Towards Counterexample-Guided *k*-Induction for Fast Bug Detection

Mikhail R. Gadelha*
University of Southampton
United Kingdom

Felipe R. Monteiro
Federal University of Amazonas
Brazil

Lucas C. Cordeiro
University of Manchester
United Kingdom

Denis A. Nicole
University of Southampton
United Kingdom

ABSTRACT

Recently, the *k*-induction algorithm has proven to be a successful approach for both finding bugs and proving correctness. However, since the algorithm is an incremental approach, it might waste resources trying to prove incorrect programs. In this paper, we extend the *k*-induction algorithm to shorten the number of steps required to find a property violation. We convert the algorithm into a meet-in-the-middle bidirectional search algorithm, using the counterexample produced from over-approximating the program. The main advantage is in the reduction of the state explosion by reducing the maximum required steps from k to $\lfloor \frac{k}{2} + 1 \rfloor$.

CCS CONCEPTS

• **Software and its engineering** → **Formal software verification**; • **Theory of computation** → *Verification by model checking*; • **Hardware** → *Bug detection, localization and diagnosis*;

KEYWORDS

Bounded Model Checking; *k*-induction; Formal Software Verification; Bug Detection.

ACM Reference Format:
Mikhail R. Gadelha, Felipe R. Monteiro, Lucas C. Cordeiro, and Denis A. Nicole. 2018. Towards Counterexample-Guided *k*-Induction for Fast Bug Detection. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3236024.3264840

1 INTRODUCTION

Embedded systems are used in a variety of applications, ranging from nuclear plants and automotive systems to entertainment and games [10]. This ubiquity drives a need to test and validate a system before releasing it to the market, in order to protect against system failures. Even subtle system bugs can have drastic consequences, such as the recent Heartbleed bug on OpenSSH, which might have leaked private information from several servers [7].

One promising technique to verify embedded software is called bounded model checking (BMC) [3]. The basic idea of BMC is to check the negation of a property at a given depth: given a transition system M, a property ϕ, and a bound k, BMC unrolls the system k times and generates verification conditions (VC) ψ, such that ψ is satisfiable *iff* ϕ has a counterexample of depth k or less. BMC tools based on Boolean Satisfiability (SAT) or Satisfiability Module Theories (SMT) have been applied on the verification of both sequential and parallel programs [5, 6, 14, 15]. However, BMC tools are aimed at finding bugs; they cannot prove correctness, unless the bound k safely reaches all program states [9].

Although BMC cannot prove correctness by itself (unless it fully unwinds the program), there are algorithms that use BMC as a "component" to prove partial correctness. In particular, the *k*-induction algorithm is an incremental approach that aims to find bugs and prove correctness using an ever increasing number of unwindings. In this paper, we propose to extend the algorithm originally developed for *k*-induction to shorten the number of iterations required to find a property violation. Our main original contribution is an extension to the *k*-induction algorithm, which converts the algorithm into a meet-in-the-middle bidirectional search by using the counterexample generated by the inductive step (*cf.*, Section 3). In fact, the preliminary results show empirically that the number of steps required to find a property violation is reduced to $\lfloor \frac{k}{2} + 1 \rfloor$ and the verification time for programs with large state space is reduced considerably (*cf.*, Section 4).

```
1 int main() {
2   uint32_t n;
3   uint64_t sn = 0;
4   for (uint64_t i = 1; i <= n; i++) {
5     sn = sn + 2;
6     assert(sn == i * 2);
7   }
8   assert(sn == n*2 || sn == 0);
9 }
```

Figure 1: ANSI-C program example with an upper-bound limit up to $2^{32} - 1$.

*E-mail: esbmc@googlegroups.com

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3264840

2 THE *K*-INDUCTION ALGORITHM

The first version of the k-induction algorithm was proposed by Sheeran *et al.* [16]; they apply BMC to find bugs and prove correctness. BMC tools cannot prove correctness unless the bound k is appropriate to reach the completeness threshold (*i.e.*, a value that will fully unroll all loops occurring in the program, often impractically large) [12]. For instance, consider the simple program shown in Figure 1, the assertion in line 8 always holds, regardless of the initial value of n in line 2. BMC tools such as CBMC [5], ESBMC [6] or LLBMC [13] typically reproduce the loop k times (lines 4 – 7 in Figure 1) as the code snippet in Figure 2 and are unable to verify that program unless the loop is fully unrolled, *i.e.*, the *unwinding assertion* fails if $k < 2^{32} - 1$.

```
1  uint64_t i = 1;
2  if(i <= n) {
3    sn = sn + 2;
4    assert(sn == i * 2);   } k copies
5    i++;
6  }
7  // unwinding assertion
8  assert(!(i<=n));
```

Figure 2: Finite k unwindings done by BMC.

Let a given program P under verification be a finite transition system M. Consider that $T(s_i, s_{i+1})$ is the transition relation for M over the state variables s_i and s_{i+1}, Φ is the set of safety properties, $\phi(s)$ is the formula encoding for states satisfying a safety property, and $\psi(s)$ is the formula encoding for states satisfying a completeness threshold [12], which can be smaller than or equal to the maximum number of loop-iterations occurring in the program. Based on such formalization, the k-induction algorithm performs three checks for each step k: the base case $B_k(k)$, forward condition $F_k(k)$ and inductive step $I_k(k)$, for $k = [1, d]$, where d is the depth of the transition system M [9]. In the first check, the base case $B_k(k)$ works as the standard BMC approach and it is satisfiable *iff* $B_k(k)$ has a counterexample of length k or less [2]:

$$B_k(k) = \exists s_1 \dots s_k . I(s_1) \wedge \bigwedge_{i=1}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=1}^{k} \neg\phi(s_i). \quad (1)$$

In the second check, the forward condition $F_k(k)$ checks if the completeness threshold $\psi(s)$ holds for the current k. This is established by checking if the following is unsatisfiable:

$$F_k(k) = \exists s_1 \dots s_k . I(s_1) \wedge \bigwedge_{i=1}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=1}^{k} \neg\psi(s_i). \quad (2)$$

No safety property $\phi(s)$ is checked in $F_k(k)$ as they were already checked for the current k in the base case. Finally, the inductive step $I_k(k)$ checks if whenever $\phi(s)$ holds in k states (*i.e.*, $s_1, \dots, s_k$), $\phi(s)$ also holds for the next state s_{k+1}. This is established by checking if the following is unsatisfiable:

$$I_k(k) = \exists s_1 \dots s_{k+1} . \bigwedge_{i=1}^{k} (\phi(s_i) \wedge T(s_i, s_{i+1})) \wedge \neg\phi(s_{k+1}). \quad (3)$$

Therefore, the k-induction algorithm at a given k is:

$$k_{ind}(P,k) = \begin{cases} P \text{ has a bug}, & \text{if } \neg B_k(k), \\ P \text{ is correct}, & \text{if } B_k(k) \wedge [F_k(k) \vee I_k(k)], \\ k_{ind}(P, k+1), & \text{otherwise.} \end{cases} \quad (4)$$

It worth noticed in Eq. (4) that a bug is only reported in the base case (*i.e.*, $B_k(k)$) and if a violation is reported in the inductive step (*i.e.*, $I_k(k)$) the algorithm assumes the results is spurious, thus, it calls itself recursively for the next iteration. The k-induction algorithm is a complete and optimal search algorithm (*i.e.*, always find the shortest counterexample), with complexity $O(bd)$ and state space $O(b^+d^+)$ [9]. Indeed, Jovanović et al. [11] show that the k-induction proof rule can be more powerful and concise than regular induction.

3 COUNTEREXAMPLE-GUIDED *K*-INDUCTION ALGORITHM

The k-induction algorithm is being applied to solve a number of different verification problems, but it has its own limitations. The biggest one is the fact that it performs three checks for each k (*i.e.*, base case, forward condition and inductive step). The inductive step $I_k(k)$ is the most computationally expensive one; it is an overapproximation, forcing the SMT solver to find a set of assignments in a larger state space than the original program [9]. Moreover, the computation is wasted if a counterexample is found by the inductive step, as it is assumed to be spurious (*cf.* Section 2).

Consider a program P modeled as a state transition system M contains a set of variables $V = \{v_1, \dots, v_n\}$, where n is the number of variables in the program. In the state transition system M, an state s_i is a tuple $\langle p_c, V_i \rangle$, where p_c is the program counter and $V_i = \{v_1^i, \dots, v_n^i\}$ are the values of all program variables in that state. A transition t is a guarded assignment $\langle [\gamma], \texttt{x := e} \rangle$, where γ is a predicate over the program variables and e is an expression assigned to x.

Definition 1. **Counterexample** *is a sequence of states $\pi = \langle s_i, \dots, \xi \rangle$ of length k that represents a path from an initial state s_i to an error state ξ.*

In order to tackle the aforementioned problems in the k-induction algorithm, we propose to use the counterexample generated by the inductive step to speed up the bug finding check (*i.e.*, the base case). Our extension converts the k-induction algorithm into a bidirectional search approach by searching simultaneously both forward (*i.e.*, from the initial state s_1) and backward (*i.e.*, from the error state ξ detected in the inductive step $I_k(k)$) and stop if both searches meet in the middle as shown in Figure 3.

The base case $B_k(k)$ is the forward part of the algorithm, since it tries to find a counterexample $\pi_B = \langle s_1, \dots, \xi \rangle$ that represents a path from the initial state of the program P (*i.e.*, s_1) to an error state ξ. The inductive step $I_k(k)$ is the backward part of the algorithm; it tries to find a counterexample $\pi_I = \langle s_d, \dots, \xi \rangle$, from any depth d in the state transition system M.

Lemma 1. *The counterexample $\pi_I = \langle s_d, \dots, \xi \rangle$ produced by the inductive step $I_k(k)$ is a path that leads to a property violation (i.e., an error state ξ); reaching any value in that path (i.e., $s_d, s_{d+1}, \dots, \xi$) will lead to this property violation.*

Based on Lemma 1, if at least one state of π_I is reachable from the initial state s_1, then the error state ξ is reachable from the initial state s_1. Thus, given a counterexample $\pi_I = \langle s_i, \dots, \xi \rangle$ from the inductive step, our extension selects the first state $s_d = \langle p_c, V_d \rangle$

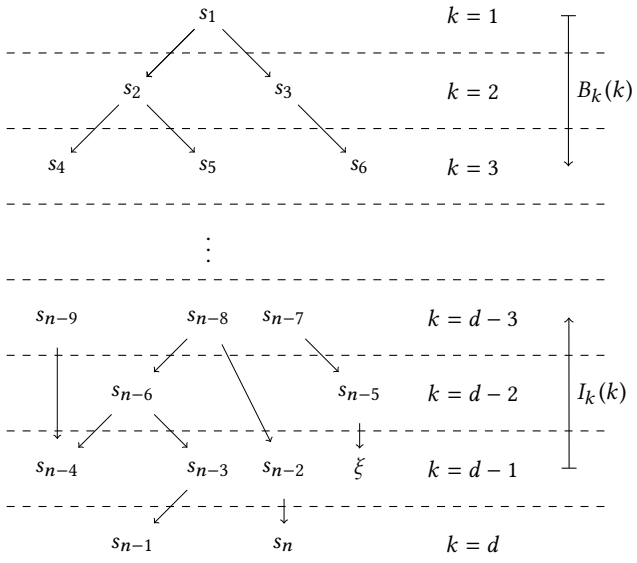

Figure 3: Visual representation of our extension. Each dashed section represents the states reachable after k iterations. The arrows show the "direction" of the verification by the base case $B_k(k)$ and the inductive step $I_k(k)$. The forward condition $F_k(k)$ is not shown in this representation but it is a forward check, similar to $B_k(k)$.

and translates it into a new safety property:

$$\varphi(s_i, s_d) = \bigvee_{j=1}^{n} v_j^i \neq v_j^d \tag{5}$$

which checks if a given state s_i is the first state in the counterexample. Given the optimal nature of the algorithm, this is sufficient to find the property violation. Our algorithm then defines a new base case step:

$$B'(k, s_d) = \exists s_1 \ldots s_k . I(s_1) \wedge \bigwedge_{i=1}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=1}^{k} \neg(\phi(s_i) \wedge \varphi(s_i, s_d)). \tag{6}$$

Thus, our proposed extension checks, in the new base case $B'(k, s_d)$, whether we can reach any value in path π_I. Note that this will only be applied to assertions inside a loop, as the inductive step only overapproximates loop variables. The algorithm will still be complete and, assuming that the error state is reachable in k steps from the initial state, the solution will be found in $\lfloor \frac{k}{2} + 1 \rfloor$, because the forward and backward searches each have to go only half way. Our extension repurposes the goal of the inductive step, from proving correctness to find paths that lead to error states ξ.

There are cases where our approach is not effective, in particular, in cases where the error state found by the inductive step is unreachable (originated from a spurious counterexample), in which case the verification process is equivalent to plain k-induction (little to no overhead is introduced by our extension). Multiple reachable error states are not an issue, as all of them will be checked in the base case individually.

3.1 Running Example

```
1 unsigned int a = 1;
2 while(1)
3 {
4   if(a == 6)
5     assert(0);
6   a++;
7 }
```

(a) Original program

```
1 unsigned int a = 1;
2 while(1)
3 {
4   if(a == 6)
5     assert(0);
6   a++;
7   // added assertion
8   assert(a != 5);
9 }
```

(b) Modified program

Figure 4: Code snippet example.

Consider our extended k-induction algorithm applied to the code snippet shown in Figure 4a. It requires 6 iterations to reach the assertion failure. This means that the base case $B_k(k)$ will be called 6 times (*i.e.*, $k = [1 .. 6]$), thus, the forward condition $F_k(k)$ and the inductive step $I_k(k)$ will be called 5 times each (*i.e.*, $k = [1 .. 5]$). The base case will produce, for $k = 6$, the counterexample $\pi_B = \langle s_1 \rightarrow \langle 1, \texttt{a = 1}\rangle, s_2 \rightarrow \langle 2, \texttt{a = 2}\rangle, s_3 \rightarrow \langle 3, \texttt{a = 3}\rangle, s_4 \rightarrow \langle 4, \texttt{a = 4}\rangle, s_5 \rightarrow \langle 5, \texttt{a = 5}\rangle, s_6 \rightarrow \langle 6, \texttt{a = 6}\rangle, \xi \rightarrow \langle 7, \texttt{assert(0)}\rangle\rangle$, which is a set of assignments that leads to an assertion failure. Now, consider the counterexample $\pi_I^1 = \langle s_6 \rightarrow \langle 6, \texttt{a = 6}\rangle, \xi \rightarrow \langle 7, \texttt{assert(0)}\rangle\rangle$ generated by the inductive step for $k = 1$, or $\pi_I^2 = \langle s_5 \rightarrow \langle 5, \texttt{a = 5}\rangle, s_6 \rightarrow \langle 6, \texttt{a = 6}\rangle, \xi \rightarrow \langle 7, \texttt{assert(0)}\rangle\rangle$ for $k = 2$. For $k = 1$, the property violation is reachable when `a == 6`, in that case, the inductive step can be interpreted as the question "is there counterexample of size 1 in the program?". Furthermore, each k increment extends the set of assignments in the path back to the initial state, *e.g.*, $k = 2$ can be interpreted as the question "is there counterexample of size 2 in the program?", and so forth.

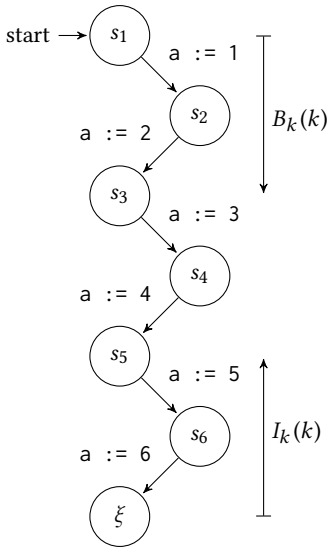

Figure 5: Unrolled state transition system M from code snippet in Figure 4a and the "direction" of the verification for the base case $B_k(k)$ and the inductive step $I_k(k)$.

Figure 4b shows the modified program from Figure 4a, based on the counterexample π_I^2; the program contains one variable so our extension only asserts one inequality. For $k = 2$, the first state reachable in the path to the error state

is $a := 5$, as previously shown by the counterexample. For $k = 3$, the first reachable state is $a := 4$ and the program will be changed accordingly during the verification. Figure 5 shows the unrolled state transition system M and the "direction" of the verification for the base case and the inductive step, the latter based on the counterexample. Through that approach, every time that the inductive step produces a counterexample, our k-induction extension will collect the first state of this counterexample and will add a new safety property in the state transition system M, which will reduce the state space to be explored.

4 EXPERIMENTAL EVALUATION

In order to evaluate our k-induction algorithm extension, we selected a number of benchmarks from the International Competition on Software Verification (SV-COMP) 2017 [1]. We compare the results from the original k-induction and our extended version. Currently, our extension is able automatically to identify the initial state from the counterexample generated by the inductive step, but it is unable to correctly generate and add the new verification property $\varphi(s_i, s_d)$ required by $B'(k, s_d)$. As a result, the programs evaluated by our extension were manually changed to add the initial state s_d of the counterexample generated by the inductive step. In order to establish a fair comparison, we also add the time to obtain the initial state to our extended k-induction. We do not compare our extended k-induction to plain BMC since we are interested in checking the efficiency and efficacy of our new approach compared to the existing k-induction algorithm.

Benchmark description. The benchmarks called *sum0** are similar to the program in Figure 1, but contain a bug in different depths. The benchmarks *rangesum** check if a function is "deterministic" w.r.t. all possible permutations of an input array; the number in the benchmark name represents the size of the array. The benchmark *const* checks if a constant holds after 1024 iterations (but checks the wrong value after the iterations); *diamond* checks if a counter that is being nondeterministically incremented is even after 99 iterations; and *Problem01_label15* is the representation of a reactive system.

Experimental setup. All experiments were conducted on a computer with an Intel Core i7-2600 running at 3.40GHz and 24GB of RAM under Fedora 25 64-bit. We used ESBMC $v5.0$ [8] and no time or memory limit were set for the verification tasks.

Availability of data & tools. Our experiments are based on a set of publicly available benchmarks. All tools, benchmarks, and results of our evaluation are available on our web page.[1]

4.1 Preliminary Results

Table 1 shows the preliminary results obtained from the original k-induction and our proposed extension. Here, *LOC* is the number of lines in the program, T is the time needed to verify the program in seconds, M is the memory used by the tools to verify the programs in megabytes[2] and k is the number of steps needed to find the bug. The last lines show the average and cumulative numbers for each of the columns. We order the benchmarks in relation to the memory required by the original k-induction.

Table 1: Preliminary evaluation over the SV-COMP 2017 benchmarks.

Benchmark	LOC	k-induction			Extended k-induction		
		T(s)	M(MB)	k	T(s)	M(MB)	k
sum04.c	19	**1**	38.7	9	**1**	38.7	5
sum01.c	18	**1**	38.9	11	**1**	38.8	6
sum03.c	25	3	39.1	11	**1**	38.8	6
diamond1.c	24	13	43.6	51	**6**	39.1	26
rangesum.c	64	7	66.2	4	**1**	39.0	2
rangesum05.c	59	11	72.3	6	**1**	65.4	3
rangesum10.c	59	28	78.2	11	**16**	47.5	6
Problem01_label15.c	594	7	87.3	5	**5**	70.3	4
rangesum20.c	59	101	99.9	21	**26**	78.2	12
rangesum40.c	59	847	269.5	41	**90**	113.9	22
const.c	20	2606	796.6	1025	**890**	253.2	513
rangesum60.c	59	80272	1106.9	61	**159**	134.6	32
Average	88	6991	228.1	104	**99**	79.8	53
Total	1059	83897	2737.2	1255	**1197**	957.5	638

The first noticeable aspect of the results is that the time of the verification is not related to the number of steps or the program size. The closest predictor of the verification time is the state space explored by each step (more specifically the inductive step), the bigger the state space, the longer it will take to find a solution; this can be approximated by the memory used by the tool during the verification.

The evaluation for this set of benchmarks shows that our extension to the k-induction algorithm potentially cuts the verification time considerably in cases where the state space explored is large. For small cases (*e.g.*, the *sum0*.c* benchmarks), our extension does not slow things down or use more memory than the original k-induction; for large cases, the gains are substantial (*e.g.*, the verification time of *rangesum60.c* is 504x faster). In terms of the steps needed to find the bug, the extended version of the k-induction required $\lfloor \frac{k}{2} + 1 \rfloor$, as expected.

We also compared the results of the extended k-induction with an incremental BMC approach and we observed that our extended k-induction is as good as an incremental BMC, in most cases. The extended k-induction is as fast as the incremental BMC for small bounds (and it is even faster than the incremental BMC approach on *rangesum60.c*), and it is not as slow as the original k-induction for large bounds. Our approach lies between the original k-induction and the incremental BMC, it is able to prove correctness and find bugs consuming less resources (*i.e.*, time and memory) than the original k-induction but, when the program is unsafe, it is slower than the incremental BMC.

5 RELATED WORK

Bischoff *et al.* [4] propose a methodology to use BDDs and SAT solvers for the verification of programs. The BDDs are responsible for the target enlargement, collecting the under-approximate reachable state sets, followed by the SAT-based verification with the newly computed sets. The authors implemented the technique in the Intel BOolean VErifier (BOVE) and showed that the time was up to five times smaller. Compared to this work, we only use k-induction and SMT solvers; the inductive step in the k-induction algorithm is responsible for enlarging the target and the SMT solver checks for satisfiability.

[1] http://esbmc.org/

[2] We used the command `/usr/bin/time -v` from linux to measure both the time and the memory usage

Jovanović *et al.* [11] present a reformulation of IC3, separating the reachability checking from the inductive reasoning. They further replace the regular induction algorithm by the *k*-induction algorithm and show that it provides more concise invariants. The authors implemented the algorithm in the SALLY model checker using Yices2 to do the forward search and MathSAT5 to do the backward search. They showed that the new algorithm is able to solve a number of real-world benchmarks at least as fast as other approaches. Compared to this work, our proposed extended *k*-induction uses consecutive BMC calls to find a solution. We also implement our approach independent of solvers and it can be used with any SMT solver supported by ESBMC; both searches, however, will be done with the same solver.

6 CONCLUSION

In this paper, our main contribution is a novel extension to the *k*-induction algorithm, to perform a bidirectional search instead of the conventional iterative deepening search. The extension is currently under development using ESBMC. We plan to evaluate the improvement over the SV-COMP benchmarks, where the original *k*-induction algorithm already proved to be the state-of-art, if compared to other *k*-induction tools [1]. The preliminary results show that the extension has the potential to substantially improve the verification time for problems with large state space, while maintaining a small verification time for small programs. In one particularly large program (in terms of state space), our extension allowed the *k*-induction algorithm to find the property violation on average using half of the steps and a fraction of the resources.

REFERENCES

[1] Dirk Beyer. 2017. Software Verification With Validation Of Results (Report On SV-COMP 2017). In *TACAS (LNCS)*, Vol. 10206. 331–349.

[2] Armin Biere. 2009. *Handbook Of Satisfiability*. Vol. 185. IOS Press, Chapter 14, 455–481.

[3] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking Without BDDs. In *TACAS (LNCS)*, Vol. 1633. 193–207.

[4] Gabriel P. Bischoff, Karl S. Brace, G. Cabodi, and S. Nocco, S.and Quer. 2005. Exploiting Target Enlargement And Dynamic Abstraction Within Mixed BDD And SAT Invariant Checking. *Electronic Notes in Theoretical Computer Science* 119, 2 (2005), 33–49.

[5] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool For Checking ANSI-C Programs. In *TACAS (LNCS)*, Vol. 2988. 168–176.

[6] Lucas C. Cordeiro, Bernd Fischer, and João Marques-Silva. 2012. SMT-Based Bounded Model Checking For Embedded ANSI-C Software. *IEEE Transactions on Software Engineering* 38, 4 (2012), 957–974.

[7] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. 2014. The Matter Of Heartbleed. In *IMC*. 475–488.

[8] Mikhail R. Gadelha, Felipe R. Monteiro, Jeremy Morse, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. 2018. ESBMC 5.0: An Industrial-Strength C Model Checker. In *ASE*. ACM, 888–891.

[9] Mikhail Y. R. Gadelha, Hussama I. Ismail, and Lucas C. Cordeiro. 2017. Handling Loops In Bounded Model Checking Of C Programs Via K-induction. *STTT* 19, 1 (2017), 97–114.

[10] Steve Heath. 2003. *Embedded Systems Design*. Newnes, Oxford, United Kingdom. 430 pages.

[11] Dejan Jovanović and Bruno Dutertre. 2016. Property-directed *k*-induction. In *FMCAD*. 85–92.

[12] Daniel Kroening, Joël Ouaknine, Ofer Strichman, Thomas Wahl, and James Worrell. 2011. Linear Completeness Thresholds For Bounded Model Checking. In *CAV (LNCS)*, Vol. 6806. 557–572.

[13] Florian Merz, Stephan Falke, and Carsten Sinz. 2012. LLBMC: Bounded Model Checking Of C And C++ Programs Using A Compiler IR. In *VSTTE (LNCS)*, Vol. 7152. 146–161.

[14] Felipe R. Monteiro, Erickson H. da S. Alves, Isabela S. Silva, Hussama I. Ismail, Lucas C. Cordeiro, and Eddie B. de Lima Filho. 2018. ESBMC-GPU A Context-Bounded Model Checking Tool To Verify CUDA Programs. *Science of Computer Programming* 152 (2018), 63 – 69.

[15] Shaz Qadeer and Jakob Rehof. 2005. Context-Bounded Model Checking Of Concurrent Software. In *TACAS (LNCS)*, Vol. 3440. 93–107.

[16] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. 2000. Checking Safety Properties Using Induction And A SAT-Solver. In *FMCAD*. 108–125.