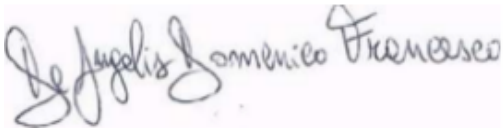
PhD in Information Technology and Electrical Engineering
Università degli Studi di Napoli Federico II

PhD Student: Domenico Francesco De Angelis

Cycle: XXXIX

Training and Research Activities Report

Academic year: 2024-25 - PhD Year: Second



Tutor: prof. Anna Rita Fasolino

Co-Tutor: 

Date: October 21, 2025

Training and Research Activities Report

PhD in Information Technology and Electrical Engineering

Cycle: **Author:**

1. Information:

- **PhD student: Domenico Francesco De Angelis PhD Cycle: XXXIX**
- **DR number: 1076**
- **Date of birth: 19/08/1993**
- **Master Science degree: Computer Engineering University: Federico II**
- **Scholarship type: scholarship**
- **Tutor: Anna Rita Fasolino**
- **Co-tutor: -**
- **Period abroad: -**

2. Study and training activities:

Activity	Type[1]	Hours	Credits	Dates	Organizer	Certificate[2]
IOT Data Analysis	Course	16	4	28-31/01 - 4-7-11-14-25/02/2025	prof. Raffaele Della Corte, DIETI	Y
How to boost your PhD	Course	24	5	08-15-22-29/01/2025 05-12/02/2025	Prof. Antigone Marino	Y
AI Code Generation: Foundations, Evaluation, and Security	Course	11	3	7-10-14-15-17-31/10/2025	dr. Pietro Liguori	Y
DNS 2025	Seminar		3	23-24-25-26/06/2025		Y
ICTS 2025	Seminar		3	2-3-4/4/2025	Prof. Anna Rita Fasolino	Y
Guardians or Threats? AI at Frontlines of Cybersecurity	Seminar	4	0,8	17/10/2025	Prof. Antonia Tullino	Y

1) Courses, Seminar, Doctoral School, Research, Tutorship
2) Choose: Y or N

2.1. Study and training activities - credits earned

	Courses	Seminars	Research	Tutorship	Total
Bimonth 1	-	-	6	-	6
Bimonth 2	9	-	6	-	15
Bimonth 3	-	3	6	-	9
Bimonth 4	-	3	6	-	9
Bimonth 5	-	-	6	-	6
Bimonth 6	3	0.8	6	-	9.8
Total	12	6.8	30	-	48.8
Expected	30 - 70	10 - 30	80 - 140	0 – 4.8	

3. Research activity:

During the second year of my PhD program, I continued my research in the field of software engineering, focusing on the analysis and testing of legacy embedded software in safety-critical domains. My activities are primarily centered around the automatic extraction of functional equivalence classes from undocumented C functions, with the goal of supporting certification processes aligned with standards such as **ISO 26262** [2] and **IEC 61508** [3].

The motivation behind this research stems from the challenges faced when applying systematic testing techniques, such as **Equivalence Class Partitioning (ECP)**, to legacy codebases that lack formal specifications and documentation.[1] These challenges are particularly relevant in industrial environments where software reuse is common, but traceability and functional clarity are often missing.[1]

To address these issues, I developed a methodology based on symbolic execution, which enables the exploration of all feasible execution paths of a function by treating inputs as symbolic variables. This approach allows the automatic generation of path conditions, which are logical constraints describing the input values that lead to specific output behaviors. These path conditions are then grouped into *equivalence classes*, representing sets of inputs that produce functionally identical outputs.

The research activities carried out during this year include:

4. **Design and implementation of a symbolic execution pipeline**: The pipeline includes control-flow graph construction, symbolic path exploration, constraint simplification, and equivalence class generation.
5. **Optimization of symbolic constraints**: Through pattern recognition [4] and clustering, the extracted conditions are transformed into compact and readable expressions suitable for documentation and test generation.

To illustrate our methodology for automatically deriving equivalence classes, I present these **running examples** demonstrate the effectiveness of symbolic execution in recovering functional boundaries and generating equivalence classes:

- **Case 1 – Branching Function**: A simple function with conditional branches returning different constants based on input ranges. Symbolic execution identifies distinct path conditions, which are grouped into equivalence classes based on output values.

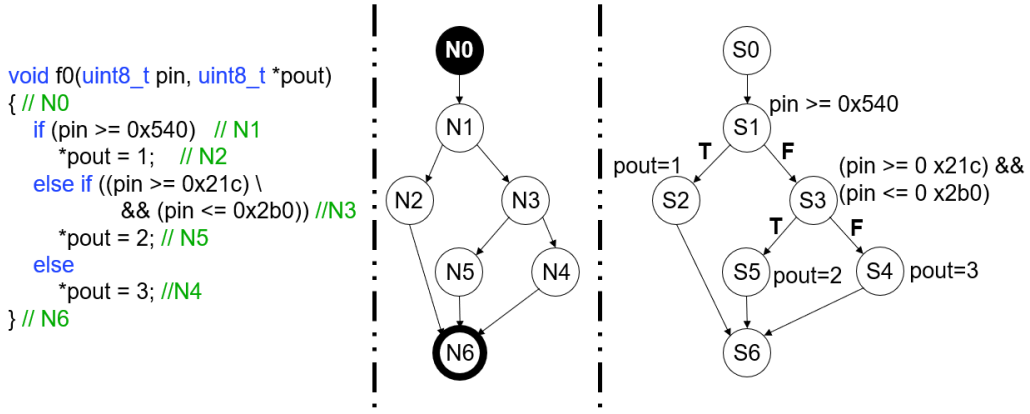


Fig. 1 Function, Control flow graph, symbolic execution trace than generate path conditions

In this case, we want show how was perform the optimization on path conditions to have a more human-readable conditions. The conditions wrote by symbolic execution are defined as follows:

Symb Exec State	path condition
S2	$!(\text{pin} == 0x540) \wedge 0x540 \leq \text{pin}$
S5	$!(\text{pin} == 0x21c) \wedge 0x21c \leq \text{pin} \wedge$ $(!(0x540 \leq \text{pin}) \vee \text{pin} == 0x540) \wedge$ $(!(0x220 \leq \text{pin}) \vee \text{pin} == 0x220)$
S6	$(!(0x21c \leq \text{pin}) \vee \text{pin} == 0x21c) \wedge$ $(!(0x540 \leq \text{pin}) \vee \text{pin} == 0x540) \vee !(\text{pin} == 0x220) \wedge 0x220 \leq \text{pin} \wedge$ $(!(0x540 \leq \text{pin}) \vee \text{pin} == 0x540)$

Table 1: Equivalence classes (EC) with return value, output `pout`, and input constraint on `pin`.

To improve readability and eliminate redundancy, we apply a *pattern recognition* step that identifies recurring constraint structures. We optimize patterns of the form $!(CONST <= var) \,||\, (var == CONST)$ into the simpler equivalent $CONST > var$ and this other pattern: $!(CONST == var) \,||\, (var <= CONST)$ into the simpler equivalent $CONST < var$.

By merging these patterns into concise interval expressions, we produce compact equivalence classes that preserve the original semantics while significantly reducing complexity. The result is summarized in Table 2

EC id	return value	pout	ECP
1	none	1	$0x540 < \text{pin}$
2	none	3	$\text{pin} \leq 0x21B \vee (\text{pin} > 0x2B0 \wedge \text{pin} \leq 0x540)$
3	none	2	$0x21B < \text{pin} \wedge \text{pin} \leq 0x2B0$

Table 2: Equivalence classes (EC) with return value, output `pout`, and input constraint on `pin`.

- **Case 2 – Bitwise Loop Function**: A function that returns the index of the first set bit in a byte. Symbolic execution explores all feasible paths, each corresponding to a different bit position, as

```
uint8_t f16(uint8_t p1)
{ //N0
  for (uint8_t i = 0u; i < 8u; i++) { // N1
    if ((p1 & (1u << i)) != 0u) // N2
      return i; // N3
  } // N4
  return 0xFFu; // N5
}
```

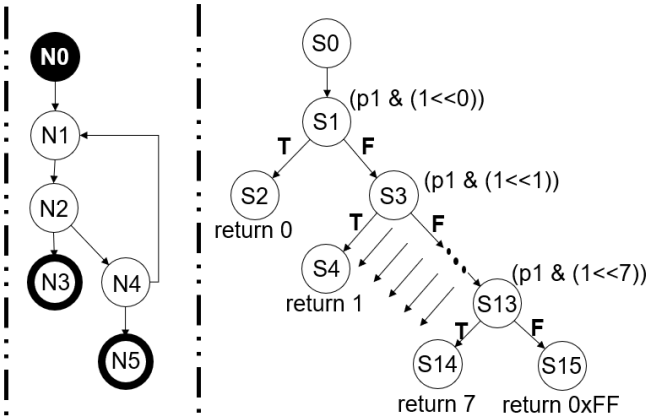


Fig.2

Fig 2. Function with loop, CFG, and symbolic execution trace

To avoid redundancy, path conditions are clustered based on output behavior: inputs with at least one bit set form one equivalence class, while inputs with no bits set form another as Fig 3.

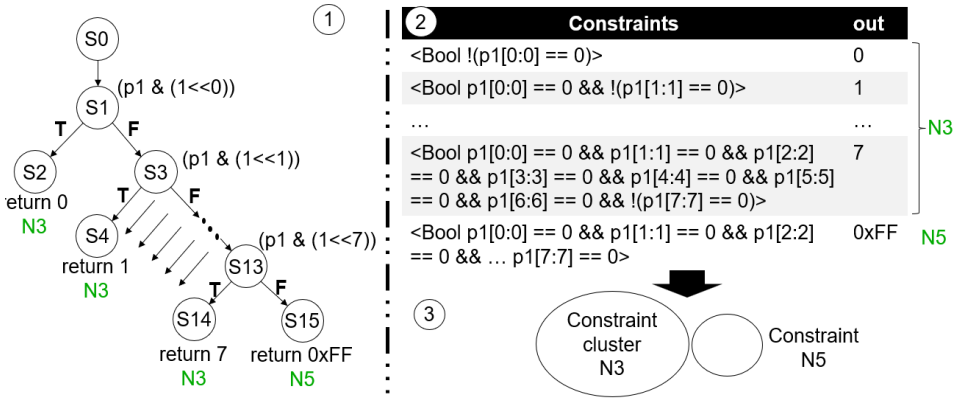


Fig.3

This grouping is based on the common structure of the constraints and on the fact that they all lead to the same kind of outcome. By recognizing this recurring pattern, the system replaces many nearly identical constraints with one optimized condition capturing the entire group.

A separate cluster is formed for the path where the loop completes without finding any set bit. This path is unique and represents the case where the function returns *0xFF*.

After clustering, the function behavior reduces to two clear equivalence classes: one where at least one bit is set, and one where no bits are set. Table 3 summarizes the resulting equivalence classes.

EC id	return value	ECP
1	0xFF	p1 == 0
2	$0 <= ret < 8$	p1! = 0

Table 3: Equivalence classes (EC) with return value, p1.

- **Case 3 – Arithmetic and Bitwise Function**: A more complex function involving multiple inputs and a boolean flag that determines whether a sum or difference is computed. The equivalence classes are defined by combinations of input values and flag states that lead to distinct outputs.

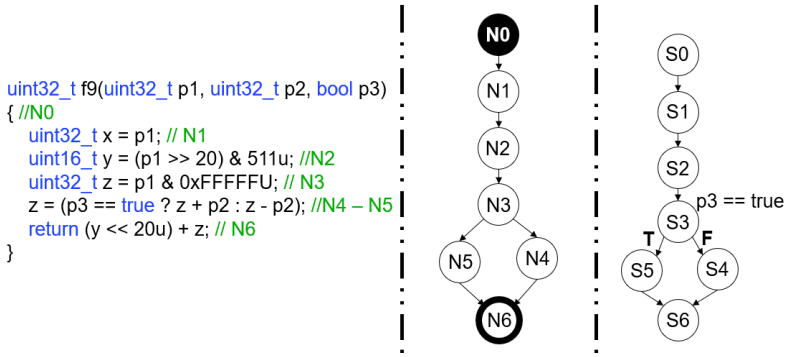


Fig. 4 Function some math operation computed, CFG, and symbolic execution trace

Fig. 4 shows the control-flow graph and symbolic execution trace, where each path condition corresponds to a unique input configuration. These are grouped into equivalence classes based on the resulting arithmetic operation and output value, demonstrating how symbolic analysis captures functional distinctions driven by both control and data dependencies.

EC id	return value	ECP
1	$p2 + p1[19:0] + (p1[28:20]..0x0)$	$p3 == 1 \wedge$ $p2 + p1[19:0] + (p1[28:20]..0x0) <= 0xffffffff$
2	$p1[19:0] - p2 + (p1[28:20]..0x0)$	$p3 == 0 \wedge$ $p1[19:0] - p2 + (p1[28:20]..0x0) <= 0xffffffff$

Table 4: Equivalence classes (EC) with return value of f9.

The expression defined in Table 4 is expressed in *bitvector logic*. This means each equivalence class is represented using bit-level operations and ranges, which precisely capture how the function manipulates subsets of bits from the input parameters. For example, *p1[19:0]* refers to the lower 20 bits of *p1*, while *p1[28:20]* extracts the upper segment. These slices are combined with arithmetic operations (addition or subtraction) depending on the boolean flag *p3*. The constraints ensure that the computed result stays within the 32-bit range *(<= 0xffffffff)*, preserving correctness for embedded systems.

To conclude, I have started validating the proposed methodology through controlled experiments on representative functions, labelling a dataset of equivalence classes and check the outputs if it follow the concept of correctness and readability. In the short term, we plan to extend this validation with an industry-oriented survey, aiming to assess the applicability and effectiveness of our approach in real-world automotive software development contexts.

Reference

[1] Malladi, S., Ramakrishna, G., Rao, K., Babu, E.: Analysis of legacy system in software application development: A comparative survey. International Journal of Electrical and Computer Engineering (IJECE) 6, 292–297 (02 2016). https://doi.org/10.11591/ijece.v6i1.8367

[2] SO/TC 22/SC 32: Road vehicles – functional safety – Part 1: Vocabulary. Standard ISO 26262-1:2018 to ISO 26262-12:2018, International Organization for Standardization, Geneva, Switzerland (2018), https://www.iso.org/standard/68383.html

[3] Commission, I.E.: Functional safety of electrical/electronic/programmable electronic safety-related systems (2010)

[4] Amon, T.T., Loffredo, T.J.: Creating human readable path constraints from symbolic execution. Tech. rep., Sandia National Lab. (SNL-NM), Albuquerque, NM (United States) (2020)

6. Research products:

-

5. Conferences and seminars attended

-

6. Periods abroad and/or in international research institutions

- To be done in 3rd year

7. Tutorship

-

8. Plan for year three

- Refine and publish current work on equivalence class extraction, especially for embedded firmware.
- Further investigate software testing and code quality, focusing on code smells and structural issues.
- Explore methods to support code understanding and technical debt prediction in large-scale industrial codebases.