Domenico Francesco De Angelis

Static Analysis Techniques for Software development project in Safety-Critical context

Tutor: Anna Rita Fasolino

Cycle: XXXIX

Year: 2024/2025

Candidate's information

- MSc degree: Computer Engineering
- Research group/laboratory: ReverSE Group - **Re**search Group in **S**oftware **E**ngineering
- PhD start date: 1st November 2021
- Scholarship type: Not funded
- Sr. Firmware Engineer at Micron Technology (based in Arzano)
  - ~6 years

Summary of study activities

- **Courses:**
  - IOT Data Analysis - prof. Raffaele Della Corte, DIETI
  - How to boost your PhD - Prof. Antigone Marino
  - AI Code Generation: Foundations, Evaluation, and Security – dr. Pietro Liguori
- **Seminars (and Conferences)**
  - DNS 2025
  - ICTS 2025
  - Guardians or Threats? AI at Frontlines of Cybersecurity

Research overview (1/2)

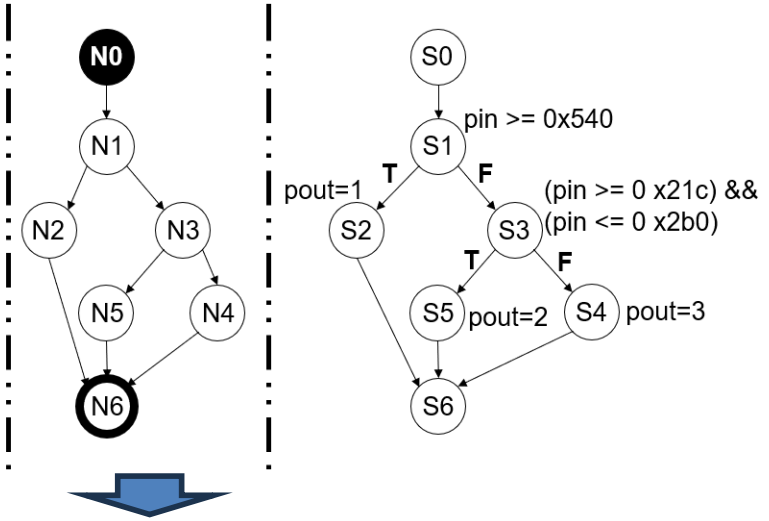
- Context:
  - The automotive industry is adopting several standards to ensure that software systems are fault-tolerant and meet stringent safety and quality requirements;
  - In the automotive industry, legacy code is consistently reused to expedite the time-to-market;
  - Legacy code could not always follow a defined code-style and use mechanism to hide code behavior (as macro definition to hide code)
- Problem:
  - Lack of design documentation in industrial code practices.
- Objective:
  - Implement a tool to recovery design specifics of function unit in legacy code basing on Equivalence classes partitioning ECP.

Research overview (2/2)

- Methodology:
  - *Definition of a technique for recovering the equivalence classes by function in legacy code*
  - *Implementation of the technique in a tool*
  - *Validation using artificial functions and survey*
- *The Technique includes:*
  - ***Symbolic analysis**: using ANGR framework performing a symbolic execution.*
  - ***Human Readable optimization:*** starting from path condition transform it in more human readable way
  - ***Generate automatic stub for symbolic execution**: After the symbolic execution of a function the constraints are used to create a stub used to generate stub (avoiding emulation overhead).*

Running Examples – case 1

```
void f0(uint8_t pin, uint8_t *pout)
{ // N0
   if (pin >= 0x540)   // N1
      *pout = 1;    // N2
   else if ((pin >= 0x21c) \
            && (pin <= 0x2b0)) //N3
      *pout = 2; // N5
   else
      *pout = 3; //N4
} // N6
```



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.

Optimizations Rules:

1 - $!(CONST <= var) \ OR \ (var == CONST)$

→ $CONST > var$

2 - $!(CONST == var) \ || \ (var <= CONST)$

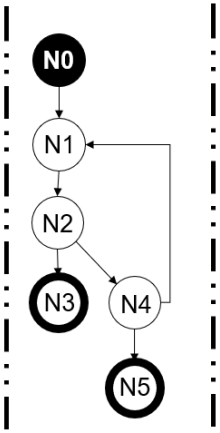
→ $CONST < var$

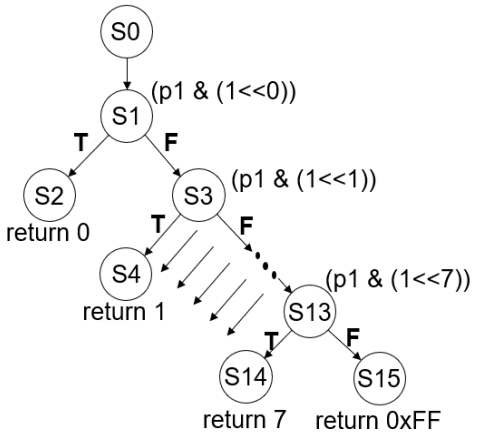
EC id	return value	pout	ECP
1	none	1	$0x540 < \text{pin}$
2	none	3	$\text{pin} \leq 0x21B \ \vee \ (\text{pin} > 0x2B0 \ \wedge \ \text{pin} \leq 0x540)$
3	none	2	$0x21B < \text{pin} \ \wedge \ \text{pin} \leq 0x2B0$

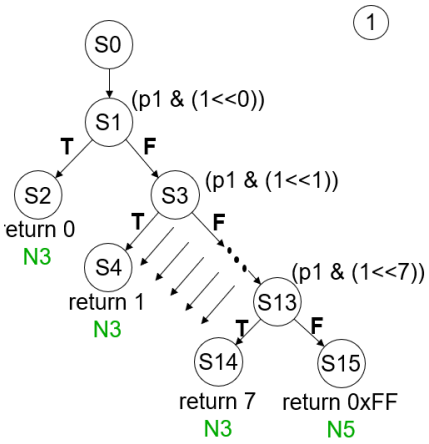
Table 2: Equivalence classes (EC) with return value, output pout, and input constraint on pin.

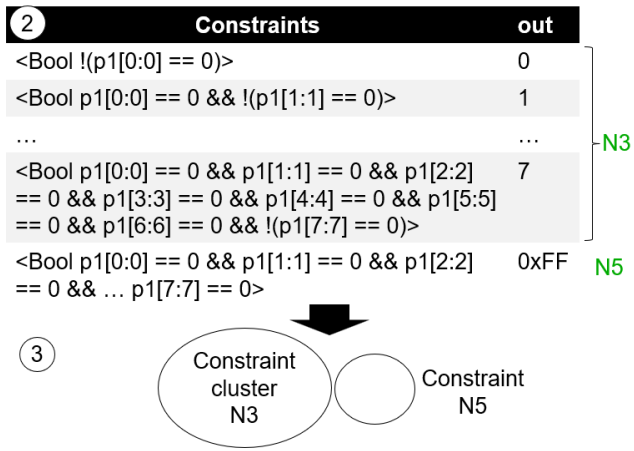
Running Examples – case 2

```
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
}
```









Optimizations Rules:

1 – $CONST == var\ OR\ (CONST + 1) == var\ OR\ (CONST + 2) == var\ OR\ \dots\ OR\ (CONST + N) == var$

→ var $\geq CONST\ and\ var \leq CONST + N$

2 – $(var[0:0]! = 0)\ OR\ (var[0:0] == 0\ and\ var[1:1]! = 0)\ OR\ \dots\ (var[0:0] == 0\ \dots\ and\ var[N:N]! = 0)$

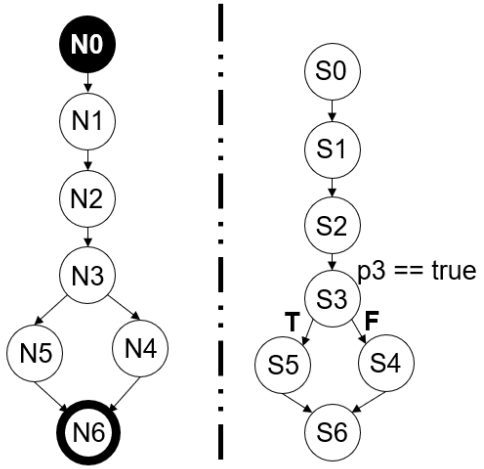
→ $var\ \&\ BITMAP\ ! = 0$

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

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

Running Examples – case 3

```
uint32_t f9(uint32_t p1, uint32_t p2, bool p3)
{ //N0
   uint32_t x = p1; // N1
   uint16_t y = (p1 >> 20) & 511u; //N2
   uint32_t z = p1 & 0xFFFFFU; // N3
   z = (p3 == true ? z + p2 : z - p2); //N4 – N5
   return (y << 20u) + z; // N6
}
```



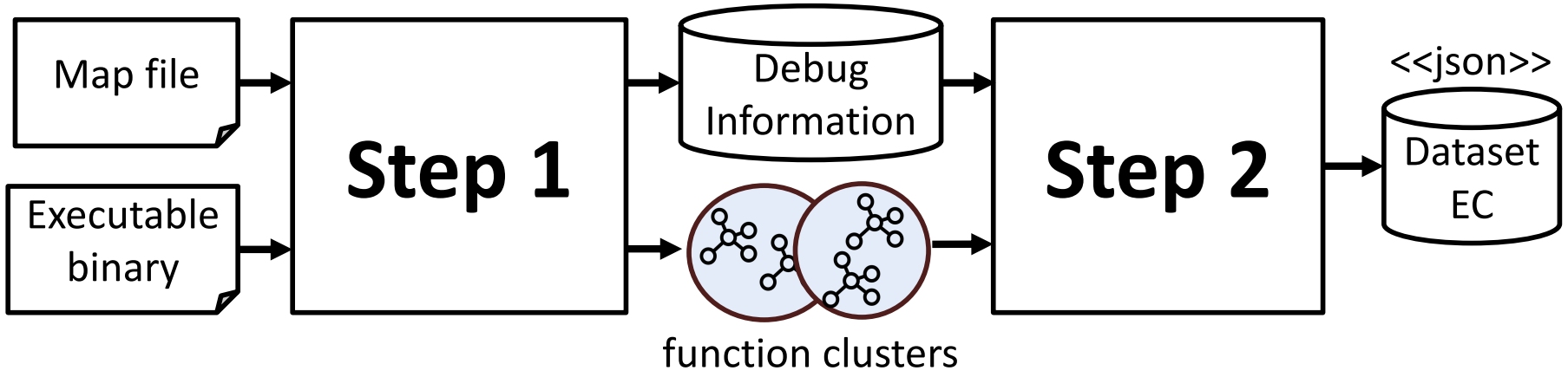
Optimizations Rules:

1 – $0xFFFFFFFF * var \rightarrow -var$

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.

Proposed Solution



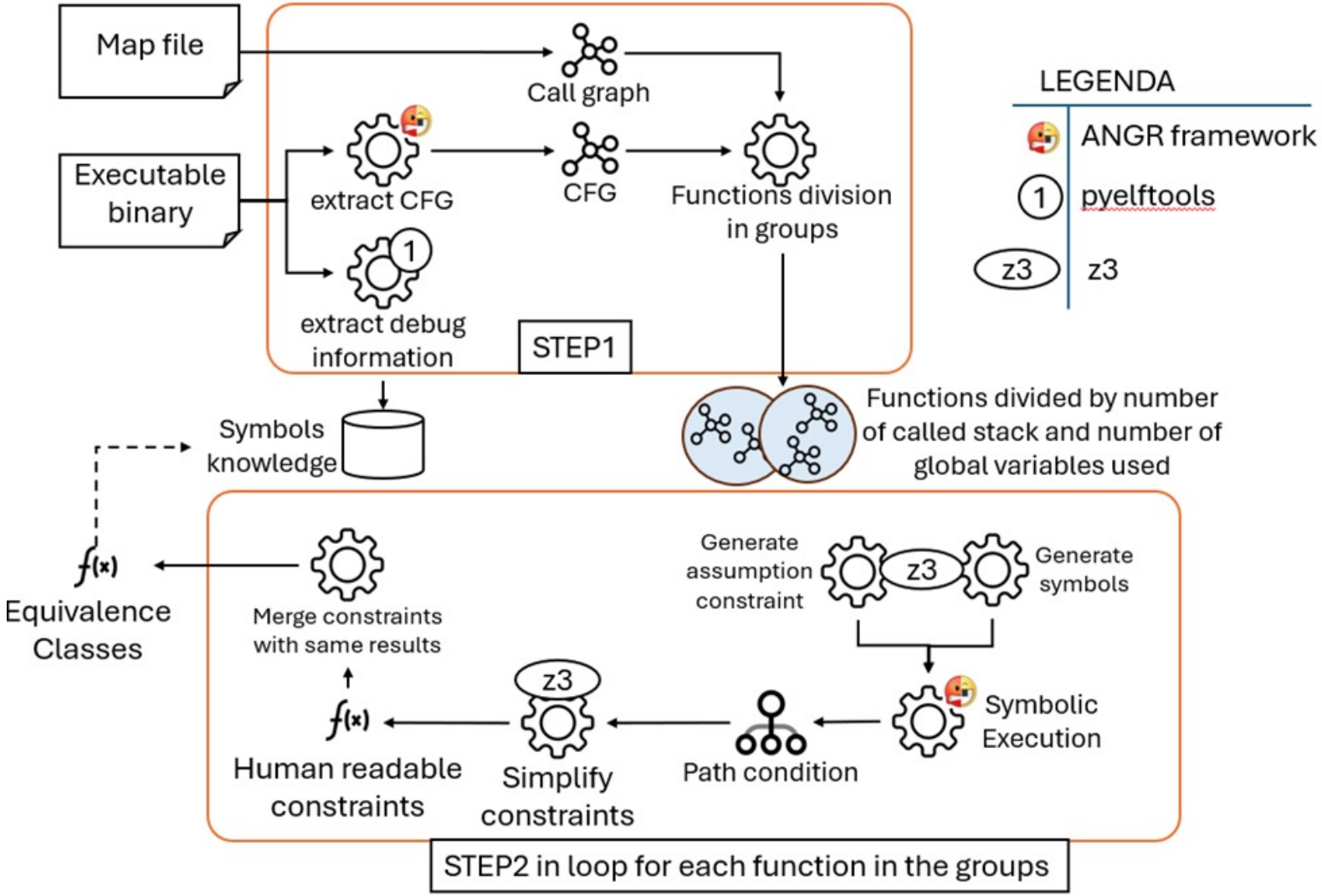
Step 1

- Extraction debug info
- generate function clustering on top of 2 metrics:
  - Num global used
  - Num function called

Step 2

- Generate constraint assumption on inputs
- Perform symbolic execution
- Merge path conditions and simply
- Generate symbolic stub

Proposed Solution



Research results

- ***RQ1:*** To what extent does the tool correctly generate equivalence classes that match the expected functional behavior?
- ***RQ2:*** Are the generated equivalence classes human-readable and interpretable by firmware engineers?

Validation dataset: 75 functions labeled by an industry expert.
Next step: Extend validation through an industry survey to evaluate applicability and effectiveness in real automotive software contexts

Metric	Value
True Positives (TP)	44
False Negatives (FN)	23
False Positives (FP)	7
Recall	0.65
Precision	0.86

Table 1: Evaluation results for equivalence class generation

Next Year

- 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.