A PSEUDO-EXHAUSTIVE SOFTWARE TESTING FRAMEWORK FOR EMBEDDED DIGITAL DEVICES IN NUCLEAR POWER

Athira Varma Jayakumar[1], Richard Kuhn[2], Brandon Simons[1], Aidan Collins[1], Smitha Gautham[1], Richard Hite[1], Raghu Kacker[2], Abhi D. Rajagopala[1], and Carl Elks[1]

[1]Virginia Commonwealth University, Richmond, VA, USA
[2]National Institute of Standards and Technology, Gaithersburg, MD, USA

jayakumarav@vcu.edu, d.kuhn@nist.gov, simonbj@mymail.vcu.edu, collinsag@mymail.vcu.edu, gauthamsm@vcu.edu, hiterd@mymail.vcu.edu, raghu.kacker@nist.gov, rajagopalaad@vcu.edu, crelks@vcu.edu

[Digital Object Identifier (DOI) placeholder]

ABSTRACT

The major challenge faced by the nuclear industry related to software testing of digital embedded devices is the identification of practical software (SW) testing solutions that provide a strong technical basis and are at the same time effective in establishing credible evidence of software common cause failures (SCCF) reduction. Towards this effort, we conducted a systematic empirical study on pseudo-exhaustive SW testing methods for embedded digital devices. In this paper, we describe the realization of a testbed for conducting an automated pseudo-exhaustive software testing on embedded digital devices and the intricate interactions between the multiple software tools involved in the workflow. The collected results and derived findings confirm the ability of the automated pseudo-exhaustive testing methodology to economically exercise the interaction input/state space in a systematic, rigorous, and comprehensive manner.

Key Words: Pseudo-exhaustive testing, *t*-way combinatorial testing, Nuclear I&C, Smart sensor

1 INTRODUCTION

One of the Modernization pathways for the US Nuclear Industry is toward all-digital solutions for the instrumentation and controls (I&C) operations in Nuclear Power plants. Of specific interest is the use of advanced software-based embedded digital devices (EDDs) that have the potential to substantially increase plant reliability and performance operations. EDDs are typically edge devices like sensors, transmitters and actuators. That said, there is the concern in the nuclear community that latent software common cause failures (SCCF) may manifest in these devices and potentially inhibit or degrade safety functions. Traditionally, diversity and defense-in-depth architectural methods for I&C systems have been the norm for addressing vulnerabilities associated with SCCF. However, these methods incur increased system and plant integration complexity along with high implementation and maintenance costs. To this end, there has been significant interest in the regulatory community towards finding cost-effective design assurance and testing methods for critical software systems that could provide options beyond diversity and defense-in-depth practices.

The major challenge faced by the nuclear industry related to software testing of digital embedded devices is the identification of practical software (SW) testing solutions that provide a strong technical basis and is at the same time effective in establishing credible evidence of SCCF reduction. Towards this effort, we conducted a systematic empirical study on pseudo-exhaustive SW testing methods for embedded digital devices. To our knowledge, this is new and novel work in the application of pseudo-exhaustive testing

methods to nuclear applications. In this work, we aim to investigate the efficacy of pseudo-exhaustive methods by answering the following two questions: (1) Can pseudo-exhaustive testing based on *t*-way combinatorial testing (CT) provide evidence that is congruent with exhaustive testing for an embedded digital device? Under what assumptions and conditions is this claim true? (2) Is *t*-way combinatorial testing effective at discovering logical and execution-based flaws in software-based devices?

As a part of this study, we devised a pseudo-exhaustive testing framework based on *t*-way combinatorial testing that serves as a systematic solution for achieving rigorous software testing while still working within a tractable input and state space. The collected results and derived findings confirm the ability of the automated pseudo-exhaustive testing methodology to economically exercise the interaction input/state space in a systematic, rigorous, and comprehensive manner.

2 SYSTEMATIC PSEUDO-EXHAUSTIVE TESTING

The nuclear regulatory guidelines (NRC BTP 7-19) [1] strongly infer that the testability of software in critical nuclear embedded digital devices is an important criteria for evaluation. It suggests a goal of 100% testability of the software, or if that criteria cannot be met to use diversity and redundancy to mitigate the possibility of SCCF. Exhaustive software testing is a testing approach in which the software needs to be tested with all combinations of input values and state variable values. Considering that the input and state space of typical EDD or I&C software is nearly infinite, literal exhaustive testing is infeasible if not impossible [2]. The natural question arises as to what are the options for addressing this challenge? The number of methods and approaches for testing software are considerable [3], with varying degrees of power to test and exercise the software. One recent method that has gained interest in other industries is so-called pseudo-exhaustive or bounded-exhaustive testing [4]. Pseudo-exhaustive is actually a collection of techniques organized in a systematic process. Another factor we counted is that pseudo-exhaustive testing approaches have an established technical basis and some history of use in safety critical systems [5].

Software testing is considered pseudo-exhaustive when well-formed relations between input space and state-space allow the testable state-space to be reduced, enabling a feasible testable set. The key assumption here is that the state space reduction process must preserve the properties of and among the elements from the original state space. The properties to be preserved are the reachability property of the state space, and the equivalence and interaction relations of the input space. The reduced pseudo-exhaustive tests are assumed to be able to reach and exercise all paths through the code. The reduced pseudo-exhaustive tests are also assumed to be able to exercise all the behaviorally equivalent partitions of the input variables and parameters and all logical interactions between them as would a truly exhaustive test set do. From this description, different approaches to achieve pseudo-exhaustive criteria are possible, and several different approaches have been reported [6] [4]. These assumptions of pseudo-exhaustive criteria are discharged using our proposed pseudo-exhaustive testing methodology that is built upon the following basic techniques: *t*-way combinatorial testing, equivalence partitioning, boundary value analysis (BVA), structural path coverage analysis, and function interaction verification. We discuss these methods briefly; more detailed discourse can be found in the technical report [7].

- Equivalence Partitioning – This method partitions the domain of variable, parameter and configuration space into regions that are control flow and data flow equivalent.
- Boundary Value Analysis (BVA) – This method helps to confine the input model by deriving the most test-effective and error-sensitive set of sample values that lie on the boundaries of these equivalence partitions.
- t-way combinatorial testing – This method provides a means to generate effective tests for testing the interactions between variables, parameters, and configurations across all the equivalence partitions. Previous investigations made by National Institute of Standards and Technology (NIST) on a variety of software applications [8] including medical devices, an HTTP server, a web browser, NASA's distributed database and FAA Traffic Collision Avoidance System [4] indicate that most

of the software faults are triggered by one or two parameters and progressively fewer by the interaction of three, four, five and up to six parameters.

- Structural Path Analysis (Modified Condition/Decision Coverage (MC/DC) metric) – This method helps to ensure that the reduced test set traverses all paths in the code and exercises all possible logical outcomes of the decisions and conditions in the code. It is an important step to evaluate the completeness of the input model and the generated test vectors.
- Function Interaction Verification – This is the method adopted during software integration testing to verify if function call interactions and data flow couplings between functions happen as expected.

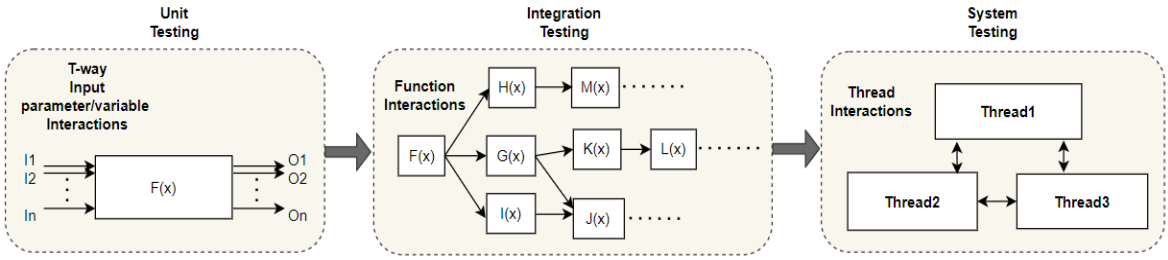

Figure 1: Testing Levels

2.1 Test Strategy

Our testing methodology follows white box testing principles. We assume the source code is available for analysis and review. This assumption is reasonable for safety critical systems certification, where code structural analysis is typically required. Conceptually, the methodology we employ is shown in Figure 1. *t*-way combinatorial testing is applied at the unit function level to test the behavior of all the *t*-way input parameter and variable interactions within the functions. To construct a well-formed input space model for a given function, we employ Equivalence Partitioning and BVA prior to *t*-way testing. During integration testing, we exercise the function call sequences and function interactions at the intra-thread level. Lastly, we perform system testing where interactions across all threads occur. While unit and integration tests focus on the structural coverage of the control flow graphs and call graphs, system testing level focuses on functional coverage of the system level requirements. During system testing, functionality of the system is tested by feeding in data sequences that closely resemble the data sequences coming from a real sensor and off-nominal data (e.g. fuzzing or randomized testing) verifying if the processing and filtering functions within the system are capable of providing a stable and correct output. With this systematic approach we identify interaction faults at all levels of the software.

3 EMBEDDED TARGET: VCU SMART SENSOR

To assess the effectiveness of our proposed systematic software testing workflow we applied it on an embedded target called the VCU Smart Sensor revision 3 [[7], Appendix B]. The VCU smart sensor is a surrogate and representative EDD from a software architectural perspective. The smart sensor performs barometric pressure and temperature state measurement sensing, processes those measurements with a smoothing filter, and then produces filtered state measurements for the controller. The smart sensor is a derivative from VCU ARIES_2 Advanced Autopilot Platform which consists of a mature design and code, and has over 10,000 hours of tested flight time. The VCU smart sensor is built on an STM32F429-Discovery board, which uses the ARM Cortex-M4 168 MHz microcontroller unit (MCU) featuring 2 MB of Flash Memory and 256 KB of RAM. The VCU Smart Sensor software consists of several threads executing periodically in a real time operating system (RTOS). This multithreaded real-time embedded software is representative of the software present in safety-critical I&C devices and sensor devices used in nuclear applications. The VCU smart sensor, though similar in functionality to the smart sensor device assessed by Bishop *et al* [9], has more SW complexity (e.g., more lines of code (LOC) and larger code size) due to the

usage of RTOS, multi-threading, and filtering functions. This additional software complexity of our device is viewed as a benefit for this study as it stresses the testing methodology in capability and scalability.

Table 1: Cyclomatic Complexity of a few functions in smart sensor software

Flash Usage	102 KB		
RAM Usage	81 KB		
	Serial Modem Thread	**Barometer Thread**	**Communication Thread**
Cyclomatic Complexity	102	33	101
Thread Priority	2	4	3
Allocated Stack size	256	256	2048

The three main threads in the smart sensor software are the serial modem thread, communication thread, and barometer thread. The serial modem thread is responsible for receiving and transmitting data from and to the host machine through the USART serial port. The communication thread is responsible for analyzing received requests from the host machine and creating responses to them. The barometer thread (ms5611 thread) is responsible for communicating with the sensor head through the I2C bus, reading in sensor data, and processing and filtering it. The memory footprint on the MCU, the Flash and RAM usage of the entire VCU smart sensor software, is provided in Table 1. Table 1 also gives the priority and working area/stack size allotted for the various threads and cyclomatic complexity at the thread level for the smart sensor software. The cyclomatic complexity of most of the functions within the threads is less than 10 which indicates a modular and testable code as is usually the expectation with safety-critical software. The few functions having a cyclomatic complexity above 10, but still less than 22, indicate moderately complex code with medium testability.

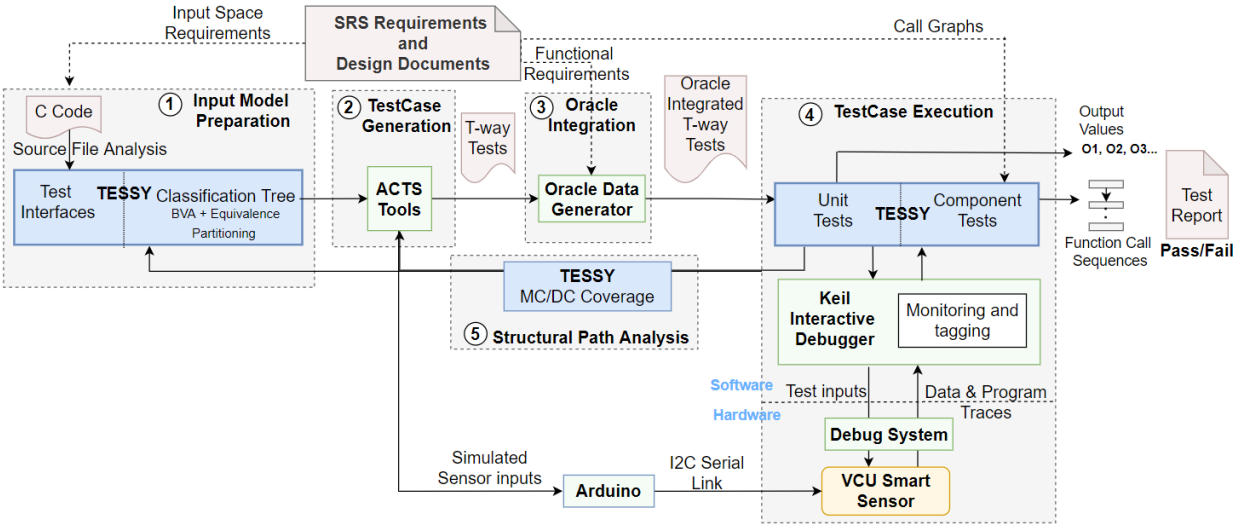

Figure 2: Test Bed Architecture

4 TEST BED ARCHITECTURE AND WORKFLOW

The test bed architecture for realizing the systematic pseudo-exhaustive testing is shown in Fig 2. The test bed diagram depicts the tools involved in the testing process and the major steps in the workflow. The NIST ACTS tool [10] performs automatic test case generation of *t*-way combinatorial test vectors. Razorcat TESSY tool [11] provides the environment to prepare the input model, automate the execution, and code coverage analysis of the tests on embedded software on the actual target. The workflow broadly consists of

five different processes: Input Model Generation, Testcase Generation, Oracle Generation, Test Execution, and Coverage Analysis.

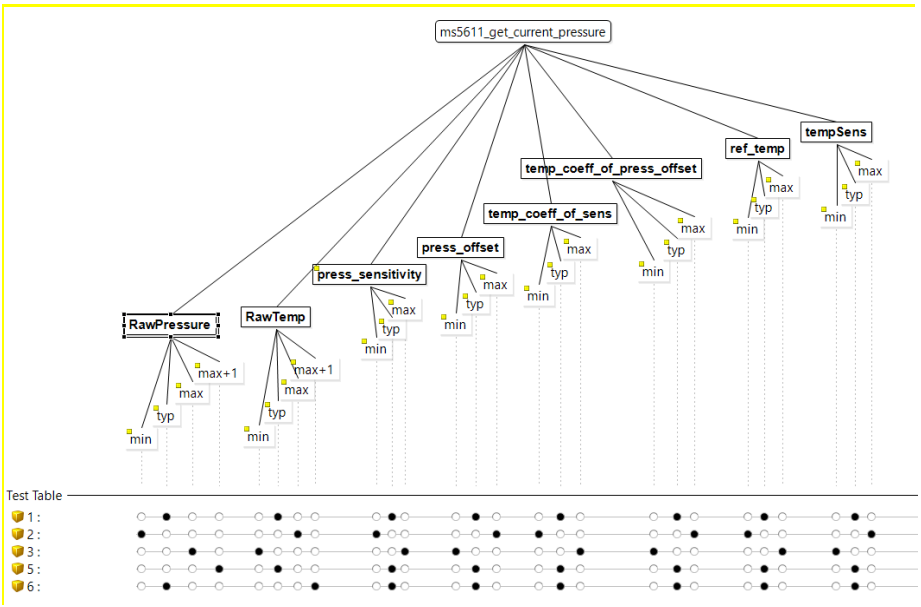

Figure 3 Classification tree for ms5611_get_current_pressure function

4.1 Input Model Generation

Referring to step 1 in Fig 2, the process starts with inputting the C source files into a test module in Tessy. The 'module analysis' conducted in Tessy results in the population of all functions (external and local) and all variables present (external, global, function parameters, return values etc.) in the source files. Input model preparation phase is heavily guided by the principles of Equivalence partitioning and BVA. Tessy's automated analysis of source files help in the identification of the input variables to a function and outputs for monitoring the functional correctness criteria of the function. The identified test interfaces are further refined manually by reviewing them against the Software Design and Requirements documents. Tessy's classification tree editor is used to systematically partition the vast input space of the software under test and collect a set of error-sensitive and effective set of test input values. By referring to the software requirements and design documents and applying a combination of interface-based and functionality-based equivalence partitioning approach, the tester selects appropriate characteristics for each input parameter to partition its vast input domain. The chosen characteristics become the classifications or parent nodes for the tree and sub trees in the classification tree. Each classification is subdivided into behaviorally equivalent partitions (classes). The partitions are expected to be disjoint and to completely cover the input domain of the variable under consideration. Boundary and inlying value partitions for each characteristic are considered separately and become the leaves of the classification tree. The representative boundary and inlying values that are fed into the leaf classes become the set of test inputs used for combinatorial test case generation. Fig 3 shows the classification tree created for input parameters for one of the functions in the smart sensor software. In this case, there are 8 input parameters used in this function. These 8 input parameters are classified based on their valid input ranges specified in the barometer device datasheet. The minimum, maximum and typical values for the sensor calibration parameters and temperature and pressure data are derived from the datasheet. To test for outside normal range values, variables that have invalid values within its datatype range have a classification for "outside normal range value". In this example, valid pressure and temperature data ranges between 0 and 16777215, but the datatype is 'unsigned int32' which ranges from 0 to 4294967296. Hence all values greater than 16777215 are considered as invalid values. To consider this value for testing, an equivalence class 'max+1' is created which is assigned a test input value 16777216.

4.2 Combinatorial Testcase Generation

Referring to step 2 in Fig 2, after the input model preparation, the set of test values for each variable are extracted from the classification tree and fed into the ACTs tool. ACTs tool accepts the input parameters, the parameter types, and the set of parameter values to be used for combinatorial testcase generation. In addition it accepts any constraints and relations between the variables that need to be considered when generating the *t*-way combinatorial testcases. ACTS generates *t*-way test vectors that contain all *t*-way combinations of input values in the input model. The *t*-way (2, 3, 4, 5 and 6-way) testcases generated are exported from ACTs and converted into a format that can be imported into Tessy as unit tests, using a translation script tool we created.

4.3 Oracle Integration

Referring to step 3 in Fig 2, the next step in the workflow is the oracle integration. In order to verify the correctness of the software functions we need to feed in expected output values or the oracle data for the automatically generated *t*-way combinatorial tests. Manually calculating the expected output values for the numerous functions contained in most EDD software is not practically feasible as many thousands of testcases are the norm. The different techniques we can adopt for determining the oracle data are given below:

- Statically feeding in expected output values for each testcase – Practically infeasible for higher interaction levels 3,4,5,6 with >2000 testcases.
- Runtime calculation of test oracle data – Algorithm/equations can be inserted into Tessy's Epilogue/Prologue fields or the stubs of external functions to calculate the expected outputs and Tessy's evaluation macros can be used to compare them with the actual outputs during runtime. These act like a "shell" for automating the comparison of oracle data to actual results.
- Diverse oracle – To support diversity in our oracle module we synthesized model based design of the software function (e.g., a Kalman filter model) in MATLAB Simulink. The auto generated code could be used as the oracle algorithm or the actual output values from the model could be used as the expected values to compare to the output values from the software under test.

4.4 Test Execution

Referring to step 4 in Fig 2, tests are executed on the target using the On Chip Debugger port (Serial Wire Output (SWO) port for ARM processors) of the microprocessor which allows direct unintrusive insertion of the test vectors into the memory and register stack of the executing software. We used Keil µvision and ST-Link debugger to run the tests directly on the STM32F4 microcontroller. By collecting the real time values and time stamps of all program data and comparing them with the expected output values, Tessy automatically evaluates testcases as Pass or fail. The use of embedded debuggers facilitates automated test execution, but additionally it provides high levels of controllability and observability with respect to the executing embedded code – which is highly desirable. This helps testers have a "window" into the execution flow thereby aiding in the root cause of failed testcases, and discerning complex failures like timing and synchronization issues. The article by Weiss *et al* [12] gives more detail on the use of embedded debuggers for SW testing. At the unit testing level, for functions within the threads, test executions starts with the baseline test of 2-way combinatorial testing which then proceeds to 3, 4, 5 and 6-way tests.

4.5 Structural Path Analysis and Feedback

Referring to step 5 in Fig 2, the structural path coverage information generated from Tessy after each test execution is fed back to Input Model Creation and Testcase Generation stages to improve the input model or the *t*-way level of the tests until we achieve a 100% MC/DC coverage on the code or as close as

possible to that metric level. It has been previously proven [13] that given an input model with necessary variable values, 100% branch coverage for t-way conditionals is obtained if the sum of minimum combinatorial coverage (M_t) and minimum proportion of t-way combinations needed to trigger a branch within the code (B_t) is greater than one, i.e., $M_t + B_t > 1$. This theorem implies that by using covering arrays that have minimum combinatorial coverage of 100% ($M_t = 1$) and a complete input model, 100% branch coverage is guaranteed. Therefore by using MC/DC coverage which subsumes branch coverage, we verify the adequacy of the equivalence classes and boundary values in our input model.

We observed that low coverage is in most of the cases due to a deficient input model; either an unconsidered input parameter or an unconsidered sample value. This could be due to human errors that happen during input model creation which is a fully manual effort. As shown in Fig 4, if low coverage was found to be the result of not considering an input parameter in the tests, the missing parameter is added to the test interfaces and combinatorial testcases are re-generated with the updated input model. If low coverage is due to not considering a value for an input parameter in the tests, this value is added as a new class to the classification tree and combinatorial testcases are re-generated with the updated input model. In some cases, the input model would be comprehensive and the low code coverage obtained with t-way tests would be due to the presence of '$t + n$'-way conditionals present in the code. This is fixed by increasing the interaction level of the test vectors to '$t + n$'. In very rare cases, low coverage could be due to the test vectors not executing error handling portions of the code. Fault injection tests needs to be performed to cover them. We observed that with an insufficient input model, we need a higher t-way interaction level test vectors to cover the entire code and meet the goal of 100% MC/DC coverage. For example, for the circular buffer read function in the serial_modem thread, with an input model that does not include a value that is greater than the size of buffer for the length parameter, the 2-way tests containing 20 test vectors only achieved an MC/DC coverage of 87.5%. However, with the same input model, a 100% MC/DC coverage was achieved when switched to the 3-way interaction level containing 64 test vectors. This is not economical as with each increase in the interaction level, there is an exponential increase in the number of test vectors. Whereas, when the input model was improved by adding a boundary value for the length parameter that is greater than the size of the buffer, we achieved 100% MC/DC coverage with just the 2-way interaction level containing 28 tests.

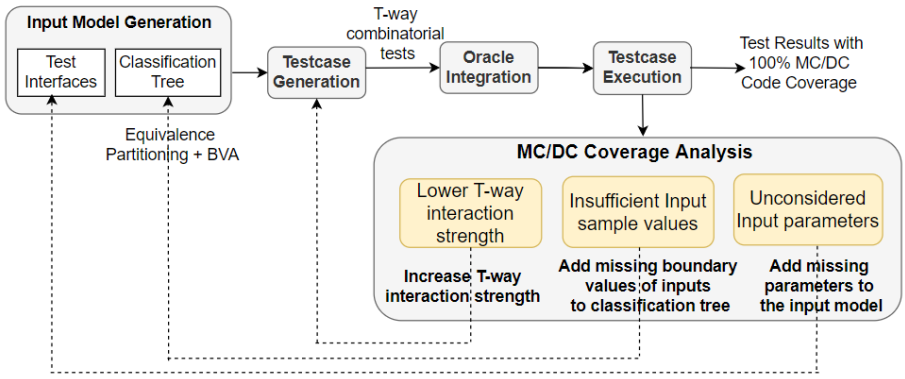

Figure 4: Coverage Analysis Workflow

4.6 Component/Thread Level Integration Tests

Integration testing is essential to verify that the interactions between the functions happen as intended. Functions interact by calling one another and passing data between them as arguments and return values. These interactions are depicted in functions call graphs. Function calling sequences can be complex structures, involving decisions, modes of operation, predicates, etc. The function call graphs need to be analyzed and these complex interactions need to be understood by the tester to formulate effective integration testcases. Covering the nodes (ensuring each function is called at least once) and edges (every

call of function is invoked at least once) in a function call graph is essential for rigorous integration testing, and Pseudo-Exhaustive Testing. In order to traverse the different paths in the function call graphs, we need to identify the different sequences of input data values to be fed into the progressing time steps of thread execution. Tessy's component test feature helps to conduct integration testing on a software component or a software thread that consists of several internal functions which makes calls to underlying external functions. Scenarios can be created in Tessy to invoke the component/thread level function and to verify the order of the external function calls made by the thread at every periodic time steps (eg: 0ms, 10ms, 20ms… for 10ms periodic thread). By providing data values to the inputs at every time step, the identified outputs at the thread level can be verified against expected values after thread execution for every time step.

5 RESULTS, ANALYSIS AND FINDINGS

Several critical native bugs in the smart sensor software were uncovered by applying the proposed systematic test methodology of incrementally conducting the different levels of interaction testing on the smart sensor software. All the bugs were caught during unit testing by applying the 2-way combinatorial test vectors. Although the same test failures re-appeared during the 3, 4, 5 and 6-way combinatorial tests, no additional bugs were caught in the 3 to 6-way combinatorial tests. It was also seen that with a carefully constructed input model, we were able to achieve 100% MC/DC coverage with the 2-way combinatorial tests for all functions. Based on the time taken to execute the *t*-way tests on the smart sensor software functions of varying complexity levels, it is seen that on an average it took 1sec to execute a single testcase on the physical hardware device. This faster test execution rate, achieved due to the test automation in Tessy, helped to reduce the total testing time even when dealing with larger test suites (>5000 test vectors) for the 5 and 6-way tests, thereby increasing the test efficiency. Table 2 indicates the average number of variables, values/variable and the number of *t*-way testcases generated for few functions in smart sensor software. On average, most of the functions in the smart sensor software dealt with 5 input parameters and 5 sample values/parameters. The number of testcases can be seen to be exponentially increasing as the *t*-way interaction level increases.

Table 2: Input model and Test generation details for a few functions

Tested function	# Variables	# Values/ Variable	2-Way Tests	3-Way Tests	4-Way Tests	5-Way Tests	6-Way Tests
Circular_buffer_read	4	4-5	28	125	500	N/A	N/A
get_current_pressure	9	4-5	20	76	285	870	2411
kalman_filter	5	6-7	48	316	1608	7776	N/A

Table 3 shows the test failures that were observed during 2-way combinatorial tests and the root causes for them. It can be noted that a majority of the testcase failures is due to the software not handling invalid inputs, such as invalid buffer size, number of bytes, sensor data, etc. Not handling invalid inputs could end up in software responding with false or even meaningless data. 'Missing buffer overflow checks' in software is another more critical bug that is caught with this test methodology. Not checking if the length of bytes to be copied exceeds the destination buffer size before calling standard C library 'Memcpy' function results in the function corrupting other valid memory locations and causing the software to get hung in an undefined state. The test methodology is thereby found capable of detecting potential buffer overflow security vulnerabilities in the code. Few others bugs due to the software not following good programming practices like divide by zero and overflow checks during mathematical computations have also being caught in the 2-way combinatorial tests. Errors in boundary guard conditions (e.g., incorrect buffer full check) also get caught with the 2-way tests. The different classes of native bugs being caught and their severity indicate the effectiveness of our pseudo-exhaustive testing methodology.

Table 3: List of native bugs caught during testing

TestCase Failure	Root cause
Unable to fill the buffer completely. Can only fill buffersize-1 elements.	Incorrect buffer full check
TestExecution Timeout - Buffer overflow and corruption of neighboring memory addresses cause the 'Memcpy' function to hang when called with a length greater than the destination buffer size.	Missing destination buffer overflow check
Indicates successful data read operation even with invalid configurations of buffer, 'size of buffer', 'head' and 'tail' pointers.	Invalid buffer configurations are not handled.
Returns varying negative values of buffer read length when the requested 'number of bytes' is negative.	Invalid negative values of the number of bytes to be read is not handled.
Negative values of buffer size are accepted during buffer initialization and the buffer is filled with negative size value.	Invalid buffer size is not considered during buffer initialization
Actual output value indicates 'Infinity'	Missing Divide by Zero check
Actual output value indicates 'NaN' (Not a Number)	Missing Overflow check in float computations
Function processes input values outside valid range	Missing invalid input value handling

5.2 Seeded Fault Testing

To further evaluate the effectiveness of our methodology, we applied our approach on a set of reference faulty versions of the code i.e., seeded faults in the original code. By using a wide variety of mutants that included operator, operand, datatype and constant mutations in the code, this activity helped us ascertain the effectiveness of the approach to different fault classes. While several mutants were detected by combinatorial tests at the function unit level, few mutants were detected only when the entire thread was tested with the main thread function calling the sub functions and passing data between functions. The general categories of seeded bugs that were caught in intra-thread level integration testing but not in unit testing are:

- Bugs related to values and datatypes of function arguments being passed.
- Bugs that affect the call sequences of 'external functions'.
- Bugs related to initialization of global or file static variables which need to consider the entire C file and not just a function.

This seeded fault testing also indicated that along with valid values derived from equivalence partitions and BVA, an "Undefined" value also needs to be considered for all the input variables. This forces the software to use the initialization values of the variables given in the code or the default initialization values depending on their storage classes. Making this consideration of using the special undefined/ignore value (specified in Tessy as '*none*') for variables during the input model generation phase, helps to find bugs related to faulty or missing initialization values. Table 4 shows a snapshot of the seeded faults/mutations made in the software, the stage of testing that caught the fault and the test failure that was observed due to the embedded fault in the code. All valid mutants applied to the code were 'strongly detected' by the combination of *t*-way combinatorial unit tests and intra-thread level integration tests. Few mutants that were not detected revealed dead/unreachable code portions within the software. The seeded fault testing thereby validates the quality of the *t*-way combinatorial unit tests and function interaction integration tests and their ability to detect a wide variety of software errors.

Table 4: Seeded Fault Testing Results

Seeded Fault	Test Failures	Caught by
Relational Operator Replacement: Operator: Replaced '<' with '>' in sensor value processing code.	Processed pressure data deviated from expected values.	Get_current_pressure function 2-way tests
Shift Operator Replacement Operator '>>' replaced with '<<' in sensor value processing code.	Processed pressure data deviated from expected values.	
Constant Replacement: Hardcoded value for pressure coefficient used in pressure calculation is faulted.	Processed pressure data deviated from expected values.	
Scalar Variable Replacement: Replaced float variables varM with varP in kalman filter function	Filtered pressure output deviated from expected values	Kalman_filter function 2-way tests
Arithmetic Operator Replacement: Replaced + with - in kalman filter.	Filtered pressure output deviated from expected values	
Statement deletion: Removed initialization values for the kalman filter coefficients.	Filtered pressure output deviated from expected values	Thread level integration test.
Datatype Modification: Replaced uint32 datatype of a function argument with uint16.	Data truncation during function call results in erroneous pressure output value.	
Logical Inversion: Check for Calibration was negated.	Function call trace during thread execution deviated from the expected call trace.	
Operand in Relational Operation Off-by-one: Thread Run Reset condition changed	Function call trace after the allotted cycles for the thread, deviates from the expected call trace.	
Constant Replacement: Faulty Initialization values for the kalman filter coefficients	Filtered pressure output data deviated from expected values immediately after thread initialization.	

6 RELATED WORK

The effectiveness of combinatorial testing (CT) on a number of critical application domains ranging from detecting software design faults in automotive [14], avionics [5] to detecting vulnerabilities in web applications [15] have been demonstrated. A comparative study [16] conducted by Wu *et al*, using nine real-world programs indicate that CT is more efficient than random testing in detecting hard to find faults in software. Li *et al* studied [17] the effectiveness of CT in improving MC/DC coverage of the code and Kuhn *et al* [18] developed the relationship between structural coverage and input space coverage. A previous study [19], aimed at assessing the capability of combinatorial interaction tests to detect input model mutations using model-based mutation testing indicated that 3-way test suites were capable of detecting all the input mutants which 2-way tests could not. An oracle free testing approach using two-layer covering arrays introduced by Kuhn *et al* [20] is capable of verifying the correctness of equivalence classes in the input model and to detect certain classes of software errors. The descriptions on the application of CT on safety-related prioritization module that is part of AREVA NP I&C platform to preclude the presence of software common cause failures in it can be found in US EPR report [21]. The technical analysis of smart sensors to justify their use in safety-critical applications in the Nuclear industry is conducted by Bishop *et al* [9]. A closely related previous work is where Wood *et al* [23] employed a systematic model-based testing

approach on the VCU smart sensor. In this work, a hierarchical mutation-based testing approach was used to detect seeded design faults. Our approach differs from this work, in several ways, namely we use *t*-way testing instead of mutation testing as a basis for stimulating interaction faults. Another closely related work [22] is where the authors developed an automated black-box software reliability test system to test software in EDDs. The authors evaluated the effectiveness of Software Reliability Testing (SRT) by using it to test the VCU smart sensor software with seeded faults.

7 CONCLUSIONS AND RECOMMENDATIONS

The proposed systematic test methodology provides a promising pathway for rigorous systematic testing of safety critical embedded software in nuclear power applications, offering potential options for 'device diversity' and 'defense in depth' solutions for addressing software common cause failures. Our experiments and preliminary results indicate that Pseudo-Exhaustive testing at all levels of interaction (variable, function and thread interactions) on an embedded device can detect a diverse range of software flaws. We discovered a number of native defects (in addition to the seeded flaws) in the code that were latent for some time, and in one case the defect was there for years. Our study also provides affirmative evidence to the claim made by previous studies that most faults are found at lower levels of variable interaction (level 2 in our case). The combination of *t*-way combinatorial tests guided by path analysis (MC/DC) is a compelling approach for reasoning about "completeness or stopping point" of software testing. Careful selection of boundary values during the input model generation phase, combinatorially verifying the unit functions up to 6-way variable interactions and verifying the function and thread interactions help to exhaustively cover the functionality and structure of the software being tested. One of the objectives of this project was to conduct the research on state-of-the-art commercially available or open-source tools. For the most part, the tools used in this study implemented and executed the pseudo-exhaustive testing strategy very well. The automated testing efficiency was in 150K tests/day. That said, the use of automated tools is not without challenges, especially in real time multi-threaded software where timing and ordering of tasks is critical as we noted above with the thread level testing. While the tools were very capable, the main challenges we faced were building the testbed architecture to conduct the study [7].

8 ACKNOWLEDGMENTS

This research was supported by the US Department of Energy, Idaho National Laboratory through the Light Water Reactor Sustainability program under contract FP 217984.

9 REFERENCES

1. U. N. R. Commission, "Guidance for Evaluation of Diversity and Defense-in-Depth in Digital Computer-Based Instrumentation and Control Systems," Branch Technical Position, pp. 7–19, 2007.
2. R. W. Butler and G. B. Finelli, "The infeasibility of quantifying the reliability of life-critical real-time software," IEEE Transactions on Software Engineering, vol. 19, no. 1, pp. 3–12, 1993.
3. Jovanovic, Irena, "Software Testing Methods and Techniques," The IPSI BgD Transactions on Internet Research 30 (2006), 2006.
4. D. Kuhn and V. Okum, "Pseudo-Exhaustive Testing for Software," in 2006 30th Annual IEEE/NASA Software Engineering Workshop, Columbia, MD, USA, Apr. 2006, pp. 153–158.
5. J. D. Hagar, T. L. Wissink, D. R. Kuhn, and R. N. Kacker, "Introducing Combinatorial Testing in a Large Organization," Computer, vol. 48, no. 4, pp. 64–72, Apr. 2015.
6. K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson, "Software Assurance by Bounded Exhaustive Testing," p. 10.
7. Elks, Carl, et al. "Preliminary Results of a Bounded Exhaustive Testing Study for Software in Embedded Digital Devices in Nuclear Power Applications." Idaho National Laboratory US Department of Energy Office of Nuclear Energy report INL/EXT-19-55606 (2019).

8. R. Kuhn, Y. Lei, and R. Kacker, “Practical Combinatorial Testing: Beyond Pairwise,” IT Professional, vol. 10, no. 3, pp. 19–23, May 2008.
9. P. Bishop, R. Bloomfield, S. Guerra, and K. Tourlas, “Justification of Smart Sensors for Nuclear Applications,” in Computer Safety, Reliability, and Security, vol. 3688, R. Winther, B. A. Gran, and G. Dahll, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 194–207.
10. L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn, “ACTS: A Combinatorial Test Generation Tool,” in 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, Luxembourg, Luxembourg, Mar. 2013, pp. 370–375.
11. “TESSY - Test System - Razorcat Development GmbH.” https://www.razorcat.com/en/product-tessy.html (accessed Feb. 29, 2020).
12. A. Weiss et al., “Understanding and Fixing Complex Faults in Embedded Cyberphysical Systems,” Computer, vol. 54, no. 1, pp. 49–60, Jan. 2021.
13. D. R. Kuhn, R. N. Kacker, and Y. Lei, “Measuring and specifying combinatorial coverage of test input configurations,” Innovations Syst Softw Eng, vol. 12, no. 4, pp. 249–261, Dec. 2016.
14. Dhadyalla, Gunwant, Neelu Kumari, and Timothy Snell, “Combinatorial Testing for an Automotive Hybrid Electric Vehicle Control System: A Case Study,” in IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops. IEEE, 2014., 2014, p. 7.
15. D. E. Simos, J. Zivanovic, and M. Leithner, “Automated Combinatorial Testing for Detecting SQL Vulnerabilities in Web Applications,” in 2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST), May 2019, pp. 55–61.
16. H. Wu, C. Nie, J. Petke, Y. Jia, and M. Harman, “An Empirical Comparison of Combinatorial Testing, Random Testing and Adaptive Random Testing,” IEEE Transactions on Software Engineering, vol. 46, no. 3, pp. 302–320, Mar. 2020.
17. D. Li, L. Hu, R. Gao, W. E. Wong, D. R. Kuhn, and R. N. Kacker, “Improving MC/DC and Fault Detection Strength Using Combinatorial Testing,” in 2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), Jul. 2017, pp. 297–303.
18. R. Kuhn, R. N. Kacker, Y. Lei, and D. Simos, “Input Space Coverage Matters,” Computer, vol. 53, no. 1, pp. 37–44, Jan. 2020.
19. M. Bures and B. S. Ahmed, “On the Effectiveness of Combinatorial Interaction Testing: A Case Study,” in 2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), Jul. 2017, pp. 69–76.
20. D. R. Kuhn, R. N. Kacker, Y. Lei, and J. Torres-Jimenez, “Equivalence class verification and oracle-free testing using two-layer covering arrays,” in 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Apr. 2015, pp. 1–4.
21. “Methodology for 100% Combinatorial Testing of the U.S. EPR Priority Module Technical Report,” AREVA NP Technical Report ANP-10310P, Revision 1, Mar. 2011.
22. R. Wood, H. M. Hashemian, B. Shumaker, C. Smidts, and C. Elks, “Development of A Model Based Assessment Process for Qualification of Embedded Digital Devices in NPP Applications: Research Approach and Current Status,” p. 6, 2017.
23. B. D. Shumaker, R.T. Wood, C. Elks, and C. Smidts, “Software Reliability Testing of Digital Devices for Nuclear power applications,” presented at the NPIC-HMIT 2021, Mar. 2021.