A Novel Approach in Proving Unreachable Paths in Hardware-dependent Software

Bryan Olmos*†, Wolfgang Kunz †, Djones Lettnin*
*Infineon Technologies AG - †Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau

***Abstract*—Achieving reliable code coverage is of fundamental importance in embedded systems development, especially in compliance with automotive standards like ISO26262. Firmware designs, with their hardware interaction, require thorough testing to minimize the risk of undetected bugs. Achieving 100% code coverage is challenging. This paper adds a step in the development process to detect unreachable paths in the initial phase of the development process. This paper proposes a methodology using MC/DC (Modified Condition Decision Coverage) with formal verification to analyze code functions individually within a time-bound. The paper presents the common bugs found in 7 firmware designs. For example in a FW design with 16k lines of code, 51 of 613 paths were unreachable, with 25 paths being due to bugs. Before addressing initial bugs, 1332 potential software weaknesses were identified using formal verification, which increased to 1370 after correction. It shows the benefits of including the detection of unreachable paths and avoiding, later corrections during integration.**

***Index Terms*—Formal Verification, Firmware Verification, Code Coverage**

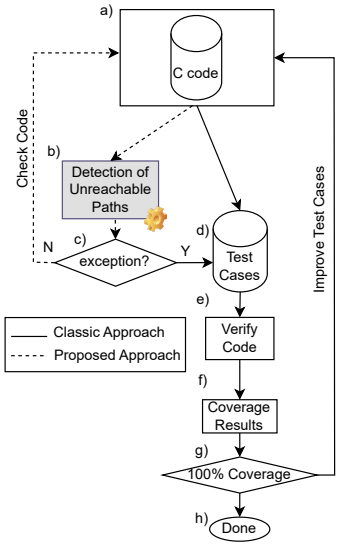

Fig. 1. Tool chain for code coverage

I. Introduction

One of the main challenges in software testing for safety-critical systems is to decide when the code has been tested enough [1]. For this reason, safety embedded systems, such as road vehicles, are required to comply with the safety standard ISO 26262-6 to ensure safety, quality, reduction of liability risks and enhancement of the customer confidence about software requirements [2]. This standard requires that the code and its test cases are analyzed concerning metrics such as statement coverage, branch coverage and MC/DC—for all ASIL(Automotive Safety Integrity Levels). Additionally, the cost of repairing C code in embedded system designs increases significantly as defects are detected later in the development timeline [3], creating an incentive for the manufacturer to detect bugs earlier.

Previous studies have shown how test cases can be obtained in an automated or non-automated way [4]. Once the source code is available, steps a), d), e), f), g), and h) of Figure 1 can be followed for a classic approach. Unit tests are used to verify the code, and obtain the coverage results, not reaching 100% code coverage implies that the test cases require improvement. Using simulation and branch coverage, for instance, would require a thorough search for edge cases to cover specific branches [4]. However, there are situations where some branches cannot be covered due to contradictions inside the design or in the case of firmware, events related to hardware [5]. Furthermore, in the case of using bounded model checkers, the reached states are limited to the loop-bound for verification —loop-bound means the number of times a loop in software is unwinded before its analysis. Thus, the design could be overconstrained making the results not sound —an unsound model appears correct according to formal verification but fails to represent the behaviour of a system leading to potential failures [6]. In all these situations, coverage will not change with any approach. For this reason, an initial analysis of unreachable paths is necessary prior to the verification process. It will save time not only in obtaining code coverage but also avoid re-running the verification flow in case bugs and software weaknesses reside in these unreachable paths.

For this reason, this paper proposes to begin with the sequence a), b), c) of Figure 1 and will focus on block b) which is the detection of unreachable paths in firmware designs. In case the unreachable path is intended (See Subection IV-5), this is considered an exception and the code does not need to be corrected, as shown in the step c). This detection can be done by individual functions using the tool CBMC [5]. Nevertheless, a functional analysis of the code coverage becomes unfeasible when the size of the code or the loop-bound of the verification begins to increase. This is because one of the main challenges of model checking is the state explosion problem [7]. To make the approach scalable for the analysis of huge codes, a pessimistic approach is used considering the worst-case scenario of a bounded model checker, which means, that at some point in time, the analysis will fail due to the size of the function under analysis. For this reason, a time-bound t is assigned to get the unreachable paths of each function as explained in Section III.

This approach identifies unreachable paths of firmware de-

signs based on C code. For example, analyzing a source code during the development phase —with 16k lines of code, 51 of 613 paths were identified as unreachable in a runtime of 31 min by using the methodology described in Section III. Of these 51 paths, 25 were due to bugs and 26 were intended. Before correcting these initial bugs, 1332 potential software weaknesses were detected with a runtime of 3 h and 14 min and after correction 1370 were detected in 3 h and 24 min. The main contributions of this paper are:

- Introducing the detection of unreachable paths in the development process proposing a new methodology based on formal verification of each function in the source code and a time-bound for the analyzed functions.
- Automating the methodology and presenting the results for branch coverage and MC/DC for 7 industrial firmware designs.
- Categorization of the most common bugs found in 7 industrial designs and proving the methodology by detecting software weaknesses.

This paper is organized as follows: In Section II, the main concepts related to code coverage, CBMC and related work are discussed. Section III introduces the methodology. The main causes of unreachable paths are presented in IV. Section V shows the experimental results of 7 industrial designs. Conclusions and future work are presented in Section VI.

II. Background

A. CBMC and MC/DC Code Coverage

CBMC is a tool for formal verification of ANSI-C programs using BMC. It converts the code into a mathematical representation and solves it using formal methods [8] [9]. The main advantages of using CBMC for formal verification of C code are a) Ensuring the absence of bugs through an exhaustive analysis of the code; b) Detection of software weaknesses of the CWE (Common Weakness Enumeration) community [10] [11]. c) Providing code coverage metrics such as branch coverage and MC/DC. Listing 1 shows an example of the assertions used by CBMC to get the code MC/DC coverage.

```
int test(int a, int b) {
// Built-in assertions CBMC for MC/DC:
assert (!(a > 0) && !(b < 0)) // MC/DC independence
    condition
assert (!(a > 0) && b < 0)    // MC/DC independence
    condition
assert (a > 0 && !(b < 0))    // MC/DC independence
    condition
assert (!(a > 0 || b < 0))    // decision is false
assert (a > 0 || b < 0)       // decision is true
assert (!(a > 0))             // condition is false
assert (a > 0)                // condition is true
assert (!(b < 0))             // condition is false
assert (b < 0)                // condition is true
    if ( a > 0 || b < 0) {
        return 1;
    }
}
```

Listing 1. Example MC/DC coverage

Branch coverage states that each branch direction must be traversed at least once, for example, the condition of an if-else statement must be evaluated for true and false [12]. MC/DC criterion enhances the condition/decision coverage criterion by requiring that each basic condition be shown to independently affect the outcome of the decision [13], where a basic condition is an atomic Boolean valued expression that cannot be broken into Boolean sub-expressions [14]. Concerning MC/DC, CBMC adds built-in assertions to analyze the code coverage.

B. Related Work

Previous works have focused on code coverage metrics based on test cases. Williams et al. [15] propose exhaustive branch coverage using concolic test generation for C code, analyzing designs up to 340 LoC and 128 branches. Gay et al. [4] compare direct and indirect branch coverage for Java, evaluating their effectiveness in fault detection. Ahishakiye et al. [16] present MC/DC results using a trace-based approach. For embedded systems, Shen et al. [17] introduce Tardis, an OS fuzzer detecting abnormal behavior by repeatedly feeding varied test cases. Unlike these approaches, we aim to detect unreachable paths before running test cases. Alavizadeh et al. [18] classify unreachable code as dead code without proposing a specific detection method. Dong et al. [19] detect unreachable paths in Java by partitioning programs into sequential, conditional, and loop statements. Lee and Böhme [20] use statistical methods to estimate reachability probability, showing the benefits of incorporating structural information. None of these works consider formal methods for hardware constraints. The concept of robust reachability is introduced by [21], focusing on scenarios where bugs are influenced by controlled inputs. In contrast, this paper considers all inputs as uncontrolled. Regarding soundness, Cousot and Cousot [22] have advanced static analysis tools through abstract interpretation, which uses mathematical abstractions to reduce the state space explored by model checkers. We focus on enhancing soundness in FW designs, which have unique challenges due to established bounds and hardware constraints.

C. Contribution

As shown in the previous studies, estimating the number of reachable paths is a challenging task, main studies have focused on the analysis of high-level software. However, the effect in firmware has not been analyzed. To contribute to filling this gap, we proposed:

- New Methodology to Detect Unreachable Paths:
  - Use of MC/DC coverage and formal verification in reachability analysis for firmware designs.
  - The methodology partitions the analysis of the source code into the analysis of its functions, enabling a more targeted and systematic approach.
  - The use of a time-bound is proposed for the verification of each function ensuring that the analysis remains bounded and efficient.
- Analysis of Firmware designs:
  - Seven FW designs developed in C are analyzed.

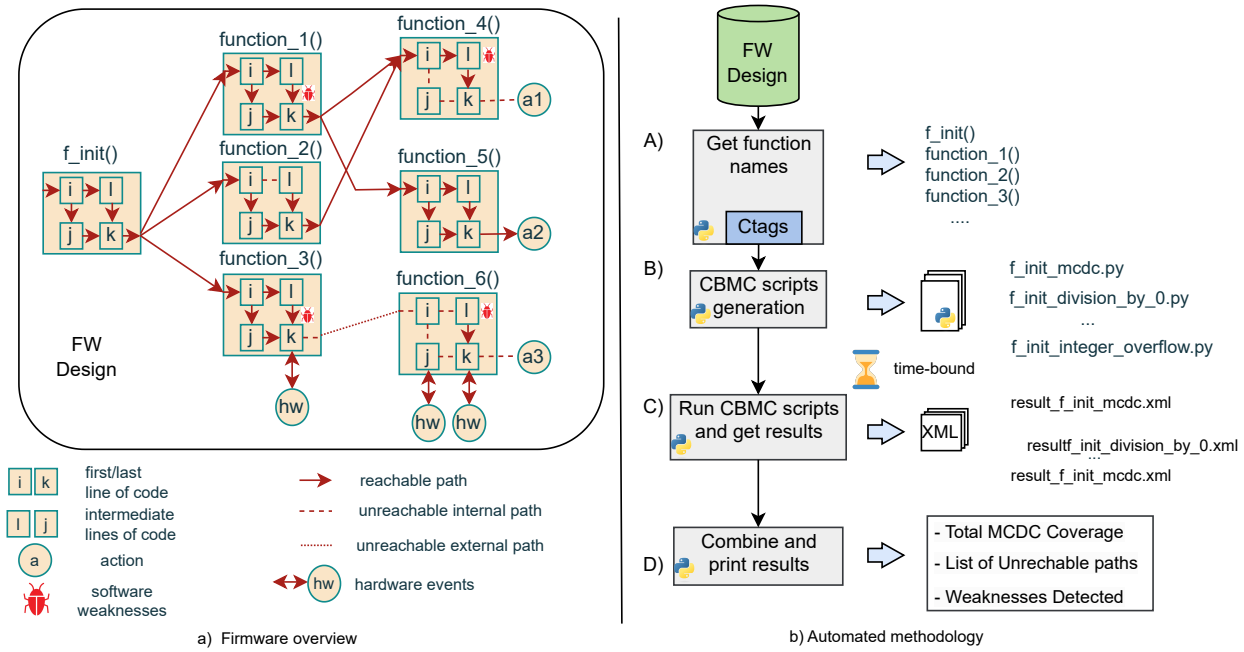

Fig. 2. Methodology for the detection of unreachable paths and weaknesses detection based on CBMC and Python scripts in FW designs

  - The trade-off between proposed time-bound, loop-bound and code coverage is presented.
  - Identification and categorization of main bugs in FW, offering valuable insights into the potential impact of unreachable paths on firmware reliability.
- Automated Framework:
  - The methodology is automated based on CBMC and Python scripts.

III. Methodology

As shown in Figure 2a), FW designs usually consist of a set of functions that interact with HW, for example, the FW is planned to execute some action over the HW or wait for some hardware events to continue. If these events never occur, the FW will not be able to continue its execution, for example, $function\_3()$ in Figure 2a) is waiting for an event and will not reach $function\_6()$ until that event occurs. If we use static analysis and do not consider these HW events, $function\_6()$ will be never reached and some potential weaknesses will not be detected, additionally, action 3 (a3) will never occur. Furthermore, some internal paths in the functions could not be reached due to contradictions in the design, and wrong assumptions about the functions accessing the register values or constant values. The main 8 causes for unreachable paths detected during our research are described in Section IV. To detect these unreachable paths, we propose the use of CBMC to get MC/DC coverage of all functions without constraints following the steps of Figure 2b).

A. Get Function Names

All functions of the C code must be analyzed in the FW design. If only the main function is analyzed, then the tool will not consider all code after an unreachable path. In this situation, it is feasible to detect an initial unreachable path but the process of running the tool and detecting each path becomes tedious and hard to debug especially for large designs. To get the function names, a Python script is used with the library Ctags [23]. This library returns a list with all the function names.

B. CBMC Scripts Generation

A second Python script is used to generate other Python scripts to run CBMC for all functions, an example of MC/DC coverage is shown in Listing 2. Note that in line 7, a signal alarm is set to 40, which means the proof analysis will run out in 40 seconds; this time-bound is necessary because some functions will contain infinite loops or get stuck in an unreachable path and the execution will not stop. In these cases, some constraints are needed for the analysis. The trade-off analysis between the time-bound and the code coverage is presented in Section V-A. Additionally, scripts for weakness detection with CBMC were generated. They show the analysis of the FW designs with and without unreachable paths in Section V.

```
1  import subprocess
2  import signal
3  def handler_function(signum, frame):
4      print("Alarm activated! Terminating process ...")
5      exit(1)
6  signal.signal(signal.SIGALRM, handler_function)
7  signal.alarm(40) # time-bound
8  file = open(" results_function_test .xml", "w")
9  subprocess.check_call(['cbmc',
10 '-I', ' source_folder ', # add all the folders
11 ' file_function_test .c', # add all the C files
12 '--unwind', '20', # loop-bound
13 '--32',
14 '--cover', 'mcdc', # coverage MC/DC
15 '--xml-ui', # results format XML
16 '--function', ' function_test ' # function name
17 ], stdout=file)
18 signal.alarm(0)
```

Listing 2. Script to get the MCDC coverage with CBMC for Listing1

C. Run CBMC and Get Results

In this step, a script calls all the previously generated scripts and returns an XML file with the MC/DC Coverage results. Additionally, the results were filtered to return a list with all the possible unreachable paths per function. The runtime depends mainly on the size of the code as shown in Section V.

D. Combine and Print Results

In this step, we combine the results of all the functions and print the main results of the total MC/DC Coverage, the list of unreachable paths per function, and the detected weaknesses. These results are automatically summarized in an XLSX file.

IV. Description of Detected Unreachable Paths

In this section, the most commonly detected unreachable paths in 7 FW designs are analyzed.

1) Wrong FW Assumptions: Wrong assumptions in the FW can lead to unreachable paths. Listing 3 shows the function *get_factor* which takes a 16-bit parameter called *input_value* (line 1) and returns the value called *factor* (line 16). A wrong assumption here is that all if-condition statements could be true and false for a given input_value in the range of $[0, 65535]$. However, the else-condition in line 13 is not reachable due to the intermediate operations (in lines 3-5) —the maximum value that *condition* can be is 5715.

```
1  uint16_t  get_factor (uint16_t  input_value)
2  {
3      uint16_t  factor  = 0u;
4      uint16_t  condition  = input_value  >> 9u;
5      condition  = condition  * 45u;
6
7      if (condition  < 256u)
8      {            factor  = 1u;
9      }
10     else  if  (condition  < 6000u)
11     {            factor  = 2u;
12     }
13     else
14     {           factor  = 3u;
15     }
16     return  factor ;
17 }
```

Listing 3. Wrong FW Assumption - factor = 3 in line 14 is not reached

2) Unnecessary Operations: Some operations which are not reached could reduce the performance of the code.

```
1  uint16_t  calculate_operation (uint16_t  a,  uint16_t  b)
2  {
3      uint16_t  i  = 0U;
4      if (a>= (b << 1U))
5      {
6      while (((a>>i) >= ((b << 1)>>i)) && (a<<1U)){
7          i  = i  + 1;
8          }
9      }
10     return  i;
11 }
```

Listing 4. If $if(a >= (b << 1U))$ is true, then the condition $(a >= (b << 1))$ in the while loop is always true

Listing 4 shows one example, where the function *calculate_operation* returns a value i (line 10) after the calculation of a while loop (line 6). In this case, all the lines of the code are reachable. However, after getting the initial MC/DC code coverage, the results show that the first condition of the while loop $(a >> i) >= ((b << 1) >> i)$ could never be false. This is because if the first if-statement (line 4) holds, then the while loop is always true, making it redundant. For this reason, this first condition can be corrected or removed from the while loop reducing the number of operations required to evaluate it. This cannot be detected even using line coverage or branch coverage and shows the necessity to have MC/DC coverage.

3) Overconstrained Design: Some paths can be unreachable due to intrinsic technical aspects. Listing 5 shows the snippet of a function to initialize the registers of an Electronic Control Unit (ECU). Its main features are:

1) *function_init()* loads the default values in the registers by calling $load\_and\_verify\_virgin\_data()$ (line 4) and verifies their integrity by comparing if the expected *CRC* remainder is 0x0044 (line 7). If the remainder calculated by the HW is different it sets the register *CRC_ERROR* to *TRUE* (line 14).
2) The function *get_virgin_registers()* returns the number of found virgin registers (line 5).
3) If there is no CRC error and all the registers are virgin then set the register *VIRGIN_ERROR()* to *TRUE* (line 12).

```
1  int  function_init (){
2  int   total_registers  = 32;
3  int   virgin_registers  ;
4  load_and_verify_virgin_data ();
5   virgin_registers  =  get_virgin_registers  ();
6
7  if (CRC_HW_DATA() != 0x0044) {
8      CRC_ERROR_SET(TRUE);
9  }
10 else {
11     if  ( virgin_registers  ==  total_registers  ) {
12         VIRGIN_ERROR_SET(TRUE);
13         }
14     }
15 }
```

Listing 5. Function to verify integrity of data loaded into registers

In this case, the code is implemented as expected. However, line 12 is never reached. This is because the else statement (line 10) is reached only if *CRC_HW_DATA() == 0x0044* and line 11 requires that all the registers are virgin. Both conditions cannot be true at the same time. It will happen only if the default values are equal to the virgin values, which is not the intended implementation.

4) Wrong Bit-Width Register: A function in the design can access only a certain number of bits and this number could be incorrectly assumed at the moment of writing the code.

5) Intended Unreachable Paths: Some paths are unreachable due to the intrinsic design of the code, for instance, Listing 6 shows the *f_under_test()* (line 2) always returning *READY* (line 4). The reason could be that the logic of the function is not defined at this stage of development or is a hard-coded value.

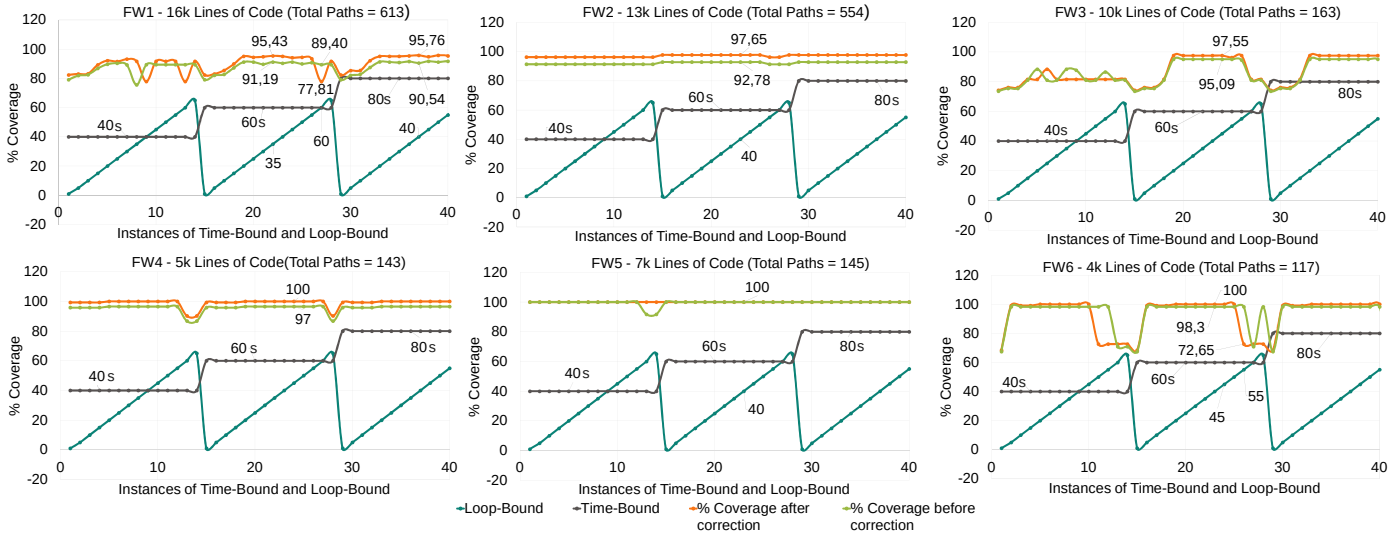

Fig. 3. % Code coverage before and after the correction of the unreachable paths

6) Missing HW Event: FW is waiting for an event or a signal value before continuing.

7) Missing Sequence of HW Events: FW is waiting indefinitely for a sequence of events during its operation [5].

```
#define READY 1 // Constant
int f_under_test () {
    // some code
    return READY; // Return always READY
}

int main() {
    if ( f_under_test () == READY) {
        // some code
    }
    return 0;
}
```

Listing 6. The predefined return value is always READY

8) Unreachable Paths due to Loop-Bound or Infinite Loops: This case is related to the loop-bound chosen for the formal verification or the presence of infinite loops that also require directives.

V. Experimental Results

A. Time-Bound, Loop-Bound, and Code Coverage

Figure 3 shows the difference in code coverage before and after correcting the unreachable paths in relation to the loop-bound and time-bound. In FW1 with a loop-bound of 35 and a time-bound of 60 s per execution of each function, the initial code coverage is 91,19% and after addressing unreachable paths it improves to 95,43%. However, if the loop-bound increases to 60 the initial code coverage is 89,4% and after the correction it decreases to 77,81%. This is because of the trade-off between the loop-bound and coverage if the loop-bound increases the time-bound also needs to increase otherwise CBMC does not have enough time to analyze the function. If the time-bound increases to 80 s, then the code coverage after correction is always higher than the initial. In FW2, code coverage after the correction of unreachable paths is higher than the initial code coverage for a loop-bound of 40, 60 and 80 s. In FW1, FW2 and FW3, it is not feasible to reach 100% of code coverage. This is because some of the paths were unreachable by design as described in Section IV-5. FW4, FW5 and FW6 were able to reach 100%. FW7 reached 99,68% due to containing an intended unreachable path. Table I shows the classification of unreachable paths in the FW designs. Most unreachable paths were detected in FW1. This is mainly because FW1 has more lines of code than other designs, also it was analyzed in the initial stage of the development phase. FW5 does not contain unreachable paths, because it was analyzed on a stable FW version. The average time to get the total code coverage for a specific time-bound and loop-bound depends on the code size. Thus, FW5 with 7k is analyzed in only 2,2 min; however, FW1 with almost double the lines of code takes 12x as long (26,8 min) due to the state explosion problem of model checking.

TABLE I
Detected Unrechable paths in FW Designs

Unreachable Path Type	FW1	FW2	FW3	FW4	FW5	FW6	FW7
Wrong FW Assumptions	0	0	0	1	0	0	0
Redundant Operations	1	0	0	0	0	0	0
Overconstrained design	5	1	2	1	0	3	0
Wrong bit width register	2	0	0	0	0	0	0
Intended Unreachable Paths	14	3	2	0	0	0	1
Missing HW Event	2	6	0	0	0	0	0
Sequence of HW Events	0	1	0	0	0	0	0
Infinite while loop	20	0	0	0	0	2	0
Total Unreachable Paths	44	11	4	2	0	5	1
Effective Lines of Code	16k	13k	10k	5k	7k	4k	9k
Average Runtime Per Instance (min)	26,8	7,2	7,4	1,0	2,2	1,4	3,9

The 3 most common causes are infinite loop, intended unreachable paths and overconstrained design, as seen in Figure 4.

Figure 5 compares the detected weaknesses with CBMC after and before the correction of unreachable paths. More weaknesses were detected in FW1, and FW2 after the correction; for example, 38 more weaknesses were detected in FW1 and 78 more in FW2. The weaknesses detected in the other FW were the same in both cases. This is because the unreachable paths were not related to code that introduced more weaknesses. For this analysis, the loop-bound was 70 because this loop-bound covered all paths of the FW designs after including constraints for the case of infinite loops only found in FW1 and FW6 as shown in Table I. The average runtime for weakness detection was almost the same for both cases in all FW designs and

depended on the size of the code. The longest runtime was 199 min for FW1 with 16k lines of code and the shortest one was 9 min for FW5.

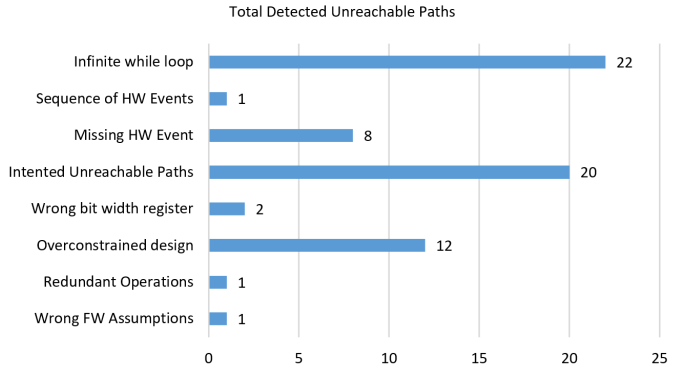

Fig. 4. Weaknesses Detection Classification

B. Weaknesses Detection Results

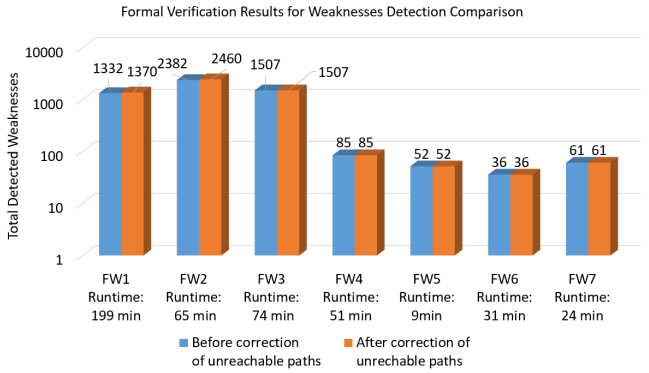

Fig. 5. Weaknesses detected before and after correction of unreachable paths

VI. Conclusion

Firmware designs were analyzed for up to 16k lines of code. All of them were analyzed for weakness detection to show the importance of including the detection of unreachable paths. 6 of the 7 designs under verification contained unreachable paths. After their correction, 38 and 78 additional weaknesses were detected in FW1 and FW2 in 3h and 24 min and 1h and 5 min, respectively. This shows a necessity to include reachability analysis even if the firmware designs are analyzed using formal verification to ensure soundness in the results. The proposed methodology was able to catch all unreachable paths using formal verification to obtain the MC/DC based on a time-bound —it was introduced in the methodology to avoid the tool never stopping due to missing constraints or infinite loops. Additionally, the necessity of including MC/DC to detect problems concerning redundant operations which cannot be detected by branch or line coverage was shown. As proposed in the introduction, this initial code coverage provides the maximum code coverage that we will be able to detect in our code. It helps to avoid testing the code trying to reach paths that cannot be detected. Finally, the reliability of the firmware was increased due to use the of formal verification and the detection of unreachable paths in an early stage of the development process. Future work will include the analysis of large firmware designs.

Acknowledgment

This work has been developed in the project VE-VIDES (project label 16ME0243K) which is partly funded within the Research Programme ICT 2020 by the German Federal Ministry of Education and Research (BMBF).

References

[1] C. Hobbs, “Embedded software development for safety-critical systems; second edition,” *Embedded Software*.

[2] “IS026262 Road vehicles – Functional safety, Part 1: Vocabulary, Part 6: Product development at the software level.” International Organization for Standardization (ISO), Standard, 2016.

[3] A. Shaout and D. Breton, “Validation and Verification For Embedded System Design “An Integrated Testing Process Approach”,” *International Journal of Computer & Organization Trends*, vol. 10, no. 1, Jul. 2014.

[4] G. Gay, “To call, or not to call: contrasting direct and indirect branch coverage in test generation,” in *Proceedings of the 11th International Workshop on Search-Based Software Testing*. Gothenburg Sweden: ACM, May 2018, p. 43–50. [Online]. Available: https://dl.acm.org/doi/10.1145/3194718.3194719

[5] B. Olmos, S. Sainath, D. Lettnin, and W. Kunz, “Automating the Formal Verification of Firmware: A Novel Foundation and Scalable Methodology,” *DVConUSA 2024*.

[6] Y. A. Manerkar, “Progressive automated formal verification of memory consistency in parallel processors, dissertation princeton university,” 2021.

[7] E. M. Clarke, *Model Checking and the State Explosion Problem*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 7682, p. 1–30.

[8] “Bounded Model Checking for Software,” http://www.cprover.org/cbmc/, accessed: 2023-07-6.

[9] E. Clarke, D. Kroening, and F. Lerda, “A tool for checking ANSI-C programs,” *Lecture Notes in Computer Science*, vol. 2988, pp. 168–176, 01 2004.

[10] “Common Weaknesses Enumeration,” https://cwe.mitre.org/, accessed: 2023-07-6.

[11] M. Byun, Y. Lee, and J.-Y. Choi, “Analysis of software weakness detection of CBMC based on CWE,” in *2020 22nd International Conference on Advanced Communication Technology (ICACT)*, 2020, pp. 171–175.

[12] G. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*. Wiley, 2011.

[13] H. Kelly J., V. Dan S., C. John J., and R. Leanna K., “A practical tutorial on modified condition/decision coverage,” Tech. Rep., 2001.

[14] M. W. Whalen, M. P. E. Heimdahl, and I. J. D. Silva, “Efficient test coverage measurement for mc/dc,” 2013.

[15] N. Williams, “Towards exhaustive branch coverage with pathcrawler,” in *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*. Madrid, Spain: IEEE, May 2021, p. 117–120.

[16] F. Ahishakiye, S. Jaksic, V. Stolz, F. D. Lange, M. Schmitz, and D. Thoma, “Non-Intrusive MC/DC Measurement Based on Traces,” in *2019 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. Guilin, China: IEEE, Jul. 2019, p. 86–92.

[17] Y. Shen, Y. Xu, H. Sun, J. Liu, Z. Xu, A. Cui, H. Shi, and Y. Jiang, “Tardis: Coverage-guided embedded operating system fuzzing,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, p. 4563–4574, Nov. 2022.

[18] A. Alavizadeh, “Classifying dead code in software development,” Bachelor of Science in Computer Science and University Honors, Portland State University, Mar. 2022. [Online]. Available: https://pdxscholar.library.pdx.edu/honorstheses/1168

[19] Y. Dong, S. Wang, L. Zhang, X. Liu, and S. Liu, “Automatic detection of infeasible paths in large-scale program based on program summaries,” 2024. [Online]. Available: https://www.ssrn.com/abstract=4760606

[20] S. Lee and M. Böhme, “Statistical reachability analysis,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. San Francisco CA USA: ACM, Nov. 2023, p. 326–337. [Online]. Available: https://dl.acm.org/doi/10.1145/3611643.3616268

[21] G. Girol, B. Farinier, and S. Bardin, *Not All Bugs Are Created Equal, But Robust Reachability Can Tell the Difference*, 07 2021, pp. 669–693.

[22] P. Cousot, *Abstract Interpretation: From 0, 1, To ∞*. Singapore: Springer Nature Singapore, 2023, pp. 1–18.

[23] Universal Ctags, “Universal Ctags Repository,” https://github.com/universal-ctags/ctags, accessed: 19-06-2024.