Towards Integrity and Reliability in Embedded Systems: The Synergy of ESBMC and Arduino Integration

Rafael G. Silvestrim*, Felipe V. Trigo*, Williame Rocha*, Michael R. S. Vieira†,
Jogno V. Junior*, Otoniel da C. Mendes*, Rafael Sá Menezes†‡, Lucas C. Cordeiro†‡
*State University of Amazonas, Brazil, †Federal University of Amazonas, Brazil, ‡University of Manchester, UK

***Abstract*—We've developed and evaluated a new method called ESBMC-Arduino that combines the ESBMC model checker with the Arduino hardware platform. This verification method helps ensure the safety and safety of Arduino C code by finding and, to some extend, preventing errors, thus making the entire system code more reliable. This collaboration is particularly useful for critical embedded systems, improving safety analysis, and promoting contract-driven development practices. We also advocate that our proposed method is valuable for teaching and advanced research in formal verification and embedded systems safety. Our experimental results show that using ESBMC for formal verification of Arduino code leads to better error detection, more accurate code, and increased reliability. This demonstrates that ESBMC-Arduino effectively identifies software vulnerabilities (e.g., memory management and overflow prevention) and enhances the safety of embedded systems.**

***Index Terms*—ESBMC, Arduino, Embedded Systems, Formal Verification, and Software safety.**

I. Introduction

Due to technological advancements, embedded systems have rapidly become integral to our daily lives [1]. Embedded systems constitute one of the three basic classes of digital systems, emulation and prototyping systems, and general-purpose computing systems [2]. Their definition can be understood as a specialized computational system integrated into a larger system or machine [3]. These systems are designed to interact continuously with a dynamic external environment [4]. They can exist in physical forms, such as mobile robots and factory process controllers, as well as purely computational entities like calendar management programs [5]. Their prevalence spans various sectors, including industry, energy, agriculture, space, health, and education. This is due to the potential benefits that these systems can provide, ranging from their compact size to energy efficiency [6]. These advantages are tied to their main characteristic, functionality, as they are designed to perform highly specific procedures [7].

One such example is the Arduino UNO Rev3 board, part of the third revision of the Arduino UNO platform [8]. This board can collect data from attached sensors and connect to wireless communication modules, enabling information exchange through networks like Wi-Fi and Bluetooth. The C/C++ programming language is often employed in this context [9]. Like a Programmable Logic Controller (PLC), Arduino plays a role in managing industrial systems [10]. Its primary proposal is simplifying automation for control levels in residential, commercial, or mobile environments [11]. Additionally, Arduino's applications can be expanded through boards incorporating various devices and easily connected to the platform [12]. These boards, known as modules or Shields, can operate in multiple functions, such as GPS receivers, Ethernet or wireless network modules, and other features [13].

As these systems become increasingly intertwined with our daily lives, ensuring their safety and reliability becomes crucial [14]. These safety requirements can be formalized and applied automatically to verify programming languages such as C [15] and C++ [16]. These works use the Bounded Model Checking (BMC) technique, which is a method to explore and verify a transition system up to a limited bound k. BMC has also been extended to support the proof by induction of safety properties in C programs [17]. To apply the technique for Arduino, there is a need to create operational models describing the system behavior [18].

Here, we propose a method to verify Arduino Software using BMC formally. The method consists of taking advantage of previous C/C++ BMC tools and extending them to support the intrinsic features of the *Arduino Language*. The prototype, named *ESBMC-Arduino*, was implemented on top of the Efficient SMT-based Context-Bounded Model Checker (ESBMC) [15], an award-winning BMC for automatic verification of C programs. In summary, this work has the following original contributions:

- A method to automatically verify Arduino software based on BMC;
- A series of operational models for the Arduino platform that BMC tools can use to verify *Arduino Language*;
- Benchmarks for the *Arduino Language*. These benchmarks range from memory properties to arithmetic and even user-defined properties.

II. Preliminaries

In the context of this work, the following terms are used:

- **Safety Properties:** Safety properties are properties that ensure a good behaviour of the systems, such as memory constraints [19] or time.
- **Reliability:** by ISO/IEC 27000:2018 [20] is the property of the system having consistent behaviour and results.

A. Bounded Model Checking

The Model Checking (MC) technique is widely used in the automated verification process of systems. This approach makes it feasible to model systems and define properties that require validation using automated theorem proofs formulated from logical expressions. Some systems, however, can lead to exponential formulas. Bounded Model Checking (BMC) was conceived to expedite the detection of property violations in fixed bound. This is achieved by imposing restrictions on the number of loop iterations and recursion depth during analysis. However, this approach cannot guarantee accuracy unless all loops and recursions are fully expanded, which may not be feasible in certain programs involving infinite loops [21]. ESBMC represents an open-source model checker with a consolidated solution that adopts a permissive license, enabling the verification of single-threaded and multi-threaded C programs [22]. ESBMC can automatically examine predefined properties, such as bounds validation, pointer protection, and overflow prevention. Furthermore, it allows users to establish custom assertions for their programs, which are also subject to automated verification [23]. Through C++ and Python interfaces, ESBMC provides APIs that allow for exploring internal data structures, enabling analysis and expansion at any stage of the verification process [15].

III. Verifying the Arduino Language

Arduino is programmed through the *Arduino Language*, which is a C-like language with additional constructs[1]. The language gives developers easy interfaces to control attached devices (e.g., GPIO controller). In addition, C/C++ languages are expected to have a "main" entry point. In the *Arduino Language*, this is replaced with the intrinsic functions `setup` (used to set up the application I/O) and `loop` (the code to be executed uninterruptedly). Finally, the language can be used with C/C++ with a few caveats[2].

The language contains intrinsic variables and functions to enable the use of Arduino boards. They are configured and set as board bases on the `Arduino.h` header. Listing 1 contains an example of using the libraries to set up a blinking LED. The header also provides facilities to write analog and digital inputs as a method to pool ports. Lastly, the header also offers interfaces to use PINs as interrupts. This lets developers set actions to happen based on the status of some ports (e.g., the user presses a button). An example of interrupt is shown in Listing 2.

When considering formal verification, the main challenges for Arduino are: modeling the behavior of the PINs (both analog and digital), the unbounded nature of the "loop" entry-point, and the non-deterministic behavior of interrupts.

A. Operational Models for Arduino

Each OM (Operational Model) is constructed by analyzing the Arduino program and its respective libraries and functions.

[1]https://www.arduino.cc/reference/en/

[2]https://playground.arduino.cc/Main/UsingCPlusPlus/

```
1  #include <Arduino.h>
2  void setup() {
3    pinMode(LED_BUILTIN, OUTPUT);
4  }
5
6  void loop() {
7    digitalWrite(LED_BUILTIN, HIGH);
8    delay(1000);
9    digitalWrite(LED_BUILTIN, LOW);
10   delay(1000);
11 }
```

Listing 1: Blink example taken from Arduino Documentation. In the example, the initial "setup" sets `LED_BUILTIN` as an `OUTPUT` in the program (by setting the `pinMode`). The LED actual number varies from board to board, and setting it as an "OUTPUT" only reserves the pin for this process. The function loop consists of setting the pin to "HIGH" (i.e., activating it), waiting for a delay of 1 second, and then setting it to "LOW" (i.e., deactivating it). This loop is repeated until the process is killed.

```
1  const byte ledPin = 13;
2  const byte interruptPin = 2;
3  volatile byte state = LOW;
4
5  void setup() {
6    pinMode(ledPin, OUTPUT);
7    pinMode(interruptPin, INPUT_PULLUP);
8    attachInterrupt(
9      digitalPinToInterrupt(interruptPin),
10     blink,
11     CHANGE);
12 }
13
14 void loop() {
15   digitalWrite(ledPin, state);
16 }
17
18 void blink() {
19   state = !state;
20 }
```

Listing 2: Interrupt example taken from Arduino Documentation. In the example, the initial "setup" sets the `interruptPin` to be used as an input with the default value being "HIGH". This pin is then attached to an interrupt event, which calls the function `blink` on any `CHANGE` on the pin value. The blink function changes the value that should be written into the `ledPin` in the main loop. If a button is connected to the pin, pressing it would turn the board LED on and off.

In our main example, "led.c", the "Arduino.h" file contains functions for controlling analog and digital pins, which were

modeled in "Arduino.c" following the function's signature, as can be observed with the `digitalWrite()`, `delay()`, and `pinmode()` functions. Listing 3 illustrates the prototypes in the "Arduino.c" file. In its core, this action enables BMCs to recognize and understand how Arduino functions should behave.

```
#include <Arduino.h>
uint8_t PINS_value[NUM_DIGITAL_PINS];
uint8_t PINS_mode[NUM_DIGITAL_PINS];

void digitalWrite
  (uint8_t pin, uint8_t value) {
    assert(PINS_mode[pin] == INPUT
        || PINS_mode[pin] == OUTPUT);
    PINS_value[pin] = value;
  }

void delay(unsigned long ms) {
  // no operation
}

void pinMode
  (uint8_t pin, uint8_t mode) {
    PINS_mode[pin] = mode;
}
```

Listing 3: Operational model of led-blinking-specific function prototypes. The models rely on a global array that will keep track of values for the PIN, and the same approach can be used for Analog inputs. The delay function is modeled as a no-op and can be used to force a thread interleaving. Finally, these models were simplified for readability, and I/O is assumed to be atomic.

B. Arduino entry-point

In Arduino, the program initializes through the `setup` function into the unbounded `loop` function. For BMC, we can describe it as an implication rule, i.e., the loop can only happen if the setup has succeeded. We need the C equivalent of the entry point to use an off-the-shelf C-verifier. Listing 4 illustrates the equivalent C version.

```
int main() {
    setup();
    while(1) {
        loop();
    }
    return 0;
}
```

Listing 4: Equivalent entry-point for an Arduino program in C. This program assumes that the setup and loop functions were defined (or are trivial).

C. Modelling interrupts

The interrupts of Arduino can be modeled similarly to those proposed by Cordeiro [24]. The algorithm consists of creating a thread that keeps pooling for the value. If the value changes, then the function can be invoked. Listing 5 illustrates an example of the interruption program.

```

void setup() {
  // ...
  create_thread(interruptLoop);
}

void interruptLoop() {
  while(1) {
    if(*) blink();
  }
}
```

Listing 5: Interrupt BMC equivalent for Listing 2. In the example, the intrinsic function `create_thread` creates a thread, and the thread keeps pooling a non-deterministic value (represented through the symbol $*$).

D. Dynamic Inputs

It is worth noting that Arduino projects can rely on real-world states [8]. For example, a sensor can read temperatures, and a set of buttons might be pressed in a specific order. This behavior can be modeled as non-deterministic inputs. Each time these values are queried, the symbolic execution [25] creates a new symbol in the formula.

IV. ESBMC-Arduino

A. Architecture

The method was integrated into our prototype ESBMC-Arduino. The prototype was implemented on top of ESBMC, a BMC with a permissive license and support for the C language[3]. To support Arduino, additional changes are required to match the method (i.e., interrupts, operational models, and entry points). The full architecture can be seen in Figure 1.

In the environment setup, the verification environment was configured to accommodate the integration of ESBMC and Arduino. ESBMC was installed along with the necessary Arduino header (i.e., `Arduino.h`[4]) and the accompanying operational models in `Arduino.c`. At the time of writing, we have made available the operational models for `pinMode()`, `digitalWrite()`, and `delay()` [15].

The major addition of the prototype was in the *C Frontend* stage (see Figure 1). Specifically, the operational models also encode safe properties through the ESBMC commands (ASSERT / ASSUME), which indicate whether pre-established safety criteria are being violated. The ASSERT command

[3]https://github.com/esbmc/esbmc

[4]This header is obtained directly from the board SDK

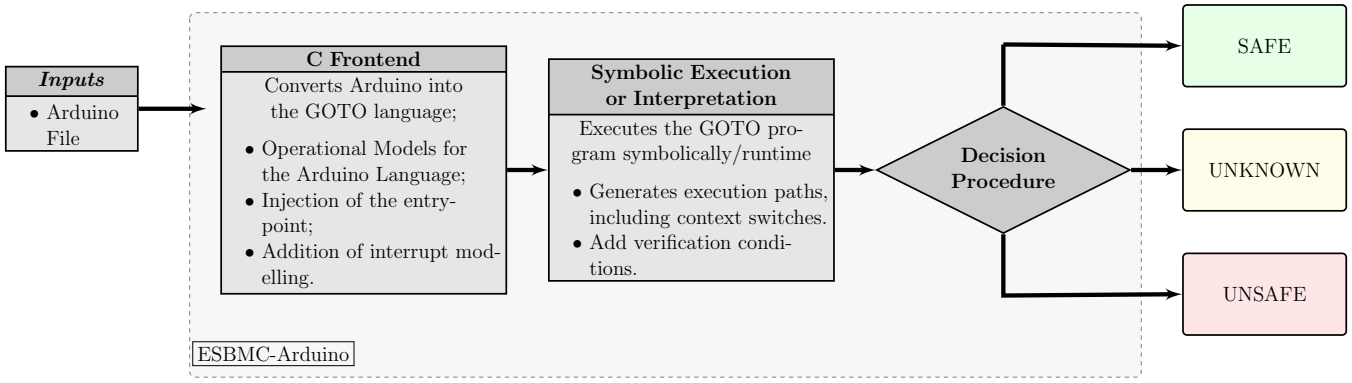

Fig. 1. The ESBMC-Arduino architecture. Starting from a supported input, the input will be sent to the C frontend of ESBMC. The C frontend will be incremented to support the Arduino-specific features and the GOTO representation. The GOTO program is symbolically executed (or interpreted) into a formula (or state), resulting in a *Decision Procedure* [26], which will evaluate whether the program is safe.

defines serious safety violations, while ASSUME handles verification pre-conditions. ESBMC success verification involves a safety check concerning the parameters indicated by the ASSERT and ASSUME programs present in the Arduino code.

B. Illustrative example

In this section, we present an illustrative example of using the Symbolic Model Checker (ESBMC) with Arduino to verify a simple Arduino program.

1) Arduino Program Description: We begin by describing the Arduino program under consideration. The program aims to control an LED connected to pin 13 on the Arduino board, making it blink at regular intervals. Below is the code for the Arduino program named "led.c":

```
1  #include <Arduino.h>
2
3  #define LED_PIN 13
4
5  void setup() {
6    pinMode(LED_PIN, OUTPUT);
7  }
8
9  void loop() {
10   digitalWrite(LED_PIN, HIGH);
11   delay(1000);
12   digitalWrite(LED_PIN, LOW);
13   delay(1000);
14 }
```

The above code follows the standard Arduino syntax and uses the `Arduino.h` library for necessary functions like `pinMode`, `digitalWrite`, and `delay`.

2) GOTO program Generation: Next, ESBMC will parse the Arduino file as a C input. If the parsing is done correctly, it will result in a GOTO program. The GOTO program is then instrumented and linked with the Arduino operational models.

3) Symbolic execution: The GOTO program is symbolically executed in the BMC engine up to the k-bound. This bound is context-aware (i.e., keeps track of thread and function interleavings) and holds a trace of the execution. This trace contains information about the program flow (i.e., assignments, assumptions, and assertions).

4) Decision Procedure: Lastly, the trace is converted into an SMT formula that uses a decidable fragment of first-order logic [27]. This formula is sent to a solver to verify the program's soundness. The SMT solver will try to find an assignment to variables that satisfies the constraints and violates at least one property. If the SMT solver finds a model, ESBMC-Arduino will generate a program counterexample: a set of assignments and the program's violated property.

V. Experimental Evaluation

A. Setup and Benchmark description

In this section, we present the benchmarks used to evaluate the integration of ESBMC with Arduino for formal verification. We assessed ESBMC-Arduino utilizing a set of benchmarks and compared its results with Arduino lint[5]. Arduino-lint is primarily geared towards scrutinizing the organization, metadata, and setup of Arduino projects, emphasizing these aspects rather than the code itself. The rules include adhering to specifications, meeting the criteria for Library Manager submissions, and following recommended practices. Unfortunately, as a consequence, considering other benchmarks was hindered as there is no other BMC tool for Arduino.

To evaluate ESBMC-Arduino, we devised a set of small-scale benchmarks as presented in Table I: columns "id" and "property" contains an identifier and the type of safety check that it violates (except "No Violation"), the column "Found" and "CE" regards wether the verifier was able to find the vulnerability and generate a Counter Example that lead to the violation, finally the "Correct Results" row summarizes the panorama of obtained results. These benchmarks were

[5]https://arduino.github.io/arduino-lint/1.2/

designed to test the correctness of ESBMC-Arduino, covering execution paths that have the potential to reveal faults, as well as execution paths that do not violate specified properties. Evaluated properties include user-defined assertions, memory management, and overflow prevention. Additionally, the benchmarks were designed to incorporate nondeterministic behaviors.

TABLE I
EXPERIMENTAL RESULTS OF ESBMC-ARDUINO AND ARDUINO LINT

Benchmark		ESBMC-Arduino		Arduino Lint	
IDs	**Property**	**Found**	**CE**	**Found**	**CE**
TC0-2	No Violation	No	N/A	No	N/A
TC3-5	Overflow	Yes	Yes	No	No
TC6-8	Assertion Fail	Yes	Yes	No	No
TC9-10	Custom Properties	Yes	Yes	No	No
Correct Results		100%		28%	

B. Results and Discussion

By comparing the verification results with traditional manual code inspection and testing, we measured the benefits of formal verification in error detection and correction. In particular, after the implemented integration, the key libraries used in the Arduino core code, `Arduino.h` and `stdint.h`, were formally verified in the ESBMC framework and did not exhibit faults. Additionally, functions like `digitalWrite()`, `pinMode()`, and `delay()` were also introduced into the verification model through their prototypes. Libraries such as `WiFi.h` and `IRtext.h`, commonly used in industrial projects, were verified but still presented errors that can be addressed in future research.

The evaluation also focused on safety analysis features of ESBMC-Arduino. Leveraging formal verification, we aimed to identify safety vulnerabilities in Arduino programs and libraries used in the industrial sector through ESBMC safety commands (ASSERT / ASSUME). Custom or non-custom properties were analyzed through test cases and compared with the Arduino lint tool. Table I presents the results, demonstrating that ESBMC-Arduino can verify benchmarks that Arduino Lint fails to, as within the benchmark analysis, we were able to confirm in TC6-8 that there was no detection of Assertion Fail property violation that exposes a safety concern, while successfully detecting safety violations for other TCs. Arduino lint did not find any bugs.

Note that Arduino lint, despite being the only available option for Arduino code verification, was not designed for safety checks; hence, it was not able to detect any property violations, which could be the reason why ESBMC-Arduino emerges as the only suitable BMC tool for Arduino code verification. safety analysis using ESBMC identified potential safety vulnerabilities in Arduino programs and libraries used in the industrial sector. We successfully detected safety flaws through rigorous code analysis. This analysis led to crucial improvements in the safety posture of embedded systems, enhancing their resilience against safety threats. The experimental evaluation results highlight the effectiveness of ESBMC-Arduino integration in verifying safety/protection properties in embedded systems. The successful formal verification of Arduino code for the industrial sector demonstrates its potential to enhance error detection, code correction, and reliability. Furthermore, ESBMC's safety analysis features improve the safety of embedded systems, making them more robust against potential cyber threats.

C. Threats to the validity

- Compilers have the potential to introduce or remove errors during the translation phase. Our methodology adopts the translation steps of processing an Arduino file as a C input (which is compiled into the GOTO language);
- ESBMC-Arduino employs operational models (OMs) to enable program verification. OMs have been developed for a specific set of frequently used Arduino libraries in industrial scenarios. These scenarios, however, can be incomplete. In our work, we only verified digital inputs;
- The test suite is limited to only sequential (no interrupt) and digital inputs.
- ESBMC-Arduino shares the same memory model used in the C/C++ analysis implemented in ESBMC. Properties that can be violated due to low resource availability may give wrong validation results.

VI. RELATED WORK

ESBMC demonstrated its effectiveness as a robust BMC tool for embedded systems verification with enhanced efficiency. Serving as a foundational SMT-based model checker for real-world C programs [15], its application follows an incremental approach to desired property satisfiability from provided counterexamples [28]. Adapting ESBMC (Efficient SMT-Based Context-Bounded Model Checker) to Arduino platforms, while promising, posed challenges arising from discrepancies between typical Arduino programming and standard C/C++.

Another direction not explored in this work is using dynamic tools such as Sanitizers and Valgrind [29]. These methods instrument the binary application, injecting assertions that can be verified during runtime. These options, however, can be very costly when used in an embedded system (Google estimates that Clang's Address Sanitizer causes a 2x slowdown in the program[6]). This is worsened if we consider that for verification purposes at least a fuzzer would be needed [30], [31], resulting in multiple executions.

VII. CONCLUSIONS AND FUTURE WORK

We present and assess ESBMC-Arduino, a groundbreaking software model checker engineered specifically for Arduino code. In our evaluation, we juxtapose and surpass the capabilities of the Arduino lint verification tool. Our approach not only identifies bugs more effectively but also provides corresponding counterexamples. While ESBMC-Arduino is in its nascent developmental stages, the outcomes attained thus far are highly promising. Our current thrust revolves around

[6]https://github.com/google/sanitizers/wiki/AddressSanitizerPerformanceNumbers

achieving comprehensive 100% coverage for the Arduino language, with a priority on the essential safety criteria that ESBMC addresses. Furthermore, we are exploring incorporating the k-induction proof method into Arduino [17], a pivotal step toward enabling safety proofs in Arduino's unbounded loop nature. We envision extending our endeavors by implementing customized safety criteria tailored to the Arduino platform. On the other hand, another future work would be to confirm whether the violations (and proofs) found by our prototype are valid. This could be done by expanding the approach used for C validation (i.e., witness validation [32]). Another idea, is to combine BMC with other static analysis techniques such as Abstract Interpretation [33]. This would enable a framework for static analysis of Arduino code.

ACKNOWLEDGEMENTS

This work was developed with the support of Cal-Comp Electronic by the R&D project of the Cal-Comp Institute of Technology and Innovation in Manaus/AM.

REFERENCES

[1] A. e. a. SOUSA, "Elicitação e especificação de requisitos em sistemas embarcados: Uma revisão sistemática," *CIbSE*, 2015.

[2] L. C. Cordeiro, E. B. de Lima Filho, and I. V. de Bessa, "Survey on automated symbolic verification and its application for synthesising cyber-physical systems," *IET Cyper-Phys. Syst.: Theory & Appl.*, vol. 5, no. 1, pp. 1–24, 2020. [Online]. Available: https://doi.org/10.1049/iet-cps.2018.5006

[3] E. Barros and S. Cavalcante, "Introdução aos sistemas embarcados," *Artigo apresentado na Universidade Federal de Pernambuco-UFPE*, p. 36, 2010.

[4] G. W. Denardin and C. H. Barriquello, *Sistemas operacionais de tempo real e sua aplicaçao em sistemas embarcados*. Editora Blucher, 2019.

[5] L. Danta, *Transformação digital e inovação*. Editora Senac São Paulo, 2021.

[6] F. R. Monteiro, M. Garcia, L. C. Cordeiro, and E. B. de Lima Filho, "Bounded model checking of C++ programs based on the qt cross-platform framework," *Softw. Test. Verification Reliab.*, vol. 27, no. 3, 2017. [Online]. Available: https://doi.org/10.1002/stvr.1632

[7] C. N. Domingos, D. A. dos Santos, W. Grignani, and D. R. de Melo, "Plataforma de integração de componentes para sistemas embarcados em aplicações espaciais," *Anais do Computer on the Beach*, vol. 14, pp. 444–446, 2023.

[8] S. Monk, *Programação com Arduino: começando com Sketches*. Bookman Editora, 2017.

[9] M. M. d. Santos *et al.*, "Programação orientada a agentes bdi em sistemas embarcados," 2022.

[10] V. V. Ballarini *et al.*, "Proposição de um sistema de automação responsável pelo monitoramento das variáveis em ambiente de armazenamento de produtos agrícolas," 2021.

[11] P. H. M. Pazini and L. F. B. Lopes, "Automação residencial de baixo custo com utilização de sistema desenvolvido em arduino," *DIVERSITÀ: Revista Multidisciplinar do Centro Universitário Cidade Verde*, vol. 3, no. 1, 2017.

[12] M. Margolis, B. Jepson, and N. R. Weldin, *Arduino cookbook: recipes to begin, expand, and enhance your projects*. O'Reilly Media, 2020.

[13] E. Ferroni, H. Vieira, J. Nogueira, R. Santos, R. Lemos, and T. Rodrigues, "A plataforma arduíno e suas aplicações," *Revista da UI_IPSantarém*, 2015.

[14] L. C. Cordeiro, C. Mar, E. Valentin, F. Cruz, D. Patrick, R. S. Barreto, and V. Lucena, "An agile development methodology applied to embedded control software under stringent hardware constraints," *ACM SIGSOFT Softw. Eng. Notes*, vol. 33, no. 1, 2008. [Online]. Available: https://doi.org/10.1145/1344452.1344459

[15] M. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole, "Esbmc 5.0: an industrial-strength c model checker," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 888–891.

[16] F. R. Monteiro, M. R. Gadelha, and L. C. Cordeiro, "Model checking C++ programs," *Softw. Test. Verification Reliab.*, vol. 32, no. 1, 2022. [Online]. Available: https://doi.org/10.1002/stvr.1793

[17] O. M. Alhawi, H. Rocha, M. R. Gadelha, L. C. Cordeiro, and E. B. de Lima Filho, "Verification and refutation of C programs based on k-induction and invariant inference," *Int. J. Softw. Tools Technol. Transf.*, vol. 23, no. 2, pp. 115–135, 2021.

[18] L. H. Sena, I. V. de Bessa, M. Y. R. Gadelha, L. C. Cordeiro, and E. Mota, "Incremental bounded model checking of artificial neural networks in CUDA," in *IX Brazilian Symposium on Computing Systems Engineering, SBESC 2019, Natal, Brazil, November 19-22, 2019*. IEEE, 2019, pp. 1–8.

[19] D. Basin, *The cyber security body of knowledge*. University of Bristol,, ch. Formal Methods for Security, version..[Online …, 2021.

[20] F. Accerboni and M. Sartor, "Iso/iec 27001," in *Quality Management: Tools, Methods, and Standards*. Emerald Publishing Limited, 2019, pp. 245–264.

[21] M. R. Gadelha, R. S. Menezes, and L. C. Cordeiro, "Esbmc 6.1: automated test case generation using bounded model checking," *International Journal on Software Tools for Technology Transfer*, vol. 23, pp. 857–861, 2021.

[22] M. Dangl, "Witness-based validation of verification results with applications to software-model checking," Ph.D. dissertation, lmu, 2022.

[23] N. Chong, B. Cook, J. Eidelman, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle, "Code-level model checking in the software development workflow at amazon web services," *Software: Practice and Experience*, vol. 51, no. 4, pp. 772–797, 2021.

[24] L. C. Cordeiro, "Smt-based bounded model checking of multi-threaded software in embedded systems," Ph.D. dissertation, University of Southampton, UK, 2011. [Online]. Available: http://eprints.soton.ac.uk/186011/

[25] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using sat procedures instead of bdds," in *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, 1999, pp. 317–320.

[26] D. Kroening and O. Strichman, *Decision procedures*. Springer, 2016.

[27] L. C. Cordeiro, B. Fischer, and J. Marques-Silva, "Smt-based bounded model checking for embedded ANSI-C software," in *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*. IEEE Computer Society, 2009, pp. 137–148. [Online]. Available: https://doi.org/10.1109/ASE.2009.63

[28] M. Li, "Counterexample-guided optimisation applied to mobile robot path planning," 2021.

[29] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[30] K. M. Alshmrany, R. S. Menezes, M. R. Gadelha, and L. C. Cordeiro, "Fusebmc: A white-box fuzzer for finding security vulnerabilities in C programs (competition contribution)," in *24th International Conference on Fundamental Approaches to Software Engineering (FASE)*, ser. LNCS, E. Guerra and M. Stoelinga, Eds., vol. 12649. Springer, 2021, pp. 363–367. [Online]. Available: https://doi.org/10.1007/978-3-030-71500-7_19

[31] K. M. Alshmrany, M. Aldughaim, A. Bhayat, and L. C. Cordeiro, "Fusebmc v4: Smart seed generation for hybrid fuzzing - (competition contribution)," in *25th International Conference on Fundamental Approaches to Software Engineering (FASE)*, ser. LNCS, E. B. Johnsen and M. Wimmer, Eds., vol. 13241. Springer, 2022, pp. 336–340. [Online]. Available: https://doi.org/10.1007/978-3-030-99429-7_19

[32] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer, "Witness validation and stepwise testification across software verifiers," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 721–733.

[33] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977, pp. 238–252.