}\neg\phi$ and means that ϕ must always hold for the next t.

Let $w = s_0, s_1, ..., s_L, ...$ be an execution trace, and denote by $w^j = s_j, ..., s_L, ...$ the portion of the trace starting from j (included). The truthfulness of the formulas can be decided using the rules described in Table 1.

$w \models F_{\leq t}\phi$	iff $w \models$ true $U_{\leq t}\phi$
$w \models G_{\leq t}\phi$	iff $w \models \neg(F_{\leq t}\neg\phi)$
$w \models \phi U_{\leq t}\psi$	iff $\exists i, t_0 \leq t_i \leq t_0 + t$ and $w^i \models \psi$ and $\forall j, 0 \leq j < i, w^j \models \phi$
$w \models X_{\leq t}\phi$	iff $\exists i, i = max(j \mid t_0 \leq t_j \leq t_0 + t)$ and $w^i \models \phi$
$w \models X_{\leq}\phi$	iff $w^1 \models \phi$
$w \models \phi \vee \psi$	iff $w \models \phi$ or $w \models \psi$
$w \models \phi \wedge \psi$	iff $w \models \neg\phi \vee w \models \neg\psi$
$w \models \phi \Rightarrow \psi$	iff $w \models \neg\phi \vee \psi$
$w \models \neg\phi$	iff $w \not\models \varphi$
$w \models$ true	always
$w \models$ false	never

Table 1. BLTL rules

BLTL allows us to express reachability properties such as "the software should eventually reach a state where variable x is equal to 1". The logic can also be used

to express more elaborated causalities such as "always, if the software reaches a state where x is equal to 1, it will eventually reach a state where y is equal to 1". This expressive power allows us to express a wide range of safety and security properties. BLTL properties can be verified with monitoring procedures [26]. Such procedures, inspect successive states of an execution trace until it can decide whether the property is satisfied. Note that the BLTL is well suited to reason on system's model where states are abstracted by Boolean variables. As we are working with C code, we will assume that such variables can be replaced by Boolean predicates on states. Such predicates can express, e.g., Boolean properties on program's variables or register in a given state. These concepts are illustrated in Section 3.2 where the architecture of C-SMC is presented.

2.2 Probabilistic Verification

We now turn to properties defined over sets of trace executions, that is properties defined on the whole system. We are interested in solving the probabilistic BLTL problem, that is to compute the probability for the system to satisfy a BLTL property ϕ. Such probability being defined as the probability that a random trace of the system satisfies ϕ.

Statistical model checking [30] has been proposed as an efficient approach to solve such problem. SMC statistically measures the truthfulness of properties over a smaller number of traces. This is done by performing a fixed number of simulations of the system and using an algorithm to estimate the probability that the system satisfies the property. As the number of observed executions is finite, this answer comes together with a confidence interval [30].

There is a wide range of statistics algorithms that can be used to estimate the probability to satisfy a given property [30]. As the study of those algorithms is not the topic of this paper, we propose to work with the most simple one that is based on the Monte Carlo estimator. The estimator relies on the following proportion:

$$\bar{\gamma} = \frac{\sum_{i=1}^{N} \mathbf{1}(w_i \models \phi)}{N} \text{ where } \mathbf{1}(x) = \begin{cases} 1 & \text{if } x \text{ is true} \\ 0 & \text{otherwise.} \end{cases} \quad (2)$$

where N is the number of simulation being performed. Let γ be the true probability to satisfy ϕ and P be a probability evaluation. Let ϵ (precision) and δ (confidence)be small values. The Chernoff-Hoeffding bound [27] guarantees that $P(|\bar{\gamma} - \gamma| \geq \epsilon) \leq \delta$ is given by $N = \lceil \frac{ln2 - ln\delta}{2\epsilon^2} \rceil$. Consequently, by controlling the number of trace executions to be verified, the user entirely controls the preciseness of the $\bar{\gamma}$ estimator.

2.3 Implementation: Plasma Lab

In this section, we focus on tools that implement Statistical Model Checking algorithms. As seen in the previous section, SMC mainly depends on sub-parts that include the type of execution trace property that has to be monitored and

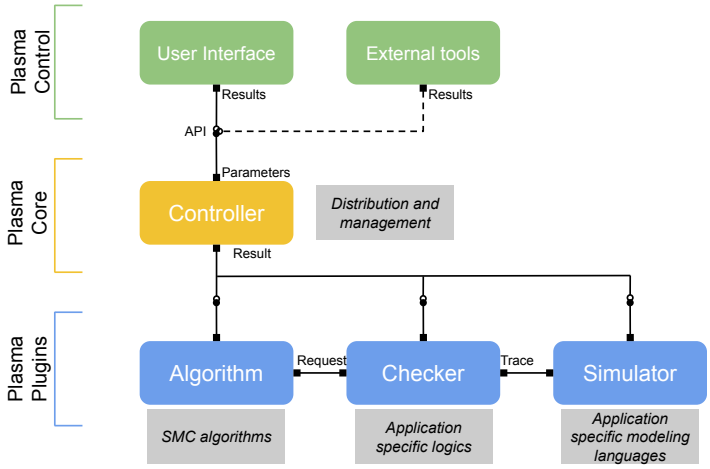

Fig. 1. Overview of the Plasma Lab architecture.

on the statistical algorithm used to compute the stochastic guarantee. Implicitly, SMC also depends on the type of system under verification and on the capacity to generate an arbitrary number of execution traces from this system.

In [40], the authors proposed `YMER`, a tool that can be used to verify BLTL properties of Markov Chains with an hypothesis testing procedure. That is, the tool decides between two hypothesis rather than computing an exact probability. In [23] Monte Carlo is applied to verify properties of Metric Temporal Logic over stochastic timed automata. Those tools have been shown to be very efficient on various problems. However, they are rather static in the sense that they do not exploit the intrinsic modularity of SMC. As an example, `YMER` does not permit replacing the hypothesis testing algorithm by a Monte Carlo one. None of those tools allows replacing classical BLTL with an algorithm that would consider debugger expression. On the top of this, none of those tools consider C code.

In [16,33], we introduced Plasma Lab a Statistical Model Checking (SMC) [30] tool that can provide the ability to create custom statistical model checkers. As shown in Fig. 1, Plasma Lab consists of three different layers. The first one is the *control layer*. It allows us to express various type of stochastic systems written in command-guarded language via a user interface. It also permits one to plug other system descriptions such as C code via an API for external control. The second one is the *core* that manages the model checking experiments, that is the interaction between all SMC components. The third one is the *plugins* that permits modularity and tool customization by introducing new SMC subparts.

Plasma Lab supports three types of plugins: Algorithm, Checker and Simulator. An *Algorithm* plugin manages the entire process and will begin by requesting new traces from the Checker. This is here that one choice between algorithms such as Monte Carlo or hypothesis testing. The user can also propose her own statistic algorithm. The *Checker* will then proceed to the monitoring of the prop-

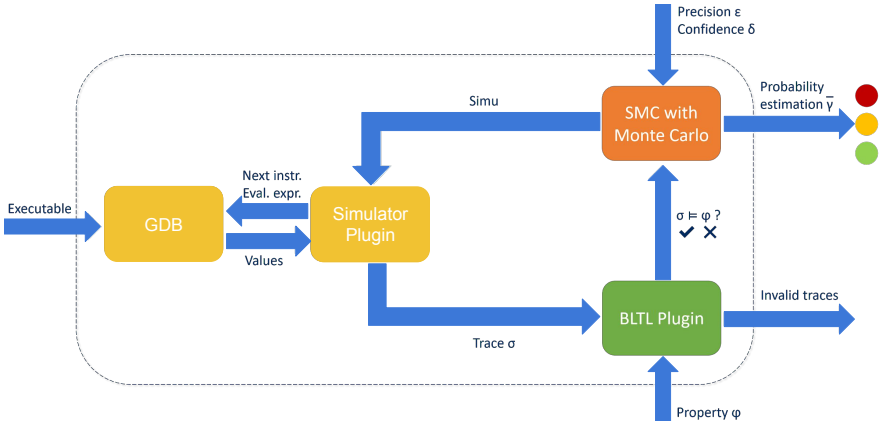

Fig. 2. C-SMC architecture with GDB as the runtime engine.

erty. To do so, the *checker* will ask the *Simulator* to initialize a new execution trace and control the simulation by requesting new states to check properties against. When the Checker obtains a verdict from the traces, the Algorithm is notified, aggregates the results statistically and sends the results through the controller API.

Several plugins provided with Plasma Lab already allow control guarded-command languages or tools such as Matlab/Simulink. In this paper, we will use the external tool to plug C programs. We will then use GDB as a simulator for trace execution of such programs. Finally, we will provide a Checker that extends Plasma's BLT checker with debugger predicates (and hence a link with GDB). By doing so, we will instantiate a new SMC tools to verify C code. Such tool shall be described in the next section.

3 Architecture of C-SMC

This section describes the architecture of C-SMC that allows us to estimate BLTL properties over C programs. Our tool builds on combining the facilities of Plasma Lab and GDB by using the plugins system described in Section 2.3.

An overview of C-SMC's architecture is shown in Fig. 2. The Algorithm (in orange, here SMC with Monte Carlo) will request a simulation from the Simulator (in yellow). The Simulator executes the executable instruction-by-instruction using GDB and generates a trace. The trace is then analyzed by the Checker (in green) that will send results back to the Algorithm. If the simulation resulted in the property not being satisfied, the Checker will output the complete execution trace to a file to be analyzed by the user. This cycle is executed several times to allow the Monte Carlo algorithm to estimate the probability of the outcomes. When this probability has been estimated with acceptable confidence the result is output and Plasma Lab terminates. C-SMC uses Plasma Lab's existing Monte Carlo Algorithm and has new Simulator and Checker plugins.

3.1 Simulator

C-SMC's Simulator executes binaries using GDB and retrieves values from GDB in order to build an execution trace. In addition to the usual values, C-SMC's simulator can also use special values by using the debugger, e.g., the processor's carry flag `CF`, overflow flag `OF`, or a predicate indicating stack memory edits between execution steps. In C-SMC, the Simulator is composed of two main components.

The *Simulator Plugin* can work with any kind of program interface that supports a small API (see the VM from Section 1 for details) for interaction with the execution. This plugin is able to produce an execution trace that Plasma Lab can use, and can then provide the information required for the Checker.

C-SMC's *GDB Communication Interface* is the component that links the Simulator Plugin with GDB. Using GDB's structured language GDB/MI [4] the GDB Communication Interface allows the Simulator Plugin to generate traces from GDB's execution of an executable. The GDB Communication Interface contains an MI message parser and generator to communicate with GDB via I/O streams.

The leftmost part of Fig. 2 shows the workflow of the Simulator Plugin in which GDB is controlled via the GDB Communication Interface to execute the executable instruction-by-instruction and retrieve the values to generate a trace. The trace is then sent to the Checker for validation.

3.2 Checker

The role of C-SMC's Checker is to verify BLTL properties on a trace coming from the Simulator. For doing so, we need an extension to Plasma Lab's existing BLTL plugin to add new expressions that relate to elements accessed by GDB. We will thus consider BLTL properties whose atoms are extended by debugger predicates, i.e., Boolean predicates on the values of variables and elements associated to each state (see Section 2.1).

As an example, the existing BLTL plugin already uses variables defined in the simulator plugin (like the line number and the program counter) with the syntax `G <= 100 line != 10`. The syntax of the BLTL plugin is extended to support debugger expression between two `$` characters, allowing expression like `G <= 100 $table[5] == 15$`. These expressions are evaluated directly by the debugger and a Boolean result is returned.

C-SMC is also able to produce execution traces in XML format to provide a insight into the simulations. An XML execution trace contains a chronological list of all the states encountered during the execution. As illustrated in Fig. 3.2, each state contains information tracked in the analysis; here the program counter `pc`, line number `line`, variables `r` and `x`, and a property `__891715540_id_0`. This provides a deep understanding of the execution flow that may lead to an error. By default, the Checker only outputs the traces of executions that lead to a property violation (`BAD/INVALID` traces), but can be configured to also output traces for executions that lead to no properties violations (`GOOD` traces) and executions that lead to a crash like a segmentation fault (`CRASHED` traces).

```
<state>
  <expr name="pc" expr="$pc">9.3824992236082E13</expr>
  <expr name="line" expr="line">37.0</expr>
  <expr name="r" expr="r">1.6843009E9</expr>
  <expr name="x" expr="x">100.0</expr>
  <expr name="__891715540_id_0" expr="x!=100">0.0</expr>
</state>
```

Fig. 3. Example of a State.

3.3 C-SMC Configuration

C-SMC comes with 2 types of configuration files both in TOML syntax: the model file and the requirement files.

The model file contains the configuration of the simulator inside Plasma Lab. It allows one to specify all the needed configurations for the execution to proceed. An example of such a file is shown in Listing 1.1. The first two lines `executable` and `function` respectively allow the configure the executable to be executed and the function to be monitored inside that executable. The `[simulator]` section allow specifying which of Plasma Lab's simulators is to be used. In C-SMC, we use the `gdb` simulator that we created. The `[simulator.options]` section contains the parameters of the simulator selected in the previous section. Our GDB simulator supports several options such as those from the example detailed below.

- `CF`: this enables the monitoring of the `Carry Flag`. This flag is used to indicate when an arithmetic carry or borrow out of the most significant arithmetic logic unit bit position.
- `OF`: this enables the monitoring of the `Overflow Flag`. This flag is used to indicate that an arithmetic overflow happened during the last instruction.
- `STACK_M`: the `Stack Modification` evaluates to true if the stack above the one of this function has been edited since the last operation (useful to detect buffer overflows).
- `gdb_path`: is the path of the `gdb` executable on the local system.

The last section, `variables` allows defining the variables we want the debugger to track for us (and use to verify properties). These variables are named expressions that will be evaluated during the simulation and accessible by their name in the properties. These expressions need to fit in a Java double value to be usable from the Plasma Lab interface. In Listing 1.1, the first line creates a variable `x` that will contain the value of the variable `x` of the `main` function and a variable `y` that contains the value of the index 5 of the array `buffer`.

To specify the conditions that should be checked while running the program, C-SMC uses requirement files as the one shown in Listing 1.2. Several requirement files might be specified for the same program. A requirement file starts with the `traces` section. This section allows configuring the output of the generated execution traces. The `type` option specifies if the trace must be output on the

```
executable = "../path/to/binary"
function = "main"
[simulator]
name = "gdb"
[simulator.options]
CF = true
OF = true
STACK_M = true
gdb_path = "/usr/bin/gdb"
[variables]
x = "x"
y = "(int) buffer[5]"
```

Listing 1.1. Example of a model file configuration.

```
[traces]
type = "one per file"
printall = "true"
folder = "../path/to/results/"
prefix = "example_prefix"
[BLTL]
G <= #1000 x > 10 =>
    $choosenString != NULL$
```

Listing 1.2. Example of a requirement file configuration.

standard output, in several files or in a single file. The `BLTL` section contains the BLTL formula to be checked on the program. In the example from Listing 1.2, the BLTL formula contains 3 parts. The first part `G <= #1000` specifies the bound (1000 steps in this case). The second part `x > 10` uses the variable x defined in Listing 1.1 to compare it against the value 10. The last part, `$choosenString != NULL$` is a predicate that uses a debugger expression (see Section 3.2) to check if the variable `choosenString`, defined inside the program, is not null.

3.4 Running C-SMC

To illustrate how to run C-SMC let us start with the simple example in Listing 1.3. In this example, the function `contains` is searching for `elem` in the array `arr`. To do so, `contains` iterates over the array. When the desired value is found `contains` memorizes it by storing the value `1` inside the variable `found` and then stops the loop by using a `break`. When `contains` exits the loop, the value of `found` is checked to verify is the value was present in the array. If so, the index of the value is returned, otherwise the function return `-1`.

```
/* return the index of elem in arr, returns -1 if absent */
int contains(int arr[], int size, int elem) {
        int i = 0, current = 0, found = 0;
        for (i=0; i < size; i++) {
                current = arr[i];
                found = (current == elem);
                if (found)
                        break; // BUG if replaced by continue;
        }
        if (found) {
                return i;
        } else {
                return -1;
        }
}
```

Listing 1.3. A simple Example.

To demonstrate the ability of C-SMC to perform an in-depth inspection of the behavior of a program, we will configure the tool in order to monitor the evolution of internal state of the `contains` function from Listing 1.3. In order for this code to work properly, once the `found` value has been set to `1` it should never go back to another value. If for some reason `found` goes back to another value, the result of the function will be incorrect. This could for instance be caused by a programming mistake where the `break` from line 8 is replaced by a `continue`. In this case, when the value is found, the loop will continue and the value of `found` will be overwritten.

To check this example with C-SMC the first step is to define the configuration model described in Section 3.3. In this file, shown in Listing 1.4, the `executable` and `function` will be set to the name of the binary (`check_array`) and the name of the function (`contains`) respectively. For this example, we do not need to track the values of the flags or to look for modifications of the stack. Thus, the next section to be configured is the `variables` section. In this section we will configure C-SMC to track the values of the variables `found`, `elem` and `i` during the execution of the program. To do so we create variables that tracks the value of the variables inside the program. For the sake of clarity we keep the same name, but we could have chosen another variable name.

```
1  executable = "./check_array"
2  function = "contains"
3  [simulator]
4  name = "gdb"
5  [simulator.options]
6  CF = false
7  OF = false
8  STACK_M = false
9  gdb_path = "/usr/bin/gdb"
10 [variables]
11 found = "found"
12 elem = "elem"
13 i = "i"
```

Listing 1.4. Model file configuration for the contains function.

```
1  [traces]
2  type = "one␣per␣file"
3  printall = "true"
4  folder = "../path/results/"
5  prefix = "check_array"
6  [BLTL]
7  G <= #1000 (found = 1 =>
8      (G <= #1000 found = 1))
```

Listing 1.5. Requirement file for the contains function.

Now that the variables that need to be tracked by C-SMC have been configured the next step is to express the behavior that needs to be verified. This part is done by adding requirement files as presented in Section 3.3. The requirement file for this example is shown in Listing 1.5. In the `traces` section we configure the format of the execution traces. The section `BLTL` contains the requirement we want to express. In this case we need to express the fact that once `found` has reached the value `1` it shouldn't change afterwards. This requirement is not trivial to express, but as explained in Section 2.1 BLTL allows the expression of elaborate causalities. With this ability, it is possible to translate "Once the value of bound switches to 1, this value should stay to 1 in the subsequent steps of the execution" into `G <= #1000 (found = 1 => (G <= #1000 found = 1))`. Now that the configuration is done, we can run C-SMC by launching Plasma Lab

using our plugins described in Section 3 to check the properties and execute the code. For a small illustration, our oracle will search for the value `2` in the array {`1,2,3,4,5,6,7,8`}, i.e. with the value `2` at the index `1`.

```
./plasmacli.sh launch -m model.toml:software-simulator -r req_01.
    toml:software-bltl-checker -a montecarlo -A"Total samples"=10
+----------------+---------------+-----------------------+--------+
| Name           | # Simulations | # Positive Simulation | Result |
+----------------+---------------+-----------------------+--------+
| req_01.toml    | 10            | 10                    | 1.0    |
+----------------+---------------+-----------------------+--------+
```

In this case, C-SMC informs us that over the 10 executions of the code, the property held 10 times. The `Result` column gives us the probability of the property holding (this value might be set to -1 when the execution crashed). For the remainder of this example we replace the `break` by a `continue`. If we run C-SMC again; the output will change to inform us that none of the execution succeeded (0 positive simulation). Understanding why the property didn't hold can be done by inspecting the executions traces to follow the flow of the execution. For this example, the end of the trace will look like this:

```
<trace>
....
<state>
<expr name="pc" expr="$pc">9.3824992235894E13</expr>
<expr name="elem" expr="elem">2.0</expr>
<expr name="line" expr="line">6.0</expr>
<expr name="i" expr="i">2.0</expr>
<expr name="found" expr="found">1.0</expr>
</state>
<state>
<expr name="pc" expr="$pc">9.3824992235897E13</expr>
<expr name="elem" expr="elem">2.0</expr>
<expr name="line" expr="line">7.0</expr>
<expr name="i" expr="i">2.0</expr>
<expr name="found" expr="found">0.0</expr>
</state>
</trace>
```

This shows that between the last two states the value of `found` indeed changed from 1 to 0. The trace shows that this happened while the loop was inspecting the index `2`. Because the value `2` was at the index `1`, we can deduce that the loop went too far and look at the `continue` statement.

4 Use Cases

This section presents three clear examples of errors that are difficult for tools to discover. These errors rely on accurate understanding of the state of the program that is difficult to model or approximate without expensive analysis.

Consider the buffer overflow in Listing 1.6. A buffer of size `8` is overflowed with the character `d` (that has value `100` when converted to an integer). Because the

```c
int main(int argc,char **argv){
  int x = 42;
  char buf[8];
  memset(buf,'\0',sizeof(buf));
  srand(time(NULL));
  int r = rand() % 14;
  printf("%d\n", r);
  // overflow on x when r=13
  strncpy(buf,"ddddddddddddddd",
      r);
  if (x == 100){
    printf("secret");
  } else {
    printf("%s\n", buf);
  }
  return 0;
}
```

Listing 1.6. Buffer Overflow Example.

```c
int search(int arr[],int size,
    int elem) {
  int l = 0;
  int r = size; //not size-1
  int m = 0;
  while (r >= l) {
    m = (l+r) / 2;
    if (arr[m] == elem) {
      return m;
    } else if (arr[m] > elem) {
        r = m - 1;
    } else {
      l = m + 1;
    }
  }
  return -1;
}
```

Listing 1.7. Binary Search Example.

```c
int main(int argc, char **argv) {
  char **myArray = malloc(ARRAY_SIZE*sizeof(char*));
  fillArrayWithValues(myArray); // put values in the array
  if ((rand() % 10) == 1) { // randomly insert null sometimes
    myArray[rand() % ARRAY_SIZE] = NULL; // NULL at a random index
  }
  printf("%s\n", myArray[rand() % ARRAY_SIZE]);
}
```

Listing 1.8. Pointer-to-pointer Example.

variable `x` is located next to `buf` on the stack, the `x` variable can be overwritten by an incorrect write to `buf`. In this case, the `x` variable protects the access to a sensitive part of the code that is not supposed to be accessed in this context. The challenge for detecting this error relies upon determining that an overrun of `buf` leads to a specific modification of `x` leading to the `secret`.

Another difficult challenge for tools is to maintain context information that may not be local and accurately detect errors. Listing 1.7 shows an example where the relation between function arguments is vital to correct behavior and detecting the error. Observe that the `size` should give the size of `arr` and is incorrectly used to initialize `r`. This dependency can lead to an out of bounds access that is very difficult to detect accurately, particularly if the `search` function is called with different arguments at different times. Because C is not a reflective language, when the function receives an array (or a pointer to an array), C has no way to determine the size of that array, hence the need for a `size` parameter. Because of this limitation, most tools will consider the size of the array to be unknown and will not be able to perform a bounds check analysis. C-SMC is able to compare the value of `m` against the value of `size` at runtime, without needing to approximate, making this bound check possible. To

detect this error C-SMC monitors the value of the variable `m` using the formula $m = 0|(0 <= m \& m < size)$.

Listing 1.8 presents a simplified version (full version in the VM from Section 1) of an error where a pointer-to-pointer is randomly set to `NULL`. A random value of the array is then printed, potentially leading to a segmentation fault if the `NULL` is selected. Because of the complexity incurred by replicating a memory model supporting pointers-to-pointers and the low probability of the error happening, this error is challenging to accurately detect. Since the probability of `NULL` being inserted in the array and then being selected to be printed is very low, this example outlines some challenges for C-SMC in term of confidence. The use of SMC as the core execution engine makes C-SMC unable to guarantee program correctness. SMC is able to provide a measure of confidence in the results based on the coverage of the possible executions. To make the verdict as accurate as possible, solutions are discussed in Section 6.

5 Examples and Evaluation

To evaluate C-SMC one clear source of evaluation examples and effectiveness is to ensure that C-SMC can detect errors that other tools on the market can detect. The following tools were used as sources for evaluation. *CBMC* [29] is chosen as another C analysis engine that uses a variation of MC to find errors, also CBMC is freely available and has examples of errors that may benefit from runtime information. *Coverity* [1], a static-analysis tool that integrates with Github [6] and Travis-CI [10] is chosen as an easy and free way for open source projects to scan their code for errors. *CPPCheck* [2] is chosen as being an open source freely available C/C++ analysis tool. *PVS Studio* [8] is chosen as another static analyser tool that can be trialed on C [8].

5.1 Methodology

The methodology used for analysis here is to gather a collection of different examples of errors that at least one of the other tools is able to detect and verify that C-SMC is able to detect this kind of error. Examples are collected from the documentation for each tool and also some examples from this paper (see Section 4 and the VM from Section 1). All the tools in their default configuration are run on all the examples to see if the errors were detected. Note that the focus is to evaluate whether C-SMC detects all the different kinds of errors, and thus all possible configurations for other tools were not explored (this makes direct comparison difficult, however this is not the goal here).

To avoid having too many variations of similar errors, when multiple tools announce support for a similar kind of error (e.g. division by 0) a "merged" single example is created. These merged examples are identified as "Multiple" for origin. The evaluation examples and brief descriptions are in Fig. 4 with the source code available in the Virtual Machine from Section 1.

Example	Origin	Description
static out-of-bounds	CPPCheck	Hard-coded array out-of-bounds.
dynamic out-of-bounds	CPPCheck	`rand()` valued array access that may go out-of-bounds.
divide by zero	Multiple.	A hard-coded a division by 0.
buffer overflow	This paper.	A potential buffer overflow, see Listing 1.6.
local binary search	CBMC	Binary search going out-of-bounds of local array.
parameter binary search	This paper.	Binary search going out-of-bounds, see Listing 1.7.
pointer to pointer	This paper.	Use of potential NULL pointer, see Listing 1.8.
read-only memory	Multiple.	Write into read-only memory.
integer overflow	This paper.	Addition resulting in an integer overflow.

Fig. 4. Evaluation Examples

	CBMC	Coverity	CPPCheck	PVS Studio	C-SMC
static out-of-bounds	X	X	X	X	X
random out-of-bounds	X	X	-	X	X
divide by zero	X	X	X	-	X
buffer overflow	-	X	-	X	X
local binary search	X	-	-	-	X
parameter binary search	-	-	-	-	X
pointer to pointer	-	-	-	-	X
read-only memory	-	-	X	X	X
integer overflow	X	-	-	-	X

Fig. 5. Evaluation results for tools on all source code samples with errors. Detected represented as X and Missed by -.

The results of running a tool on an evaluation example are classified into two possible outcomes. *Detected* where the tool accurately detected the error and *Missed* where the tool did not detect any error (of the right kind).

5.2 Results

The results of the evaluation are summarized in Fig. 5. If one of the other tools was able to detect an error, then the goal was for C-SMC to also detect the error. With that objective being met, observe that there are two examples that only C-SMC is able to detect. The first is the `parameter binary search` described in Section 4. It is expected that this kind of error would be hard for tools to detect because of C not being a reflective language. When the function under inspection receives an array (or a pointer to an array), C has no way to determine the size of that array, hence the need for a `size` parameter. Because of this limitation, most tools will consider the size of the array to be unknown and will not be able to perform a bounds check analysis. However, C-SMC is able to compare the value of `m` against the value of "size" making this bound

check possible. The second is the `pointer-to-pointer error`. C-SMC detects it but, as discussed in Section 4, this is an example of where C-SMC may not find the error. Depending on the probability of the pointer-to-pointer being NULL, C-SMC might have to run many executions to find the error. If insufficient executions are run, the error will not occur and thus C-SMC would report that the program is clean (with some confidence) when it is not (this is discussed further in Section 6). For this specific example, by running 4883 simulations, C-SMC detects the presence of the bug and evaluate its probability to 0.75773090313% with a precision of 10^{-1} and a confidence of 10^{-5}.

Performance: By design C-SMC is executing and instrumenting the binary code of a program, this comes at a cost in term of interaction with the debugger which is itself expensive in resources. Thus other approaches can have significantly better performance in terms of execution time and resource consumption. However, the SMC approach and GDB values allow for the reduction of significant abstraction both in finding good executions and avoiding explosion problems. Some of the tools performing static analysis will outperform C-SMC in terms of execution time and resources consumption. While we can't easily reduce this overhead, future work will optimize C-SMC for performance.

6 Conclusion and Future Work

Finding programming errors is a hard challenge, particularly when they rely on detailed runtime information. C-SMC's approach to combine SMC with runtime information proves effective in finding a variety of (causality) errors that are difficult or beyond the capabilities of many tools. This indicates that C-SMC's approach is useful to finding some of these specific kinds of errors, and helps in broadening the effective tools available.

Future work can proceed in various directions. One directions is to handle concurrent C programs. This would require either definition of a stochastic semantics for such programs, or using non-deterministic SMC algorithms such as those in [22]. In addition, this would require modifying C-SMC to support multi-threading instructions from GDB. We would also be interested in using debuggers with other capabilities, e.g. HP Wildebeest Debugger [3] or Radare2 [9].

Another direction is in using more efficient SMC algorithms such as those based on guided search [14, 20, 28, 31]. This would allows us to handle rare bugs in a more efficient manner.

Finally, in C-SMC properties have to be expressed with BLTL properties extended with debugger predicates. It would be worth defining a pattern-based language to express BLTL properties in a language that can be understood by engineers. This could be achieved by extending the work presented in [35] to debugger predicates. It would also be worth testing further kinds of checks such as memory allocation and freeing would be interesting to add since runtime information could resolve many complexities related to pointer aliasing.

References

1. Coverity Scan. https://scan.coverity.com/, accessed: 2021-01-18
2. CPPCheck: A tool for static C/C++ code analysis. http://cppcheck.sourceforge.net/, accessed: 2021-01-18
3. Debugging Dynamic Memory Usage Errors Using HP WDB. http://www.3kranger.com/HP3000/mpeix/en-hpux/PDF/5014-0301.pdf, accessed: 2021-01-21
4. Debugging with GDB: GDB/MI. https://sourceware.org/gdb/onlinedocs/\\gdb/GDB_002fMI.html, accessed: 2021-01-21
5. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/, accessed: 2020-10-14
6. GitHub. https://github.com/, accessed: 2021-01-18
7. Oracle VM VirtualBox. https://virtualbox.org/, accessed: 2021-04-20
8. PVS-Studio. https://www.viva64.com/en/pvs-studio/, accessed: 2021-01-18
9. Radare2 - A free/libre toolchain for easing several low level tasks like forensics, software reverse engineering, exploiting, debugging,... https://rada.re/n/radare2.html, accessed: 2021-01-21
10. Travis-CI. https://travis-ci.com/, accessed: 2021-01-18
11. Valgrind: an instrumentation framework for building dynamic analysis tools. https://valgrind.org/, accessed: 2021-01-21
12. Agha, G., Palmskog, K.: A survey of statistical model checking. ACM Trans. Model. Comput. Simul. **28**(1), 6:1–6:39 (2018). https://doi.org/10.1145/3158668, https://doi.org/10.1145/3158668
13. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (Apr 2008), google-Books-ID: 5dvxCwAAQBAJ
14. Barbot, B., Haddad, S., Picaronny, C.: Coupling and importance sampling for statistical model checking. In: Flanagan, C., König, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 331–346. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
15. Basu, A., Bensalem, S., Bozga, M., Delahaye, B., Legay, A.: Statistical abstraction and model-checking of large heterogeneous systems. Int. J. Softw. Tools Technol. Transf. **14**(1), 53–72 (2012)
16. Boyer, B., Corre, K., Legay, A., Sedwards, S.: Plasma-lab: A flexible, distributable statistical model checking library. In: Joshi, K., Siegle, M., Stoelinga, M., D'Argenio, P.R. (eds.) Quantitative Evaluation of Systems. pp. 160–164. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
17. Bradley, M., Cassez, F., Fehnker, A., Given-Wilson, T., Huuck, R.: High performance static analysis for industry. Electron. Notes Theor. Comput. Sci. **289**, 3–14 (2012)
18. Bradley, M., Cassez, F., Fehnker, A., Given-Wilson, T., Huuck, R., Junker, M.: Goannasmt–a static analyzer with smt-based refinement (2012)
19. Cadar, C., Godefroid, P., Khurshid, S., Pasareanu, C.S., Sen, K., Tillmann, N., Visser, W.: Symbolic execution for software testing in practice: preliminary assessment. In: Taylor, R.N., Gall, H.C., Medvidovic, N. (eds.) Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011. pp. 1066–1071. ACM (2011). https://doi.org/10.1145/1985793.1985995, https://doi.org/10.1145/1985793.1985995
20. Chockler, H., Ivrii, A., Matsliah, A., Rollini, S.F., Sharygina, N.: Using cross-entropy for satisfiability. In: Shin, S.Y., Maldonado, J.C. (eds.) Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC

'13, Coimbra, Portugal, March 18-22, 2013. pp. 1196–1203. ACM (2013). https://doi.org/10.1145/2480362.2480588, https://doi.org/10.1145/2480362.2480588

21. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ansi-c programs. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 168–176. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
22. D'Argenio, P.R., Legay, A., Sedwards, S., Traonouez, L.: Smart sampling for lightweight verification of markov decision processes. Int. J. Softw. Tools Technol. Transf. **17**(4), 469–484 (2015). https://doi.org/10.1007/s10009-015-0383-0, https://doi.org/10.1007/s10009-015-0383-0
23. David, A., Larsen, K.G., Legay, A., Mikucionis, M., Poulsen, D.B.: Uppaal SMC tutorial. Int. J. Softw. Tools Technol. Transf. **17**(4), 397–415 (2015). https://doi.org/10.1007/s10009-014-0361-y, https://doi.org/10.1007/s10009-014-0361-y
24. David, A., Larsen, K.G., Legay, A., Mikučionis, M.: Schedulability of Herschel revisited using statistical model checking. International Journal on Software Tools for Technology Transfer **17**(2), 187–199 (Apr 2015)
25. Havelund, K., Pressburger, T.: Model checking JAVA programs using JAVA PathFinder. International Journal on Software Tools for Technology Transfer (STTT) **2**(4), 366–381 (Mar 2000)
26. Havelund, K., Rosu, G.: Synthesizing monitors for safety properties. In: Katoen, J., Stevens, P. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2280, pp. 342–356. Springer (2002). https://doi.org/10.1007/3-540-46002-0_24, https://doi.org/10.1007/3-540-46002-0_24
27. Hoeffding, W.: Probability Inequalities for sums of Bounded Random Variables, pp. 409–426. Springer New York, New York, NY (1994)
28. Jegourel, C., Legay, A., Sedwards, S.: Importance splitting for statistical model checking rare properties. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification. pp. 576–591. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
29. Kroening, D., Tautschnig, M.: Cbmc – c bounded model checker. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 389–391. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
30. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: An overview. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) Runtime Verification. pp. 122–135. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
31. Legay, A., Lukina, A., Traonouez, L.M., Yang, J., Smolka, S.A., Grosu, R.: Statistical Model Checking, pp. 478–504. Springer International Publishing, Cham (2019)
32. Legay, A., Nowotka, D., Poulsen, D.B., Tranouez, L.M.: Statistical Model Checking of LLVM Code. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds.) Formal Methods, vol. 10951, pp. 542–549. Springer International Publishing, Cham (2018)
33. Legay, A., Sedwards, S., Traonouez, L.M.: Plasma lab: A modular statistical model checking platform. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques. pp. 77–93. Springer International Publishing, Cham (2016)

34. Li, J., Dureja, R., Pu, G., Rozier, K.Y., Vardi, M.Y.: SimpleCAR: An Efficient Bug-Finding Tool Based on Approximate Reachability. In: Proceedings of the 30th International Conference on Computer Aided Verification, Part II. pp. 37–44. Springer (2018)
35. Mignogna, A., Mangeruca, L., Boyer, B., Legay, A., Arnold, A.: Sos contract verification using statistical model checking. In: Larsen, K.G., Legay, A., Nyman, U. (eds.) Proceedings 1st Workshop on Advances in Systems of Systems, AiSoS 2013, Rome, Italy, 16th March 2013. EPTCS, vol. 133, pp. 67–83 (2013). https://doi.org/10.4204/EPTCS.133.7, https://doi.org/10.4204/EPTCS.133.7
36. Ngo, V.C., Legay, A., Joloboff, V.: PSCV: A Runtime Verification Tool for Probabilistic SystemC Models. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016 - 28th International Conference on Computer Aided Verification. LNCS - Lecture Notes in Computer Science, vol. 9779, pp. 84 – 91. Springer, Toronto, Canada (Jul 2016), https://hal.inria.fr/hal-01406488
37. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977). pp. 46–57. IEEE, Providence, RI, USA (Sep 1977), http://ieeexplore.ieee.org/document/4567924/
38. Raad, A., Berdine, J., Dang, H.H., Dreyer, D., O'Hearn, P., Villard, J.: Local reasoning about the presence of bugs: Incorrectness separation logic. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification. pp. 225–252. Springer International Publishing, Cham (2020)
39. Svejda, J., Berger, P., Katoen, J.: Interpretation-based violation witness validation for C: NITWIT. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12078, pp. 40–57. Springer (2020)
40. Younes, H.L.S.: Ymer: A statistical model checker. In: Etessami, K., Rajamani, S.K. (eds.) Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3576, pp. 429–433. Springer (2005). https://doi.org/10.1007/11513988_43, https://doi.org/10.1007/11513988_43
41. Zuliani, P., Platzer, A., Clarke, E.M.: Bayesian statistical model checking with application to stateflow/simulink verification. Formal Methods in System Design **43**(2), 338–367 (Oct 2013). https://doi.org/10.1007/s10703-013-0195-3, https://doi.org/10.1007/s10703-013-0195-3