Hamburg University of Technology
Institute of Embedded Systems

Master's Thesis

Worst-Case Execution Time Analysis for C++ Based Real-Time On-Board Software Systems

Hazem Abaza

April, 2021

Degree Program: M.Sc. Mechatronics

First Examiner: Prof. Dr.-Ing. Heiko Falk
TUHH Institute of Embedded Systems
Second Examiner: Prof. Dr.-Ing. Frank Thielecke
TUHH Institute of Aircraft System Technology
Supervisor : Dr.-Ing. Zain Hammadeh
German Aerospace Center (DLR)

Abstract

Autonomous systems are today's trend in the aerospace domain. These systems require more on-board data processing capabilities. They follow data-flow programming, and have similar software architecture. Developing a framework that is applicable for these architectures reduces the development efforts and improves the re-usability. However, its design's essential requirement is to use a programming language that can offer both abstraction and static memory capabilities. As a result, C++ was chosen to develop the Tasking Framework, which is used to develop on-board data-flow-oriented applications.

Validating the timing requirements for such a framework is a long, complicated process. Estimating the worst-case execution time (WCET) is the first step within this process. Thus, in this thesis, we focus on performing WCET analysis for C++ model-based applications developed by the Tasking Framework. This work deals with two main challenges that emerged from using C++: using objects impose the need for a memory model and using virtual methods implicate indirect jumps. To this end, we developed a tool based on symbolic execution that can handle both challenges. The tool showed high precision of nearly 90 % in bounding loops of the Benchmark suit. We then integrated our advanced analysis with an open toolbox for adaptive WCET analysis. Finally, we evaluated our approach for estimating the WCET for tasks developed by the Tasking Framework.

DECLARATION

I hereby declare on oath that the work in this thesis was composed and originated by myself and has not been submitted for another degree or diploma at any university or other institute of tertiary education. I certify that all information sources and literature used are indicated in the text and a list of references is given in the bibliography.

Hazem Abaza

Hamburg, April 08, 2021

Contents

List of Figures

List of Tables

List of Abbreviations

AI Abstract Interpretation

API Application Programming Interface

ASAP As Soon As Possible Scheduling

CFG Control FLow Graph

DSE Dynamic Symbolic Execution

GUI Graphical User Interface

ILP Integral Linear Programming

IR Intermediate Representation

ISA Instruction Set Architecture

LLIR Low-Level Intermediate Representation

SSA Single Static Assignment

WCET Worst Case Execution Time

WCRT Worst Case Response Time

Chapter 1

Introduction

Today's software for aerospace systems is complex. This is due to the increasing number of features and the demand for high safety, quality, and reliability standards. Many design architectures have been adopted to develop these systems efficiently and reliably. The model-based approach is one of the most commonly used design patterns due to its ability to handle the systems' complexity.

The model-based approach involves designing models as elementary blocks which are then used to develop the complete software for the embedded system through automatically generated code. Following this pattern improves the productivity and ensures the software's correctness as the applications are designed in systematic and error-free methodology. MATLAB/Simulink and Stateflow are the most commonly model-based frameworks used in designing aerospace applications [30] [81] [61] [57] [45].

Due to the complexity and the dynamic orientation of the developed models, its analysis is challenging, especially when considering timing aspects. The most commonly used approaches for estimating WCET for model-based application are the integration and the measurement approaches. In the integration method, timing analyzer is inserted in the model workflow to trace the execution time of each functionality. This approach was used in various projects as [38] [76] [55] to predict the timing behavior. On the other hand, measurement analysis is a technique used to acquire timing information by measuring the execution time of the code executed dynamically on the target hardware. The analysis was used in several projects as [40] [87] [86] to estimate the worst case execution time.

The Institute for Software Technology at German Aerospace Center (DLR) has developed a model-based framework called Tasking Framework.[1] It is an execution platform for real-time onboard software systems. It allows the implementation of tasks as task graphs with arbitrary activation patterns. It is written in C++ following the event-driven programming paradigm. The Framework has been used for many real-world aerospace non-safety critical applications as in [78] and [44].

Certifying the Tasking Framework for safety-critical applications is a long, challenging process. As for all of its systems, the aerospace industry demands high standards of safety assurance. The most widely used standard for certifying aerospace-embedded applications as well as qualifying the implementation and verifying the tools is the ECS-Q-ST-40C standard [1]. It establishes five design assurance levels, ranging from E-Level, which is the least, to A-Level, which requires extensive

[1] `https://github.com/DLR-SC/tasking-framework`

testing and intensive verification. To qualify the Tasking Framework for C-Level certification, the standard requires ensuring the software's functional safety through proving functional correctness and hazards absence. The demonstrated proofs have to be beyond any reasonable or logical doubts.

To prove correct time behavior, it is necessary to prove that all the real-time tasks meet their deadlines, or that violating the deadlines will not harm the safety of the system. The worst-case execution times (WCET) and later the worst-case response times have to be determined to demonstrate deadline correctness. Performing WCET analysis using the previously mentioned approaches will not offer the required proof of correctness. Static worst-case execution time analysis is a good candidate for this mission. The static worst-case execution time analysis examines the program timing behavior without executing it directly on the hardware. This approach offers safe, and sound WCET and thus can be the first step in the validation process. In this thesis, we investigate the applicability of performing static WCET analysis for C++ model-based application as Tasking Framework.

1.1 Contribution

- **Overview of the state-of-the-art static worst-case execution time analysis tools and techniques**

  We provide an overview of the available static analysis techniques and discuss their suitability for C++ model-based applications.

- **New setup for static WCET for C++ model based applications**

  We introduced a new setup that can over-come the challenges of performing static WCET for C++ model based applications.

- **Comprehensive Evaluation**

  We evaluate our setup using test-cases from Tasking Framework.

1.2 Structure

This thesis is organized as follows. In chapter two, we present the static analysis and its challenges. In chapter three, we demonstrate the Tasking Framework and discuss its applicability for static analysis. After that, chapter four focuses on the available tools' capabilities to perform WCET analysis for C++ model-based use-cases generated by the Tasking Framework. Chapter five presents our new symbolic execution approach for bounding loops and solving indirect jumps. Chapter six discusses the results of our approach. Finally, chapter seven presents our discussion and our future steps.

Chapter 2

Static Analysis

2.1 Introduction

Designing embedded systems for space applications is a challenging process. It requires accurate verification of its performance within its strict deadline. These systems are also called hard real-time embedded systems as they cannot miss their deadlines. Hard real-time embedded systems should respond and must react within particular time frames related to the events in their environment and the systems they control. Therefore, understanding the program execution time's characteristics is a fundamental key for building a reliable real-time system.

In hard real-time systems, the longest execution time of the program that shall occur must not exceed the specified time for the task to meet the system's timing requirements. The longest execution time is called Worst-Case Execution Time (WCET). The exact prediction of the program's WCET is a major issue. As the execution time is not always constant or consistent but instead shifts with various possibilities of events and inputs. These fluctuations happen due to variance in inputs and the software/hardware interactions.

The static analysis's main goal is to determine a given task's dynamic behavior without executing it on the target hardware. Instead, the task code is analyzed, sometimes with some annotations, to get the longest path in the task control flow graph. This control flow information is then combined with an abstract model of the target hardware to calculate an upper bound for the WCET. This is contrary to measurement-based analysis which executes every program fragment to calculate the maximum execution time. The result of the measurements are calculated by forcing the program to vary the initial conditions and inputs through certain control flow paths.

Recently, standard frameworks have been developed to perform a complete timing analysis for the code statically. Although most of them look similar from the general view, they have a lot of conceptual differences in between [56] [33] [37]. This chapter will give a general overview of the static analysis and the most commonly used techniques in each of its blocks.

WCET static analysis is a safe technique to estimate a program's upper bounds for a certain architecture. A complete overview of the analysis is shown in figure 2.1. Briefly, the analysis starts by reconstructing and analyzing the program's control flow graph (CFG). Then, loop analysis and flow facts are defined through the CFG of the program to bound the execution times of each block in the program. After that, The WCET for each basic block is weighted by their execution bounds. Finally, The worst-case execution path and time of the whole function or program are then defined using integral linear programming(ILP)[48].

In the following section, we will briefly present the stages of the analysis and each stage's role in driving the WCET.

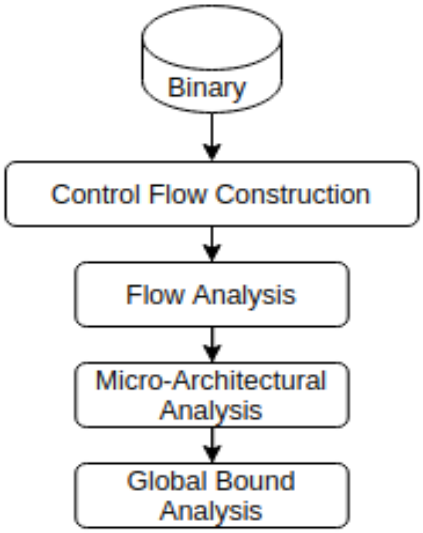


Figure 2.1: Static Analysis Stages

2.2 Static Analysis Stages

2.2.1 Control Flow Graph Reconstruction

The program's flow can be specified by the control flow graph (CFG). This graph consists of two parts: **The basic blocks graph** and **the call graph**. A basic block is defined as a group of instructions that are executed sequentially ended with a jump instruction. The basic blocks graph describes the program's flow and the relations between basic blocks. While the call graph describes only the caller-callee relations [53].

The first step of the WCET static analysis is reconstructing the CFG. Rebuilding the CFG in WCET analysis is challenging ass it starts from the binary level, not from the source code level. The process of reconstructing the CFG started by analyzing the binary and disassembling it. The control flow is recreated from the disassembled instructions and then translated to intermediate representation to be furtherly analyzed.

2.2.2 Flow Analysis

The flow analysis's main goal is to constrain the program paths to narrow the solution domain. It aims to find correlations between variable values, basic block iterations, and certain program paths. In order to achieve this, further analyses are performed as:

- **Value Analysis:** It defines an upper-bound of the values that a register or memory location may hold. This analysis is used later for exact control-flow and data-cash analysis [27].
- **Loop Bound Analysis:** It recognizes loops in the program and attempts to decide upper bounds for the number of a loop iterations [15][63]. Loop bounding problem can be seen as an extension of the halting problem. As it does not only ask whether the program will stop or not, but it also asks after how many times of execution and for this reason no automatic method for loop bounds analysis can give an exact answer regarding the loop bounds. The common used methodology is to identify loop counters, and then bound their starting values through incrementing it to the highest possible value. From this information, an upper bound for the loop iterations can be estimated [34].
- **Infeasible Path Analysis:** It detects sets of possible paths and eliminates the infeasible ones. It uses the result of the value analysis to drive correlations that define the number of executions of a basic block. These correlations are then used to bound the WCET [27].

2.2.3 Micro-Architectural Analysis

. It defines upper bound for the execution of every basic block using an abstract interpretation of the program. The analysis is supported by cash and pipeline analysis to improve its result.

- **Cache Analysis:** It determines upper bound for the cash hit and cash miss at each program point [36].
- **Pipeline Analysis:** It analyzes how instructions pass through the processor pipeline [59].

2.2.4 Global bounds analysis

.
The execution time of a basic block is calculated using execution time modeling. Execution time modeling maps a sequence of instructions to its execution time. One instruction's execution time may vary from one basic block to another based on the input parameters' values, the order of the instructions, and the processor's internal state during the execution. The execution time of a basic block ($wcet_i$) is combined with its loop bounds frequency f_i to determine the execution time of the block **i**. The upper-bound of execution of the program is defined by an implicit path enumeration through the CFG using ILP. The optimization function for path enumeration is modeled as:

$$WCET = max(\sum_{i=0}^{n} wcet_i \times f_i) \tag{2.1}$$

The target function weights the execution time of each basic block i with its frequency bound f. A solution of the ILP maximizes the sum of these weights with the path constraints and corresponds to an upper bound on the execution times [72].

2.3 C++ Static Analysis Challenges

Performing static analysis is for C++ is challenging whether it is done on the binary of the source code level for the below reasons [54]:

- **Language Evolution at a Breakneck Pace**

  Language evolution now, unlike earlier stages of C++, is much quicker. The language committee's decision for a three-year standardization cycle leads to new standards, e.g., C++14, C++17, and now C++20 in a shorter time span [2].

- **Language Front-end**

  To accommodate all language features, C++ requires a complex front-end. On the other hand, the C front-end is simpler to implement and validate. In comparison with C, there are only a few front-ends that support modern C++.

- **Hidden Complexity**

  C++ offers abstraction level that facilitates the code development process and improves the production quality. The abstraction and productivity come at the price of hidden underlying complexity. This complexity affects the analyzability of the code. For example, The standard libraries depend heavily on dynamic memory allocation, increasing the risk of having unpredicted dynamic allocation scenarios. Although the standard containers has easy-to-use data structures as maps, they are highly dynamic and hard to analyze.

- **Dynamic Memory Allocation**

  C++ depends heavily on dynamic memory allocations. The life-span of objects in the heap is not limited by the scope in which the initialization has occurred. Thus, the de-allocation of such objects is done outside their initialization context. Verification and validation of C++ with objects in the heap is challenging as it requires information about each object's life-span, which depends on the input data to the program.

- **Dynamic Polymorphsim**

  Dynamic polymorphism also called dynamic dispatch means that function calls are resolved during the run-time which implemented in C++ through virtual methods. This mean that the CFG is dependent on the flow of the data in the program. As a result, precise data-flow analysis has to be implemented to assist the control-flow analysis to resolve the dynamic dispatch [89][24].

- **Error handling**

  C++ handles errors through exceptions. The success of throwing exceptions is not always granted because most C++ compilers generate code which indirectly calls malloc to allocate heap memory for the exception item. Exception handling magnificently increases control coupling which occurs by adding indirect control path from each program point to the exception handlers. Therefore, not only the analysis complexity increases but also the accuracy of the analysis deteriorates. Analyzing exceptions is even more complex at the binary code level.

2.4 Static Analysis Challenges for C++ Safety Critical applications

In this thesis, we are only interested in analyzing a subset of C++ that fits safety-critical applications. Analyzing applications that are developed following the MISRA [3] standard down-grade the complexity of the analysis and automatically expel some of the previously mentioned challenges. This thesis aims to perform the static timing analysis for application as Tasking Framework, which is developed following safety-critical standards. For example, The most common features of the MISRA standard are:

- Single Point Function Exit (MISRA Rule 6-6-5)
- No Heap Memory Allocation (MISRA Rule 18-4-1)
- No Recursion (MISRA Rule 7-5-4)
- No Exceptions (MISRA Rule 15-0 -1)

Hardware requirements for safety-critical applications also down-grade the complexity of the problem dramatically. For safety critical applications, we can safely remove the cache and the pipeline analysis from our considerations. As due to their non-deterministic behavior, caches and multi staged pipelines are not allowed to be used in on-board systems. As a result, static WCET analysis challenges for C++ safety critical applications can be narrow down into two main problems:

- **Control Flow Reconstruction due to indirect jumps**

  In C++ code, indirect jumps result mainly from virtual methods. They ensure that the correct function is called for an object to achieve polymorphism, which is resolved during run time. Such behavior is translated on the binary level to indirect jump instructions which are branch instructions whose values are only known during the run-time, not during the compilation. These branching statements are challenging for static analysis as they cease the reconstruction of the CFG during the static analysis. For example, at the end of a basic block, branch instruction like the one in listing 2.1 will lead to the analysis's failure as the analysis will not be able to determine the branch target of the instruction statically.

Listing 2.1: Branch Instruction.

```
1 add r1, r2
2 mov r2, #0x80
3 b r3
```

- **Loop Bounding**

  Defining loop bounds is an essential step to get accurate WCET. This step is challenging for C++ code for the many reasons to name a few:

  - Most C++ loops iterate over containers as vectors. Information about the container's size and its location in the memory is essential to bound the loop. This information is not available on the source code level and requires binary analysis to extract.

- In C++ begin_expr and end_expr refer to the iterator start and end points respectively. As for C++17, the begin_expr and the end_expr may have different types, additionally the type of the end_expr does not have to be an iterator. Such behaviors increase the complexity of the analysis on the source code level [2].
- C++ loops may trigger deep copy during iteration if used with non-constant iterators affecting the code's execution time dramatically. This behavior is difficult to analyze on the source code level [2].
- Some C++ loops are hidden behind the constructor. Constructing n objects from the same class sometimes is translated to loops on the binary-level. These loops are hard to detect on the source-code level.

2.5 State of the Art Static Analysis Tools

Many static analysis frameworks have been developed to give a precise WCET estimations. **To the best of our knowledge, there are no frameworks that have been dedicated to performing WCET analysis for C++ code**. This section will give a complete overview of existing relevant frameworks, their capabilities, and limitations. At the end, we will summarize their potentials to perform static WCET analysis for C++ code.

2.5.1 Bound-T

Bound-T performs static WCET analysis on binary code. The tool design includes target-specific modules to decode the input files beside generic modules to reconstruct the control-flow graph. To determine loop bounds, Presburger Arithmetic is used to model the loop counters arithmetically. Finally, Integral Linear programming (ILP) is used to determine the worst-case execution path (WCEP) [50].

Bound-T has some challenges in its analysis techniques, for three main reasons [4]:

- **Lack of pointer analysis**
  The absence of pointer analysis affects the accuracy of the complete analysis. The results can only be relevant if it is guaranteed that the analyzed program has no pointers that may affect the program's control flow.

- **Variably-sized memory access**
  The whole CFG can be wrong if the program under analysis is accessing the same variable in different ways (sometimes as a word and others as a byte).

- **Overflow or wrap-around wrong estimations**
  The main tool that Bound-T uses to analyze CFG and loop bounds is based on Presburger Arithmetic. Presburger Arithmetic is a weak model for computer Arithmetic. It solves equations with integer variables assuming that these integers are bounded and can never overflow.

2.5.2 SWEET

The WCET research group has developed SWEET (SWEdish Execution Time tool) at the Mälardalen Real-Time Research Center. The tool was designed following the classical WCET static analysis architecture with flow analysis, low-level analysis to get the WCET estimation. The heart of the tool is the flow analysis, which is based on Abstract Execution. Abstract execution is an abstract interpretation through which loops and function calls are analyzed and executed separately in the abstract domain with abstract variables values. These values correspond to the real values that the variable may hold during execution [33]. The input to SWEET is intermediate representation in ALF format. ALF is an intermediate representation (IR) developed to facilitates the flow analysis [41].

The major functions of SWEET tool are [58]:

- **Portable analysis across different formats**
  In order to keep SWEET portable, all analyses are performed in ALF format. Thus, before using SWEET, any format must be translated to ALF.
- **Automatically annotate all program variables**
  The abstract execution model supports the tool to annotate each variable within the input source code through value analysis. SWEET abstract execution model outputs annotation log for every loop and variable in the input ALF file.

Despite its capabilities, SWEET has mainly two limitations:

- **There is no C++ to ALF translator**
  To the best of our knowledge, although it was mentioned in [41], SWEET can not take C++ programs as input. Such a translator has not been developed yet, which makes the tool only accessible by C programs or PowerPC binaries [62].
- **Recursive functions can not be analyzed**
  The SWEET tool will fail to analyze recursive function calls, making it difficult to be used in sophisticated industrial applications [58].

2.5.3 Chronos

The Chronos works on C code. If loop bounds are not annotated, data flow analysis is performed at the beginning to determine loop bounds. The core of the framework performs the analysis on the binary level. To this end, the binary is disassembled to reconstruct CFG. Timing of each basic block from the micro-architectural analysis is then combined with the loop-bound data to estimate WCET. One of the greatest advantages of Chronos that is it supports modeling of pipeline and branch prediction [60]. Chronos will be a good candidate for C++ code analysis If we manage to change the Chronos C-based front-end. This is possible as it is an open-source project where the source code is accessible for doing investigative experiments. However, the framework shows only one limitation which is the difficulty in analyzing recursion [77].

2.5.4 OTAWA

The research group of real-time systems developed OTAWA at the University of Toulouse to integrate all kinds of open-source WCET analysis. The main goal is to develop a base framework that supports experimenting state of the art WCET algorithms. To this end, OTAWA architecture is a little bit different from the previously mentioned frameworks. As shown in figure 2.2, the base of OTAWA is an architectural abstraction. Although it supports the Instruction Set Architecture (ISA) for multi-architectures, It hides the ISA information and only reveals information relevant to the WCET analysis. This facilitates the framework re-usability for different architectures [16].

The WCET estimation performed by OTAWA starts with the binary decoding and program reconstructing to get CFG and all basic blocks of the program. The heart of the framework appears in the next phases, where two kinds of analysis are performed: program and hardware analysis. The program analysis includes value, loop, and path analysis, while the hardware analysis includes cash analysis, pipeline, buffer analysis, and every processor feature that may impact the execution time. An execution graph is then created to collect timing information from the static analysis for every basic block resulting from program and hardware features. All these data are then encoded as ILP problem constrained with loop bounds to get the WCET.

OTAWA showed good results in the comparison of the WCET tools challenge [84]. The tool was able to handle dynamic function calls and recursion. The loop bound analysis tool of OTAWA showed good performance in detecting static, arithmetic, and infinite loop bounds. However, this was only tested on C code, not C++.

2.5.5 aiT

ait is the industry-standard static analyzer. The tool is used to inspect the developed applications' timing behavior and provide a tight upper bound for its WCET. It is based on Abstract Interpretation which offers a proven accurate relation to the architecture's semantics. The results of the ait analysis are deterministic for all executions (including cache and pipeline behavior). The tool depends mainly on static analysis for resolving indirect jumps and bounding loops. If the tool failed, annotations for loops and branch targets are provided in separate files by the application developer. Besides, ait covers a wide range of hardware architectures with high precision in estimating the WCET. Additionally, ait's GUI offers deep insight into the processor's functionalities, which can be used for further program optimizations [5].

One of the institute of software technology at DLR long term goal is to proof that it is safe to design safety-critical model-based aerospace applications using C++ instead o C or Ada. To present the findings to the research and the industry community a decision was made to use open source tools through this process and thus closed source analyzers as aiT are out of the scope of this thesis.

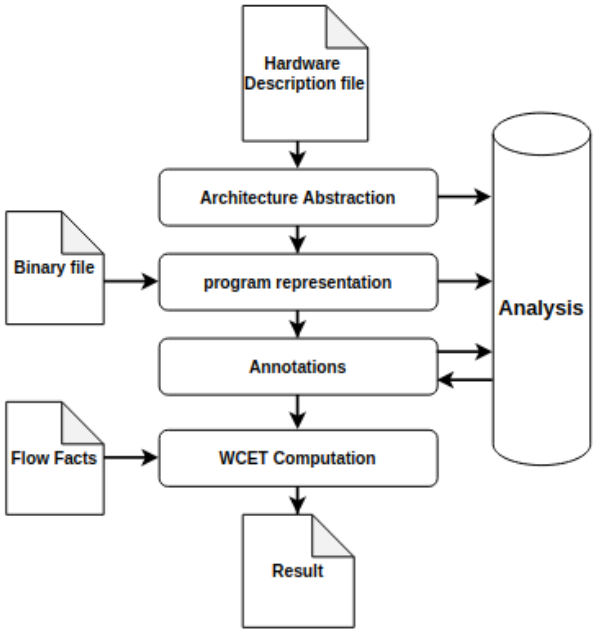


Figure 2.2: OTAWA Architecture

2.5.6 Summary

Many frameworks have been developed to estimate WCET. The common idea behind most of them is based on reconstructing the CFG of the executable then combining it with the micro-architecture analysis in order to estimate the WCET of each basic block [16][4][60]. Table 2.1 summarizes the tools' abilities concerning different criteria. Besides showing that some tools may have the potential to analyze C++ code, our exploration also shows that some tools do not contain loop bound analysis as a part of their structure. Thus, in the next sections, we will explore the available loop bound analysis techniques and their conveniences for C++ code.

Table 2.1: WCET Frameworks Comparison

WCET Tool	Tool Input	Loop bound analysis	Applicable for C++
Bound-T	Binary	✗	✗
SWEET	ALF IR	✓	Maybe
Chronos	Binary	✓	✗
OTAWA	Binary	✗	Maybe

2.6 Loop Bound Analysis

In various static program analyses, information about how frequently a loop is iterated is important. A correct static WCET analysis is based on safe upper bounds of loops. An automated analysis must be adopted to get safe WCET estimations to skip the error resulting from manually annotating loop bounds. In this chapter, we will give an overview of the available tools for loop bound analysis. We survey the annotation languages by investigating their strengths and limitations.

2.6.1 Pattern-Based Technique

Pattern-Based Technique was first introduced in [46]. The analysis uses predefined patterns to trace loop variants. It provides algorithms to bound loops with multiple exits on the assembly level with and without using annotations. However, The analysis focused only on instructions within the loop basic blocks making it difficult to analyze loops with external parameters [63].

As Pattern-Based approach is not only compiler-dependent, but it also depends on the level of code optimization. The pattern-Based approach was supported in [28] with data-flow analysis to enhance its abilities to deal with more complex loops structures. Such approach fails in resolving all kinds of loops completely. User annotations must be provided to ensure safe loop bounds.

The pattern-Based approach is implemented on the binary code. Binary code is the code that is executed, and thus it carries all the relevant flow information. However, depending only on low-level information to determine the loop bounds may lead to high-level data loss. For an effective design, loop bound analysis tools working on the executable level must maintain information as CFG and annotations in the source code [42]. aiT [5], the standard static WCET analyzer, uses this approach to provide tight loop bounds. Although applying the pattern-based approach on the binary level seems to have promising results, it does not provide a generic solution to determine loop bounds.

2.6.2 Slicing and Abstract Interpretation

The approach is based on the fact that each new iteration updates the state of the executing program. Then, bounding the number of states will deliver an estimation for the loop upper bound. Since these states may have identical program flow, counting them will lead to the maximum number of values for loop variables [34].

This analysis is developed using standard program analysis:

- **Program Slicing**
  It is introduced in [85] is a program analysis that analyzes part of the code that affects certain computation functionality. A slice of a given program regarding certain slicing criteria is a subset of the program that contains all instructions affecting this property. In loop analysis, slicing is used to define variables and statements that are relevant when estimating loop bounds.

- **Abstract Interpretation**
  Abstract Interpretation is an approximation for the semantic model of the program under analysis. This sound approximation which is first formalized in [27] is mainly used in undecidable problems as static loop analysis. Loop bound analysis uses it to define the upper value of the variables affecting loop bounds at every program's execution point.

Various loop analysis tools introduced this approach on the intermediate representation level (IR) [34][22]. The intermediate representation code is generated by the compiler for purposes of transformation and optimization. The IR is semantically equivalent to the program being executed with the following characteristics:

- Smaller number of instructions
- Type safe
- Portable as it is independent of the compiler and target architecture
- Accurate and efficient for static analysis as control-flow, symbolic execution and graph isomorphism

These characteristics make it a strong candidate for all kinds of flow analysis [35].

As a part of SWEET [33], The loop bound analysis tool [34] was introduced to work on the ALF IR. As mentioned before, there is no C++ to ALF translator. Some trials have been done to translate the LLVM IR into ALF IR, which opens the door for all languages using the LLVM compiler's front-end. However, this approach failed because of the unavailability of a complete, accurate translation to LLVM [82].

2.6.3 Polyhedrons

In addition to the Slicing and Abstract Interpretation, many loop analysis frameworks [63][66][15] supported their techniques with polyhedron models to get effective and precise computation. A polyhedron(P) is an N -dimensional figure bounded with a set of linear inequalities such that:

$$P := \{x \in Z^N | Ax = a, Bx \geq b\} \tag{2.2}$$

Where $A, B \in Z^{mN}$, $a, b \in Z^m$ and $m \in N$ [35][63].

The polyhedral can precisely represent N-dimensional nested loops as N-dimensional polyhedron shape. Driving correlation between outer and inner loop is challenging and sometimes impossible, Yet, polyhedron makes it easier and less time-consuming. Consider the code in listing 2.2.

Listing 2.2: Nested loops modeled using polyhedron

```
1 for(int i = 1; i <= 4; i++) {
2     for(int j = i + 1; j <= 6; j++) {
3         if(j > 4) {
4             //Condition;
5         }
6     }
7 }
```

To model this loop using polyhedron, the outer and the inner loop bounds has to be expressed as following:

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \geq \begin{bmatrix} 1 \\ -4 \end{bmatrix} \tag{2.3}$$

$$\begin{bmatrix} -1 & 1 \\ -0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \geq \begin{bmatrix} 1 \\ -6 \end{bmatrix} \tag{2.4}$$

The polyhedron a in figure 2.3 is modeled based on the two previous equations where each (i, j) point in the figure corresponds to a single iteration in the loop. Therefore, by determining the number of those pair points in the polyhedral area, we will restrict the loop iteration domain and obtain the number of iterations.

The polyhedron is also able to define other scenarios with branch conditions. Polyhedron b in figure 2.3 shows a new polyhedron based on the branch conditions in the code. The dotted line in polyhedron b in figure one shows the trimmed part from the polyhedron due to the code's branch condition [66].

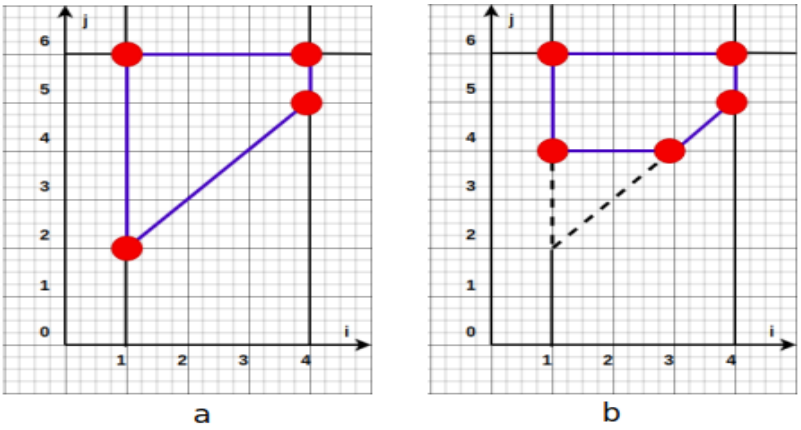


Figure 2.3: Loop Formulation with Polyhedron

2.7 Loop Bound Analysis Tools

The following section will survey the available tools useful to bound loops of a C++ code. The survey will discuss the analysis used by the tool besides its strength and weakness concerning C++ loops.

2.7.1 r-TuBound

The pattern-based approach was introduced on the source code level in the r-TuBound loop analysis tool [56]. The tool's main goal is to build a control-flow graph for variables from extracted loops in the C/C++ source code. Applying various analyses and then translating these extracted loops into the form mentioned in listing 2.3 where *a, b, c, d* are symbolic integer-valued that are independent of *i*, $g_1..g_m$ are boolean expressions and $f_1..f_m$ are non-constant linear integer arithmetic functions independent of *i*. Based on the WCET 2011 tool challenge, the tool lacks specifying variable input ranges, it supports limited number of architectures and it also fails to analyze a C++ loop like the one in listing 2.4.

Listing 2.3: r-TuBound Transformation equation

```
1 for (i = a; i {<,>} b; i = c*i+d) {
2 i = f_0(i);
3 if (g_1)i = f_1(i); else i = f_2(i);
4 ... ;
5 if (g_m)i = f_m(i); else i = f_m(i);
6 }
```

2.7.2 oRange

oRange, is a very powerful loop analysis tool. It applies all kinds of analysis mentioned above to supply the user with two values, max, and min, representing the loop's upper and lower bounds. Unfortunately, oRange analyzes only C source code, making it so hard to utilize for our case. The oRange workflow starts by detecting loop indices and constructing symbolic expressions of the loops. A syntactic tree is then built based on abstract execution to combine the loop running context. Loop bounds are then formulated in the form of symbolic expressions. Finally, the resulting tree is resolved to the flow-fact numerically [18].

2.7.3 Polymas

The tool was built on polyhedron with an abstraction execution concept to tackle loop bounds. It analyzes the binary code running on the machine, which removes the C++ overhead and complexity. The tool proposes analysis based on polyhedrons to define a subset of registers and memory locations that may be effective to define loop bounds [15]. The tool was tested using Mälardalen WCET benchmarks in [17] however; the selected tests were simple loops with simple branching statements [43]. The tool failed to analyze simple C++ code with pointers as in listing 2.4, which makes the tool incompatible with our goal.

Listing 2.4: C++ loop over array

```
1 int main() {
2   int aNumber [] = { 16, 2, 77, 40, 12071 };
3   int counter = 0;
4   for ( int num : aNumber ) {
5           if (counter > 3) {
6                   counter ++;
7           }
```

2.7.4 PAGAI

PAGAI is a tool to automate static analysis with input as LLVM IR [6] and an output as a list of primary constants for a certain control node in the source code. The tool provides control invariants based on user given assumptions or assertions using C/C++ standard macros. The tool checks the failure of a given assertion and indicates a warning in unsuccessful input. The execution of the tool terminates after falsifying the given assumption/assertion, and thus it can be used to check loop's variant values. The PAGAI framework is mainly based on abstract interpretation and convex polyhedrons to perform relation analysis between control-flow nodes [47].

2.7.5 Summary

Loop bound analysis is critical for sounding WCET analysis. In the previous sections, we explored the available loop bounds analysis tool focusing on their ability to deal with C++ loops. Table 2.2 summarizes the tools' abilities for different criteria. We found that most of the tools are based on abstract interpretation (AI) and polyhedron as main analysis techniques. In chapter four, we explore how we can use such tools to serve our goal. We integrate WCET tools with the loop-bound tools to match their input types and their analysis abilities.

Table 2.2: Loop Bound Analysis Tools Comparison

Tool	Analysis Input	Analysis Technique	Applicable for C++
r-TuBound	C/C++	Pattern-matching	✗(limited C++ patterns)
oRange	only C	Slicing, Polyhedron, AI	✗
Polymas	Binary	Polyhedron, AI	Maybe
PAGAI	C/C++	Polyhedron, AI	Maybe

.

2.8 Reconstructing The Control Flow Graph

Reconstructing the control flow graph from the binaries is challenging due to the presence of indirect jumps. An indirect branch has a dynamically defined target that may point towards number of locations in the program, while a direct jump has a statically specified target that points to a single location in the program [24]. Indirect jumps are commonly used to realize dynamic program behaviors by implementing common programming constructs, such as virtual function calls and calls through function pointers. Although indirect jumps are common and useful, due to their dynamic nature, it is usually difficult to resolve the target of an indirect jump through static analysis. This leads to inherent challenges in constructing the complete CFG [90]. In C++ binaries, indirect jumps are most commonly due to virtual methods where the target is calculated at run-time. Failure in resolving indirect branches leads to incomplete CFG and thus hinders the abilities of the static WCET.

Resolving indirect jumps falls into three categories:

- **Static Analysis**

  During the analysis, the behavior of the target program is analyzed without concrete execution. The approach has the advantage of low cost and high coverage for the whole program. Thus, it is commonly used in industrial reverse engineering tools as IDA Pro [7]. However, this approach always leads to incomplete results due to the difficulty of resolving indirect jumps statically. The most commonly used analysis in the static approach is the data-flow analysis.

  Several data-flow analysis techniques have been used to resolve indirect jumps. However, data-flow analysis requires a reliable CFG to analyze. This paradox situation is know in the literature as "chicken and egg" problem [79][73]. Program slicing is the most commonly data-flow analysis technique used to recover indirect jumps. It is a methodology for determining the set of statements of a program that potentially affects a variable's value at some point in the program. The goal of the analysis is to find all the possible influences on variables. During slicing analysis, indirect jumps are retrieved from the binary code using the classic source-destination algorithm, in which the instruction I' is said to be the lexical successor of instruction I if deleting I force the CFG to pass to I'. The technique starts with slicing the code at the indirect jump instruction, and then it performs expression substitution to high-level language. After that, it checks for pattern matching against indexed branch normal forms. Although some modifications were introduced, the proposed recovery heuristics were nearly used by most of the frameworks [25][26][32]. Despite the promising results with some tool-chains, the approach still result in incomplete CFG.

- **Dynamic Analysis**

  In dynamic execution, the program is executed in order to extract the exact control flow graph. The idea is to execute programs on a set of test-cases and extract the control-flow information from the execution traces [90]. The approach can resolve a certain of indirect jumps and discover precise control flow. However, the completeness of the constructed CFG by this approach depends on the test-cases' capability to cover indirect jumps.

Traditional dynamic analysis techniques cover only a small portion of program execution paths. To improve code coverage, forced execution was introduced in [88]. In Forced execution, the code is executed semantically to explore the directions at each branch point and to compute the targets of indirect branches at run time in a scale-able way. For the same reason, The state of the art platform for resolving indirect jumps **Syder** [83] introduced symbolic dynamic execution support by program slicing to resolve indirect jump. Dynamic symbolic execution is an approach to systematically identify the program execution given an input value. The proposed approach was able to resolve indirect jumps in real world application with high accuracy and reliability.

- **Hybrid Approach**

  Hybrid analysis has been used to resolve indirect jumps and to enhance the incompleteness of static analysis. In the hybrid approach, a set of seed tests are combined with statically computed jump targets to reconstruct the CFG. However, the results' completeness depends mainly on the generation of test-cases that can handle indirect jumps. Gray-box fuzzing has been introduced to generate test-case to handle indirect jumps. Gray-box fuzzing is an approach for software testing that is both salable and functional. It is commonly used in vulnerability identification and the generation of test-cases. Current gray-box fuzzing normally creates test-cases using an evolutionary algorithm based on input information from the execution. Evolutionary algorithms are randomized search heuristics that can be used to solve a variety of problems. They are governed by a number of criteria that are critical to the search's effectiveness [67]. In summary, the hybrid analysis is a promising approach to construct a more complete and precise CFG. Generating test-cases that handle more indirect jumps is the core of a successful hybrid approach. In the state-of-the-art approach to constructing CFGs [14][68][89], the test-case generation still faces the problem of unreliable test-cases that lead to incomplete coverage of the CFG.

Chapter 3

Tasking Framework

3.1 Motivation

Space missions are in continuous demand for computing resources to process more complex data or execute efficient algorithms. Missions like Rosetta [8] or landing of the Mars rover was designed based on lists of commands to control landing and maneuvering to reduce power consumption instead of trajectory control advance algorithms which required more power consumption.

During the development of the TET-1 satellite mission (Technology demonstrator) and Bi-spectral Infrared Detection (BIRD) missions, the estimator and observer control modules were computed in a fixed manner (order and time). The module's total computation time was the waiting time for the sensors' data plus the additional latency to ensure complete data arrival before computation. This model leads to a timing problem in the complete control cycle due to the latency overestimation. This has not been discovered until the launching where timing violation occurs in another bus application leading to changing the order of the computed tasks in a way that resulted in corrupted data and faulty behavior in the attitude and the complete orbit control system [65]. Hence, arises a need for a conceptual framework, which allows sharing resources based on certain configurations for all flight phases.

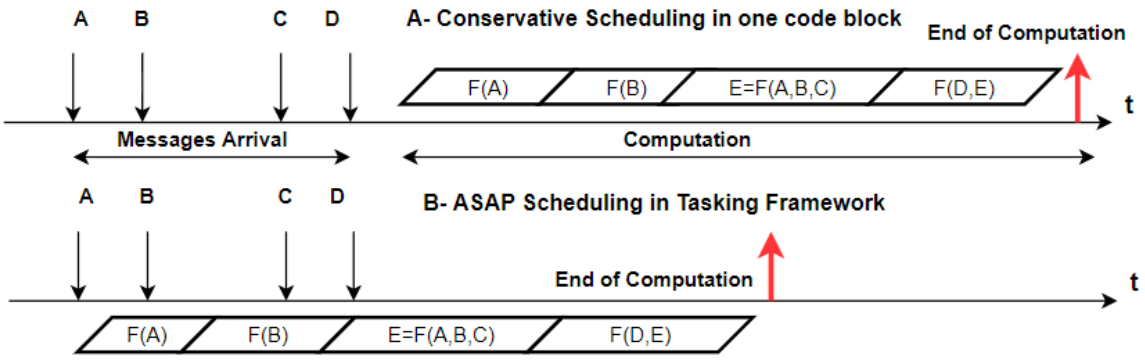


Figure 3.1: Tasking Framework Scheduling

The starting goal of the On-Board Computer–Next Generation project by DLR was to design qualified processing and network nodes with an operating system that ensures timing behavior for satellites. This design has also to cover the timing behavior of multi-cores and distributed systems. The core element in this design was the Tasking framework. The framework was designed mainly to improve the attitude control systems performance by dividing the computational data from the sensor into small parts, each of them called task which are then scheduled based on their availability.

Tasking Framework was designed as an application of inversion of control design pattern, which is used in designing lightweight frameworks. The framework was firstly used in the ATON (Autonomous Terrain-based Optical Navigation) project, which is a navigation system for a moon landing scenario using a couple of image processing algorithms [65]. The most important feature of the framework is changing the time behavior of the tasks being computed. Figure 3.1 shows the effect of following ASAP scheduling policy on the total response time instead of conservative scheduling in which computation is started at a predefined time in the computation cycle.

3.2 Task-Channel Application Model

Tasking Framework was developed based on the task-channel model introduced in [39]. The concept is to have a separation between the data and functionality. In this model, A **task** is a stateless executable program, together with its memory and I/O ports while a **channel** is a message queue that connects one task's output port to another task's input port. In Tasking Framework, the channel represents a data container that can be handled through the task object. As shown in figure 3.2, it acts as an interface between tasks and interfaces with the software inputs and outputs. Using a task-channel model improves code re-usability. It is also very useful in distributed systems where some software components are needed to be migrated across computation nodes.

Tasking Framework is designed to tackle data-flow-oriented applications. In data-flow designs, the system's functionality is comprehended by examining the data flow through the system. Data-flow oriented techniques imply that the system's input data must be identified and processed to produce the required outputs. Using this approach, the program is modeled as a series of sequential operations happening in a certain order. Tasking Framework uses this design pattern to introduce structural API that does not depend on the data's availability but only on its sequence. **Except for the Execution class APIs**, All APIs introduce a high level of abstraction, making them independent of the input data's availability and the processed task. The framework can be seen as an abstraction layer similar to the operating systems that control the whole process in the abstract, generic, and deterministic way [44].

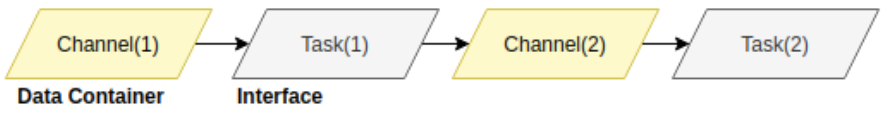


Figure 3.2: Task Channel Model

3.3 Execution model

In Tasking Framework an instance of a task τ is activated when all the inputs to the task are activated. For example, in figure 3.3 Task **A** will be activated when input #1 is activated right after receiving Msg.A from the sensor A. Another way to activate a task immediately is to mark one of its inputs as final. If this input is activated, the task will run without considering the other inputs' state. For example, in figure 3.3 Task **E** will be activated when the task event (Timer) marks the input #0 as final. After that **C** will be activated instantaneously. An example on how timing parameters as task activation times and scheduler policy are specified in Tasking environment can be found in listing A.1.

Figure 3.4 describes the sequence diagram of Tasking Framework. When a message arrives from a sensor, the main execution thread calls the **push()** method from the channel class to notify the associated inputs. If all task inputs are activated, The Tasking Framework will immediately signal a thread by calling **perform()** to execute the pending instance of this task. The task starts immediately with the scheduler of the framework. If a free resource, for example, a processor core, is available, the task will start immediately; otherwise, the task is queued [44][65].

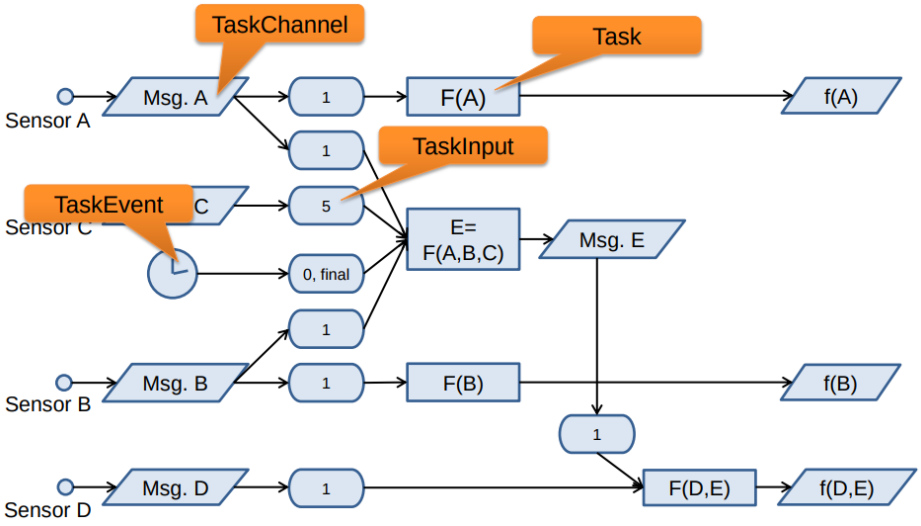


Figure 3.3: BIRD - Attitude and Orbit Control System (AOCS) and Tasking Framework Elements

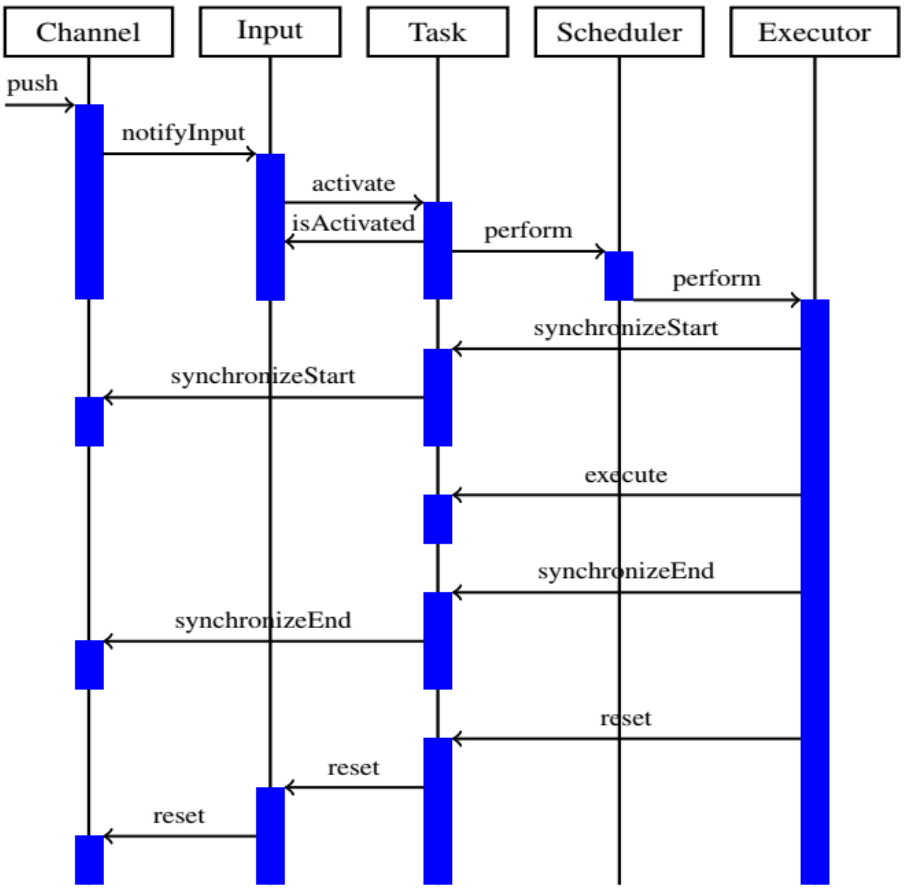


Figure 3.4: Sequence Diagram

3.4 Tasking Framework in application

Tasking Framework has been used in some DLR projects. The following paragraphs highlight 3 projects in which Tasking Framework has the main role. In Autonomous Terrain-based Optical Navigation (ATON) [78], Tasking Framework was used to implement the different functionalities as tasks and connect them via channels. Channels are data-containers storing the data, and component are activated periodically using events. The developers used 4 threads to execute the software on the prototype flight computer. In Euglena Combined Regenerative Organic food Production In Space project (Eu:CROPIS) [64], the Attitude and Orbit Control System (AOCS) was develoed using Tasking Framework. In Scalable On-Board Computing for Space Avionics (ScOSA) [80], a DLR project proposes and tests a new onboard computing architecture based on reconfigurable interconnected commercial off the shelf processors combined in one distributed system. Tasking Framework is used as a part of the middleware and the main API to develop the applications that run on ScOSA. An example of an application that will be developed using Tasking Framework to run on ScOSA is Onboard Data Analysis, Real-time Information System (ODARIS) [74] and Rendezvous Navigation [71]. Although Tasking Framework has been used in several applications, it has not been used for developing C-Level safety-critical tasks. Through all the previously mentioned projects, the WCET was estimated through extensive measurements in the operational environment.

3.5 Tasking Framework as C++ developing library

Tasking Framework is developed at the Institute for Software Technology, German Aerospace Center (DLR). It is an event-driven execution platform for real-time on-board software systems. It allows the implementation of tasks as task graphs with arbitrary activation patterns (periodic and sporadic). It is written in C++ following the event-driven programming paradigm and supports multi-threading programming [44].

Although it is not the norm for aerospace application to be developed in C++, it was used in the developing of this framework for the below reasons:

- First, The language offers modularity through the use of classes and its object-oriented functionality. Although C can also provide similar implementation to classes through struct, C++ outrages it due to constructors and deconstructors. Constructors and deconstructors guarantee that objects are always correctly initialized and are always removed after usage, preventing bugs and leakages.

- C++ allows programming abstractly and generically through templates. A template can be simplified as a macro that produces a complete full functionality for a different data type. Templates reuse algorithms for different data types efficiently and at a low resource cost.

- Type safety is another reason to use C++. Type safety means that the compiler ensures that no mix in data types occurs and that all the variables are correct. Unlike C++, C function memcpy can copy memory of double into an array of char and ending up with meaningless data.

- In large projects, using a unique descriptive naming is hard to achieve and it has always added prefixes to the names. This leads to long and unreadable names. C++ offers the simplest solution using namespaces. Namespaces allow multiple uses of the same name in different contexts, which are resolved later during compiler time. This C++ functionality ensures the multiple usages of the name in a descriptive, unique way.
- Unlike C, which uses malloc() and free() to allocate and free memory regions, C++ uses new and delete that adds constructors and destructors, ensuring that no memory leaks or bugs will occur.
- C++ introduces new concepts that do not exist in C as virtual functions, inheritance, operator overloading.
- References and smart pointers in C++ are safer than normal C pointers as they ensure that pointers can not be uninitialized or referenced to NULL.

3.6 Tasking Framework and Static analysis

In this section we discuss the applicability of static analysis to the Tasking Framework as a development library and to the applications developed using it. Our argumentation relays on few facts which we emphasize in the following points:

In data-flow programming the application is modelled as a directed graph, where data is processed by tasks then forwarded to subsequent tasks in a pipe-lined manner. The instructions do not have to wait for the preceding tasks to finish but can be executed as soon as the data are available, i.e. event-driven. The Tasking Framework API provides abstract classes to design the applications as directed graph of tasks and channels. The loops in the API that connect the tasks and channels are therefore bounded. In other words, the API has no dependencies on any data provided at run-time. The following loop for example in the function TaskInputArray::ConnectTask is bounded at compile time and its *length* represents the number of input a task has. In Figure3.5 Task2 has 4 inputs, thus, $length = 4$.

$$for\ (unsigned\ int\ i = 0u;\ i < impl.length;\ ++i)$$

In Tasking Framework the channels represent data containers. The mechanism provided by Tasking Framework to exchange data between tasks and inputs is deterministic and does not rely on the data's type or value. For example, the loop in Listing 3.1 is bounded by the number of inputs associated to the considered channel.

Listing 3.1: Push Function

```
1 void Tasking::Channel::push(void)
2 {
3   for(InputImpl* i = m_inputs; i != NULL; i = i->channelNextInput)
4    {
5         i->notifyInput();
6    }
7 }
```

The tasks in the data-flow application are, in general, stateless bare executors of algorithms. The developer should override the virtual method *execute(void)* to implement the proper algorithm for each task. It is worth clarifying the fact that tasks are generally activated on the arrival of input data. However, it is not always the case that their CFG depends on the input data type or value. Therefore, the code in Task::execute() is in general statically analyzable as long as the developer adheres to implement stateless tasks whose CFG is independent of the input data's type or value.

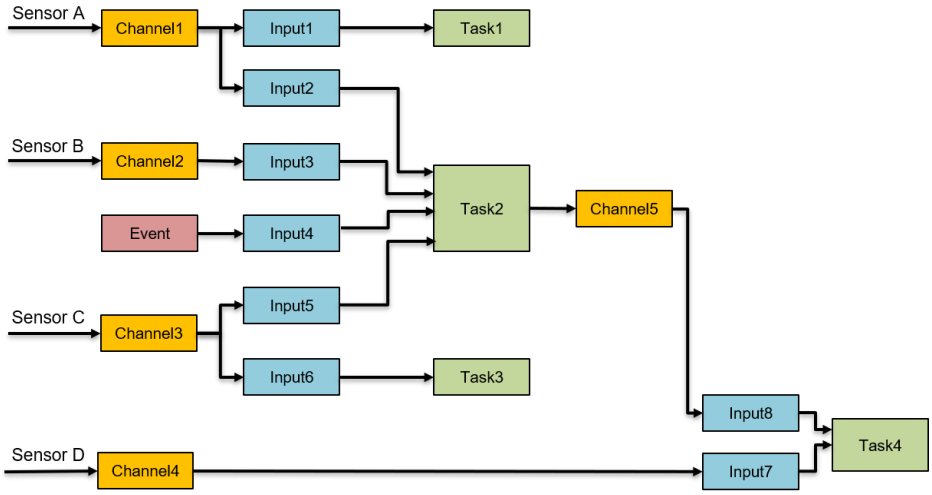


Figure 3.5: BIRD - AOCS as realized in Tasking Framework

3.7 Tasking Framework in This Thesis

In order to use Tasking Framework in hard real-time safety-critical applications, verification and validations must be carried on to ensure its real-time capabilities. Analyzing the system's real-time capabilities and proving its abilities in meeting its deadlines can only be done using static analysis. Static analysis computes abstract over-approximations for all sets of architectural states at all points of execution. It provides a safety constrain that a certain condition will never occur at a certain execution point. This safety property allows proving a safe WCET upper bound. Providing WCET is the first step towards safety-critical applications being developed by the Framework. In this thesis, Our main goal is to define a framework that can perform automatic WCET static analysis for C++ applications as Tasking Framework [44].

Two main reasons make Tasking-Framework a good candidate test-case for our thesis.

- **Real-time Guarantees**

  Tasking Framework is dedicated to develop data handling applications. It introduces a new scheduling model for satellite on-board data modeling and computation. The Framework does not follow the standard conservative scheduling where all computation processes have to wait till the arrival of one message. However, it introduces ASAP scheduling that makes better use of the available timing and improves the whole on-board system's worst-case response time. In this context, performing WCET analysis becomes essential towards computing end-to-end real-time guarantees on the proposed scheduling model for satellite on-board systems.

- **C++ Real world Application**

  Tasking Framework is not a research platform that includes only subsets of C++. It is a real-world application through which we can demonstrate whether it is possible to perform WCET for model-based C++ code. The Framework includes most of the C++ structures which are used in embedded safety-critical applications. It includes classes, Abstract classes, virtual methods. This makes it a strong candidate for performing WCET analysis.

3.8 Use-Cases from Tasking Framework

Through this thesis, we selected four use-cases from Tasking Framework to analyze. The use-cases were carefully designed to cover all the functionalities of the Tasking Framework.

3.8.1 First Use-Case

The first use-case in figure 3.6 is just a Hello World example that contains the basic components of the framework. The case is composed of *handleTask* which is activated when new data is pushed to *inChannel*. Next *handleTask* reads the values stored in *inChannel* and *outChannel*, saves the output to *outChannel* and then sets the timer, *modifyTrigger*, to 500 ms. After activation by the timer, *modifyTask* reads the value stored in *outChannel* and writes the result to *inChannel*. Then, it pushes inChannel, i.e., activating *handleTask*.

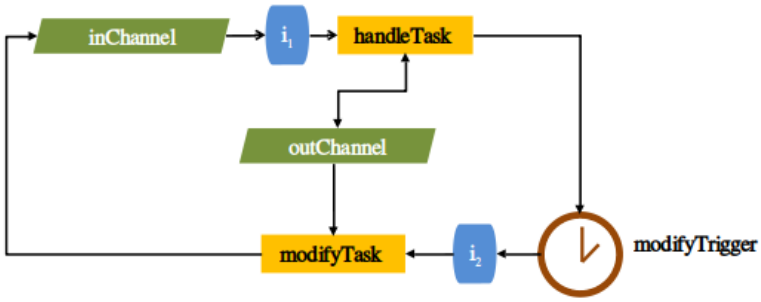


Figure 3.6: Use-Case Number One

3.8.2 Second Use-Case

The second use-case in figure 3.7 is an actual real application of the Tasking Framework from the ATON project [78]. It is an application of a data handling model in the navigation of a sub-system. The system navigates through two cameras which are used to detect craters on the Moon's surface and then control the motor based on the craters' positions. The case consists of seven tasks, five channels and two timers. *camTask1* and *camTask2* are activated periodically every 1000 ms by *inputTrigger* while, *navTask0* is activated periodically every 100 ms.

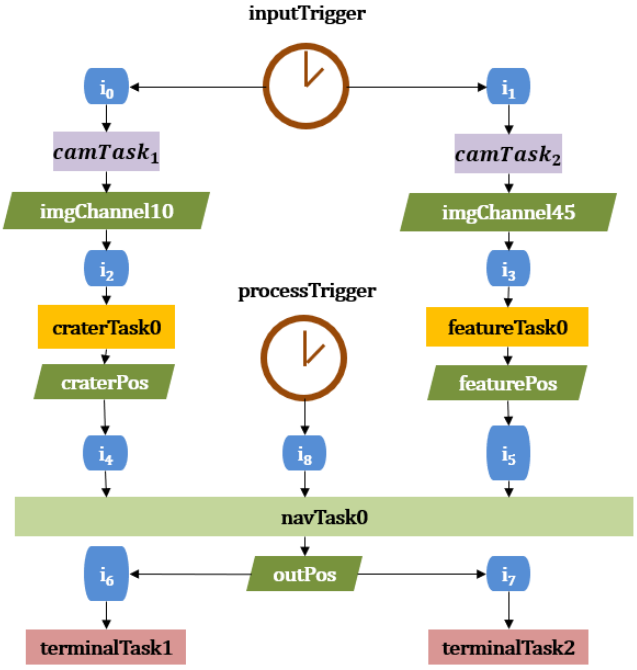


Figure 3.7: Use-Case Number Two

3.8.3 Third Use-Case

The third use-case in figure 3.8 is designed to cover the synchronization of the tasks in the Tasking Framework. The example consists of five tasks synchronized by barrier activated by the same trigger and another task starts later after the barrier. The five tasks are activated periodically every 1000 ms. When all the tasks finish execution, the barrier activates the *example task*.

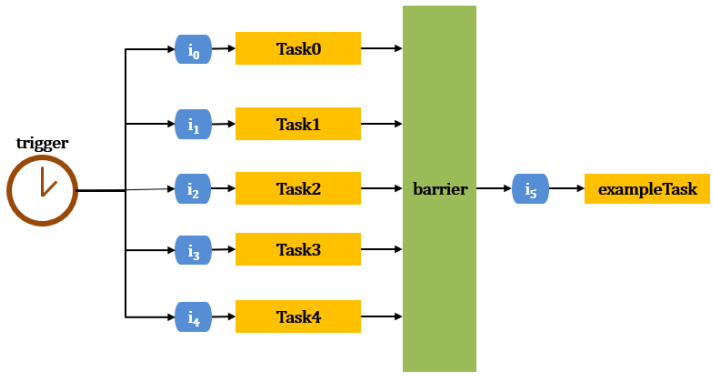


Figure 3.8: Use-Case Number Three

3.8.4 Fourth Use-Case

The fourth example in figure 3.9 is a direct example for asynchronous execution in the Tasking Framework. The example consists of five tasks that are triggered by different timers. The tasks are even triggered with different periods ranging from 250 ms to 1500 ms.

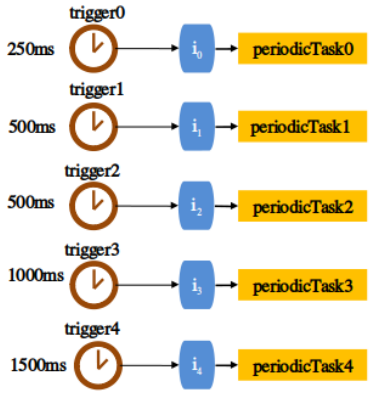


Figure 3.9: Use-Case Number Four

Chapter 4

WCET analysis – Opportunities and Challenge

4.1 Motivation

This chapter will introduce the available opportunities and challenges to perform static WCET analysis on C++ code. Our main goal through this chapter is to put the literature review into action through:

- Testing the potentials of the available methodologies to get reliable results for C++
- Showing the effect of tweaking the input types
- Testing the effect of integrating tools to get a reliable static analyzer

Our target here is to answer the following questions:

- Is there an already defined methodology that is capable of analyzing C++ applications?
- If not, does integrating small changes to the current methodologies can lead to any results? For example, the modifications introduced in [82] to allow SWEET [62] to perform static WCET analysis on LLVM IR.
- If the above points are not applicable, what should be the next starting point?

4.1.1 Motivation Example

Defining loop bounds is an essential step to get accurate WCET. Automating this step is challenging especially for C++ code. As an example, listing 4.1 shows a loop from **Clock::dequeueAll()** function from Tasking Framework. As shown, the iteration bounds for the loop depend heavily on the size of data structures in the main memory. The loop iterates over a double-linked list in the memory and resets every list element. In order to bound such a loop, information about double-linked list sizes must be exploited. Our motivation here is to explore whether any of the previously mentioned methodologies can analyze such loop and detect the number of iterations.

Listing 4.1: dequeueAll Loop from Tasking Framework

```
1   timeQueueMutex.enter();
2   // Reset pointers of all events in queue
3   EventImpl* event = queueHead;
4   while (event != NULL)
5   {
6       EventImpl* next = event->next;
7       event->queued = false;
8       event->next = NULL;
9       event->previous = NULL;
10      event = next;
11  }
```

4.2 Using State of the Art Techniques and Tools

To perform a complete automated analysis, we experimented practically the capabilities of the previously mentioned tools. We divided the analysis into three categories based on the analysis input code type: source code level analysis, intermediate representation level analysis and binary-level analysis.

4.2.1 Source Code Level

To the best of our knowledge, no tool can perform complete static analysis for C++ code. However, there are tools like oRange [18] and PAGAI [47] that perform the loop bound analysis on the source code. During testing these tools on our Tasking Framework, we faced the following challenges:

- oRange [18] performs analysis only for C Code.
- Although PAGAI [47] performs the analysis on C and C++ code, the analysis runs only on programs structured in only one file.

To assess the usability of such tools, we compiled the code of the four Tasking Framework use-cases and lifted them back to C code using RetDec [9], which is a retargetable machine-code decompiler based on LLVM [6]. It was chosen among various decompilers because it detects and reconstructs C++ class hierarchies beside its capabilities in reconstructing function, call graphs, control-flow graphs, types and high-level constructs.

Results

- oRange is not able to calculate loop bounds correctly. It reports **NOCOMP** for all loops in the four use-cases from the Tasking Framework.
- PAGAI did not return useful information regarding loop bounds. For example, for lifted loop in **Tasking::Clock::dequeueAll()** shown in figure 4.2, PAGAI returned ambiguous bounding condition for all registers in the loop body. The tool result states that all register values should be larger than zero to assert that the loop path is reachable.

- Most of Tasking Framework loops depend heavily on memory accesses. Lifting the binaries back to C while maintaining a correct memory model is challenging to achieve. As in figure 4.2, the loop in **Clock::dequeueAll()** was translated to memory pointers initialized with zeros. This translation makes it hard for tools like oRange or PAGAI to bound the loops correctly.

Listing 4.2: C-Lifted dequeueAll Loop

```
1   while ((int32_t)(v13 || v12) != 0) {
2       // 0xa54c
3       v10 = (int32_t)v13 >> 31;
4       v11 = v10 + 48;
5       *(char *)(v10 + 10) = 0;
6       *(int32_t *)v11 = 0;
7       *(int32_t *)(v10 + 52) = 0;
8       *(int32_t *)(v10 + 56) = 0;
9       *(int32_t *)(v10 + 60) = 0;
10      v12 = *(int64_t *)v11 >> 32;
11      v13 = v12 / 0x100000000;
12  }
```

4.2.2 Intermediate Representation Level

Performing the analysis on the intermediate representation was applicable using SWEET [49]. The framework performs complete automated analysis, including loop bound analysis. The recently developed LLVM IR to ALF (SWEET intermediate language) translator [82] offered a generic solution for our case. Figure 4.1 shows the required steps to perform the analysis using SWEET. Using the below bash commands in Listing 4.3, C++ code is translated to ALF IR, which is ready to be analyzed by SWEET.

Listing 4.3: SWEET to ALF

```
$ clang++ -Wall -emit-llvm -S -o - test.cpp |
opt -mem2reg -instcombine -instsimplify -instnamer |
llvm-dis -o test.ll
$ llc -march=alf -alf-standalone -o test.alf test.ll
```

Results

- Although SWEET performs automated loop bound analysis, it fails to bound the loops in Tasking Framework use-cases. The output flow facts file from SWEET was empty for all test-cases. In order for the analysis to proceed, flow-facts were given to SWEET.
- SWEET failed to build its abstract execution model due to the LLVM/ALF translator's failure in building correct memory model for the translated code.
- SWEET failed to find the worst-case execution path for all the use-cases.

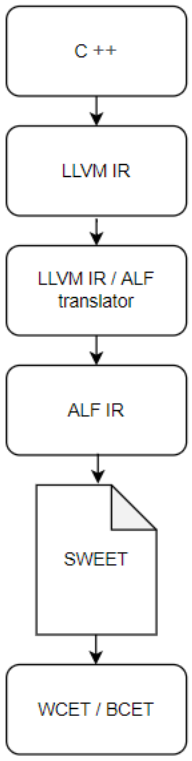


Figure 4.1: Analysis using SWEET

4.2.3 Binary Level

We move forward to perform WCET analysis on binary-level using OTAWA framework [16]. The framework was used as an eclipse plugin. The use-cases were built normally for armV7m using **arm-none-eabi** toolchain with the same hardware description file given by the tool. In order to automate the analysis, we integrate POLYMAL [15] as a flow-facts generator to OTAWA.

Results

- OTAWA failed to reconstruct the complete CFG. Although the CFG of most functions was retrieved, the tool failed to restore the CFG of functions with indirect jump instructions.
- POLYMAL failed to report any loop bounds. As the tool is based on the same infrastructure analysis of OTAWA, it reports a failure in reconstructing the CFG for the same functions reported by OTAWA due to indirect jump instructions.
- OTAWA was able to return partial results. The tool could get WCET for functions that do not contain loops or indirect jumps.

- OTAWA offers the possibility to annotate loops and resolve indirect jumps manually. However, this is not an option for our applications because real applications developed by the tasking frameworks contain hundreds of functions and thousands of function calls. Resolving each function call manually, which may recursively contain hundreds of function calls, is time-consuming and introduces human error into the calculations. The same is true for bounding loops for these applications.

4.3 Summary

In this chapter, we tested all the possible approaches with the available tools to compute the WCET. We found that no tool can capture loop bounds for industrial code as Tasking Framework. They were all designed for a simple code structure that does not fit our case. The only positive point was during testing OTAWA. It showed capabilities in calculating WCET for complicated C++ code. However, we are still unable to restore the complete CFG due to virtual methods. In the next chapter, we propose a tool that can be integrated with OTAWA to solve these two problems.

Chapter 5

Symbolic Loop Bound Analysis

As shown in the previous chapters, there is no suitable tool to perform loop-bound analysis for C++ code. This chapter introduces DELOOP, our new concept for bounding loops and resolving indirect jumps using symbolic execution with Z3 on the low-level IR. At the beginning of the chapter, we demonstrate some definitions then we present our idea, algorithms, and implementation.

5.1 Motivation

Before moving forward to discuss DELOOP components, we would like to reveal our motive behind choosing this approach.

- First, performing the analysis on the IR level ensures that the proposed analysis will be architecture and source code independent. It also resolves and downgrades the complexity of C++ structures.
- Symbolic execution approach is chosen as it is a powerful technique for program behavior analysis. It brings together the benefits of both dynamic and static analysis. Given correct and precise program trace, it generalizes that trace to predict the program's behavior for a given input. We used this approach to understand the conditions, under which a program path is executed to automatically determine the number of iterations of basic blocks in this path.
- Staring the analysis from the binaries skips the linking problem on the IR level arises from building tasks developed using libraries as Tasking Framework.

5.2 preliminaries

This section introduces some of the concepts and notations that will be used throughout the chapter.

- **Path**: In program flow-graph a path is defined as a finite sequence of basic-blocks B_1,B_2,...B_k such that k > 0 and for all 1 ≤ i < k there is a transition from B_i to B_{i+1} [23].
- **Symbolic execution**: It is a technique of executing the program abstractly. The execution covers multiple inputs of the program that shares the execution path through the code. The execution treats these inputs symbolically and returns expressions that depend on these symbols [12]. Symbolic execution is divided into two main categories:
  - **Static Symbolic execution**: It explores the control-flow of a sequential program P to determine its feasibility by supplying symbols that represent arbitrary values. The goal is to execute the instructions normally, except that values are symbolic expressions due to input symbols. This is described by formula Φ(P) that contains the set of inputs i ∈ I that determines the path's feasibility. The execution is split in case of conditional branching to find a set of inputs i ∈ I that satisfy each path independently. For each path, each instruction's execution is checked against the branching condition. If it is not possible, then Φ(P) is set to be empty, indicating that no program path can be followed [51].
  - **Dynamic Symbolic Execution (DSE)**: It takes the program exploration one step further through the execution of a program P with input **i** to provide the execution process with a feasible path. It then uses actual values from the execution P(i) to replace symbol expressions whenever possible. DSE provides a concrete execution behavior combined with symbolic expressions. Its major benefit is simplifying the symbolic execution by combining the concrete execution behavior with the symbolic expressions [51].

  To explain the symbolic execution concept, let us consider the example in listing 5.1 and its equivalent assembly in listing 5.2. In normal execution, the program reads the results calculated value as an input **i** (e.g., 6) and allocates it to R0. The program then continues with the multiplication and the conditional if, which will be evaluated to success.

  In symbolic execution, a symbolic value (e.g., Γ) is read and assigned to R0. After that, the multiplication instruction will be executed and assign Γ << 1 to R0. Then, at the **cmp** statement, Γ will be compared with 9. At this point, Γ can take any arbitrary value, and the symbolic execution can proceed along both branches. The path constraints and the program state are assigned to each of the paths.

  In the below example, the path constraint is Γ * 2 >= 9 for the B2 branch and Γ * 2 < 9 for the B1 branch. Both paths can be symbolically executed independently. When paths terminate, symbolic execution computes a concrete value for Γ by solving each path's accumulated constraints. Dynamic symbolic execution for this program will then use an actual value to replace symbol expressions Γ whenever possible.

  In this thesis, we apply dynamic symbolic execution through the steps of our proposed analysis.

Listing 5.1: Symbolic Execution Example in C

```
int perfromCalculations() {
//return a computation value
}
int fail() {return 0;}
int success() {return 1;}
int main () {
int x,z;
    x = perfromCalculations();
    z = x * 2 ;
    if (z < 10) {
        return fail();
    } else {
        return success();
    }
}
```

Listing 5.2: Symbolic Execution Example in armV7m assembly

```
        bl      performComputation()
        str     r0, [sp, #8]
        ldr     r0, [sp, #8]
        lsl     r0, r0, #1
        str     r0, [sp, #4]
        ldr     r0, [sp, #4]
        cmp     r0, #9
        bgt     B_2
        b       B_1
B_1:
        bl      fail()
        str     r0, [r11, #-4]
        b       exit
B_2:
        bl      sucess()
        str     r0, [r11, #-4]
        b       exit
```

- **Satisfiability Modulo Theories (SMT)**

  The Satisfiability Modulo Theories problem is to decide a formula's satisfiability concerning combinations of background (decidable) first-order theories. Examples of SMT theories are the real-numbers theory, integer theory and other data structure theories as array and bit-vector theories. SMT can be considered as a formalized methodology to constrain programming problems. SMT solvers are mainly used in the test-cases generation and model-bounding [29].

SMT is considered as an extension of the Boolean Satisfiability problem (SAT). The Boolean Satisfiability problem determines if it is possible to give values to the variables in a formula such that the formula yields true. For example, the formula in equation 5.1 is satisfiable if p is assigned to true and q is set to false, then the three parts conjunction (formula) yields true.

$$(p \vee q) \wedge (\neg p \vee \neg q) \wedge (p \vee \neg q) \tag{5.1}$$

The language of Boolean satisfiability solvers is Boolean logic ($\vee$, $\wedge$, $\neg$, $\rightarrow$, $\leftrightarrow$), while for the SMT solvers it is the first-order theories. The first-order theories are expressive formal language systems that break statements into things((e.g., variables and functions), relations(e.g., predicate:assert (x>y)), connectives(e.g., &&, ||) and quantifiers(e.g., $\forall$, $\exists$). The satisfiability problem of the equation in 5.2 is an example of first-order linear inequality theory where the formula is satisfiable if the variables a,b,c and d are assigned to 30, 27, 32 and 21 respectively.

$$(2 * a > b + c) \wedge (2 * b > c + d) \wedge (2 * c > 3d) \wedge (3 * d > a + c) \tag{5.2}$$

Generally, the SMT goal is to determine the satisfiability of the formula Γ for a theory **T**. The formula is defined as a set of signatures Σ which includes a set of function symbols and predicate symbols. The complexity of such a problem varies from polynomial to undecidable based on Γ and **T**. Examples for **T** from [19] are:

- Real Arithmetic Theory with $\Sigma = \{+, x, \leq\}$ includes all isomorphic structures to real numbers with +, x and $\leq$ functionalities.
- Array Theory with $\Sigma = \{select, store\}$ includes all the isomorphic structures to the memory read (select) and memory write (store) functionalities.

- **Array Theory**

  The arrays theory first introduced in [69] has the signature $\Sigma = \{select, store, =\}$. The function **select(a,i)** returns the value of array a at index i while the **store(a, i, e)** function returns array a, overwritten at index i with element e. The = predicate is only applied to array elements following the axioms of the array theory [20]:

  - First axiom: i = j $\implies$ select(a, i) = select(a, j)
  - Second axiom: i = j $\implies$ select(store(a, i, e), j) = e
  - Third axiom: i $\neq$ j $\implies$ select(store(a, i, e), j) = select(a, j)

  Additionally, the theory contains the axioms for reflexivity and transitivity of equality; however, this is not relevant for the topic of our thesis.

- **Bit-Vector Theory**

  A bit vector is an array data structure that stores data compactly in one vector unit and is defined with its width, indicating the vector's number of bits. The Bit-Vector theory problem determines if it is possible to give values to the bit-vector in a formula such that the formula yields true. This approach is effective in modeling bit-level operations on the hardware level. The bit-vector theory supports bit-wise operations as $\vee$, $\wedge$, $\neg$, <<, >>, etc.

An example of addition operation using bit-vectors is shown in Equation 5.4 where the addition of 200 and 100 yields 44 due to overflow. As the theory deals with an array of bits, the formula in 5.3 that holds for integers does not hold for bit-vectors as overflow may occur [21]. Bit-vector theory is used in the context of this thesis to check the satisfiability of the SSA instructions operations.

$$x - z > 0 \implies x > z \tag{5.3}$$

$$\begin{aligned} 10111110 &= 190 \\ +01101110 &= 110 \\ \hline = 00101100 &= 44 \end{aligned} \tag{5.4}$$

- **Z3 SMT Solver**

  It is an SMT solver from Microsoft Research. The solver was designed to handle software verification and software analysis problems. Z3 includes SAT solver, core theory solver to cover equalitities and functions, satellite solver to handle arithmetic and array theories and E-matching abstract solver for quantifiers [29]. In the context of this thesis, we used the python API of Z3.

- **Single Static Assignment (SSA)**

  In compiler theory, a single static assignment (SSA) is a characteristic property of the intermediate representation, which indicates each variable's assignment occurs only once and that this variable must be defined before its usage. The main benefit of using SSA is that it directly simplifies and improves compiler optimizations [13]. For example, in listing 5.3, In normal IR assignment in the group (a), it is clear that the value of x comes from the second instruction and that the first assignment is useless. In order to detect this, a reach definition analysis is essential. However, in SSA instructions group (b), it is immediate and clear that y_1 is useless.

Listing 5.3: SSA Instructions

```
y := 5                    y1 := 5
y := 10                   y2 := 10
x := y                    x1 := y2
  (a)                       (b)
```

In the case of control flow merges, where an SSA instruction is expected from multiple paths, an additional ϕ function is added, indicating that such conflicting instruction may have different values based on the control flow path. Listing 5.4 and figure 5.1 show how control-flow merging is performed in the SSA context. In the bottom basic block in figure 5.1, the **y** value can be considered as y1 or y2 depending on the execution path. To resolve this, the ϕ function was introduced to generate a new definition **y3** that may be **y1** or **y2** based on the control flow.

Listing 5.4: CFG Mergeing in the SSA Context

```
if ( condition )
  y := 10
else
  y := 42
x := y
```

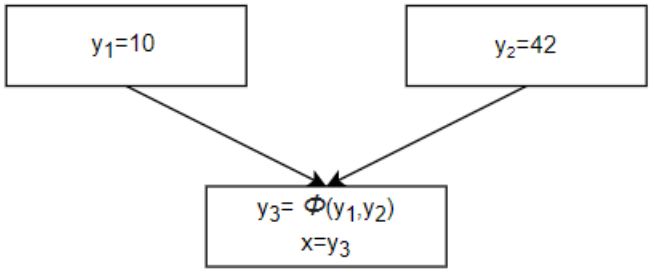


Figure 5.1: Φ Function in the SSA Context

5.3 DELOOP

To capture loop bounds, we present a new analysis based on the symbolic dynamic execution called **DELOOP** shown in Figure 5.2. The main goal of DELOOP is to determine the number of times a certain basic block **B** is executed for a given CFG **G**. DELOOP takes executable binary as program input and produces an estimation for the loop bounds as output.

5.3.1 Basic Idea

Generally, dynamic symbolic execution explores the CFG of sequential program **P** using theorem prover as Z3 [29] to determine the feasibility of a certain path Π. The exploration of the feasibility of a path Π through program **P** yields to Φ(P) that illustrates the set of inputs I to the program **P** such that for input i ∈ I the execution of **P**(i) follows the path Π. If I is an empty set, then the path is not feasible.

We modified this idea a little bit to explore loop bounds. We introduce two main concepts to be discussed in this chapter. The first approach is DELOOP with data flow analysis called **DELOOP DF**. This approach is implemented using data flow analysis supported by dynamic symbolic execution. For a feasible (reachable) path Π representing a loop path in a program **P** with set of inputs I_{init}, the approach's goal is to deduce the number of iterations needed for this set of inputs to change to I_{final}. I_{final} is defined as the set of inputs that changes the feasibility of the path Π, i.e. (exit conditions). This concept is a mixed static and dynamic approach in which the static part (data-flow) is used to define the inputs I_{init} while the dynamic symbolic execution is used to detect the required number of iterations till reaching I_{final}.

The second approach is DELOOP with full program execution **DELOOP FE** which is implemented using only dynamic symbolic execution. For a program **P** starting at an initial path Π_i with a set of initial inputs I_{in}, the approach's goal is to deduce I_{out}. I_{out} is defined as the set of outputs at the end of the path Π_i. Further, the approach defines the next path that can be reached using I_{out} as new I_{in}. Following this concept, we dynamically execute all the feasible paths in the whole CFG to resolve all the loops. This is contrary to what **DELOOP DF** does, as it only executes the loop CFG.

Although each of the two approaches has a different starting point, they follow the same pattern during the code's dynamic symbolic execution. In both cases, the symbolic execution will trace the control flow path Π taken by the execution of **P**(i). The dynamic solver will generate a path-condition from Π using a symbolic expression. It will then generate new input i that guide the search to expose the next path. In **DELOOP DF** the scope of the next path is only bounded to the CFG of the loop; this is different from **DELOOP FE** which explores all the feasible paths in the CFG.

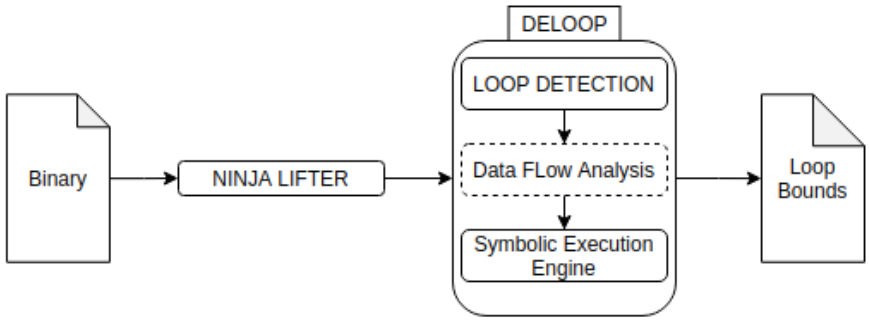


Figure 5.2: DELOOP Basic Components

5.4 Implementation

DELOOP is an analysis tool that implements a combined loop detection algorithm, data-flow analysis, and dynamic symbolic execution. The base of DELOOP is BINARYNINJA [10] which is a reverse engineering framework used mainly for binary analysis. BINARYNINJA is used to reconstruct the CFG and lift the input binaries into SSA LLIR. Its API is used to parse and facilitate all parts of the analysis. The analysis uses Microsoft Z3 [29] through their python API for the symbolic execution. The tool is implemented as shown in the following sections.

5.4.1 Loop Detection

For a given executable, the code is disassembled and lifted to a single static assignment (SSA) low-level intermediate representation. During this, the CFG $\mathbf{G}$ is reconstructed and cyclic sub-graphs $g_i \in \mathbf{G}$ are detected to capture loops in the code. The Loop detection algorithm implemented for DELOOP is a separate static analysis approach and is based on Johnson's algorithm [52]. The algorithm finds all the elementary circuits (loop) of a directed graph. A directed graph is defined as a graph $\mathbf{G\ (V,\ E)}$ consists of a nonempty set of vertices $\mathbf{V}$ and a set of ordered pairs edges $\mathbf{E}$ bounded in time by $\mathbf{O((n\ +\ e)(c\ +\ 1))}$ and in space by $\mathbf{O(n\ +\ e)}$, where $\mathbf{n}$ is the number of vertices, $\mathbf{e}$ represents the number of edges and $\mathbf{c}$ indicates the elementary circuits in the graph. A single elementary circuit is defined as a closed path where no node appears twice, except that the first and last nodes are the same. Two elementary circuits are distinct if they are not cyclic permutations of each other. Our loop detection analysis goal is to detect basic blocks grouped in a single elementary circuit (i.e., Loop). To identify an elementary circuit, the following is done:

- The algorithm in question firstly breaks down a function into basic blocks.
- Secondly, the incoming and outgoing edges of each basic block are pointed out. An incoming edge of a basic block indicates the block number of the basic block directly preceding or jumping to it. On the other hand, an outgoing edge of a basic block indicates the block number of the basic block that directly follows it.

- The next step involves utilizing information gathered about each basic block's edges to construct a cyclic graph. This cyclic graph can be viewed as a map highlighting the control flow between a group of basic blocks referred to as pathways.
- Finally, such pathways are evaluated against the criteria of whether or not the pathway connects back to a previously visited basic block on that same pathway. Meeting the proposed criteria holds the key to identifying what we define as a loop.

Each loop detected by the Loop detection algorithm is given a Loop ID number that reflects the Block ID number of the last basic block in the acquired cyclic graph of that loop. Recursive function calls or loops are not handled with the loop detection algorithm. However, they are handled automatically during the dynamic symbolic execution phase, which will be discussed later in this chapter.

5.4.2 Data-Flow Analysis

To get the initial state of registers before loop execution, a backward program slicing is performed. The goal is to find all values of registers used by the loop. The data-flow analysis is only a part of **DELOOP DF**. The data-flow analysis includes two steps:

- **Tree Generation**

  For the analysis to find the registers' initial values, the analysis comprises two parts: Inter-procedural analysis (inside function scope) and intra-procedural analysis (outside function scope). For the analysis to combine both scopes, $T \in$ G is created for every loop to define the search area. T is defined as the tree including the chain(s) of basic blocks (Loop Path) that starts from the code entry and leads to the exit basic block of the loop. The tree is formulated using the caller-callee API from **BINARYNINJA** to include the basic blocks outside the function scope. T_i and T_o are introduced as sub-trees from T where T_i refers to the path from the function entry to the last basic block of the loop. T_o refers to the tree highlighting the loop's path outside the function scope. This loop path information is extracted from the BINARYNINJA control flow analysis API, which can resolve indirect jumps partially. For example, figure 5.3 shows the construction of the loop path tree T from the CFG. For a loop starting at B4 and including B5 ,B6 and B7, The tree is generated to define the search paths for registers initial values.

- **Backward Program Slicing**

  Backward program slicing is the detection of the program points that affected (might affect) a given program point p [75]. In our analysis, backward static slicing is executed across the loop path graph T to find initial values of the register(s).

  To perform the slicing, the SSA instructions of T are modeled as nodes **V** with Edges **E**. Each node $\mathbf{v} \in \mathbf{V}$ has an input edges **ein** and an output edge **eout**. Thus, given T **(V,E)** and a register R_i in an instruction inside the loop as a current starting node, the slicing returns a list of instructions that traces back to the initial value of the register. This is done by iterating over the previous nodes until finding a node whose output edge $eout_{prev}$ is equal to the input edge of the current node $ein_{current}$. In the case of variable input edges, i.e., registers, a deep first search is executed to find the source(s) of the previous node until reaching the stopping criteria. If the input of this previous node ein_{prev} is constant (constant or load from memory), then the search stops as the slicing criteria is met.

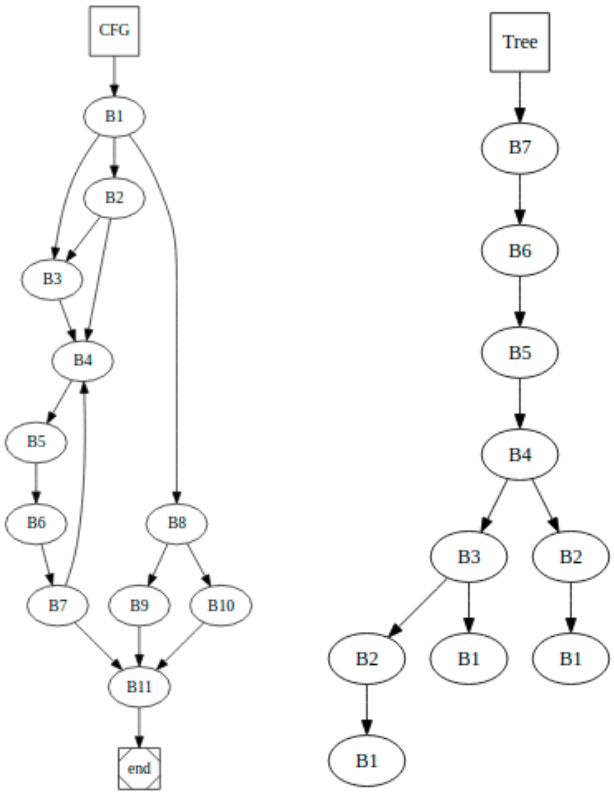


Figure 5.3: Tree Generation From The CFG

The presence of the instructions in the SSA form is essential for the program slicing. It facilitates the slicing and thus finding the initial value of the registers. The SSA form helps limiting the number of the input edges when modeling the nodes of the G_{LP} graph. In SSA form, the nodes are modeled with two input edges and one output edge, limiting the graph search space. The use of non-SSA instruction will increase the algorithm's complexity by having multiple input edges and thus increasing the complexity of the slicing and the search algorithm.

5.4.3 Dynamic Symbolic Execution

- **Memory Model**

  Two memory models are built based on the array theory. The models allow the dynamic execution of SSA memory-based instructions using SMT formulas. Data inside the arrays are formulated as bit-vectors, and thus the arrays are defined as arrays of bit-vectors. The first memory is used for the symbolic execution of the load/store instructions (model the main memory) while the other is used for the push/pop instructions (model the stack). The main memory model is initialized with the initial values of all the program's data variables in the given input-file.

- **Translation to SMT Formula**

  The first step in the symbolic dynamic execution is to take the program under analysis and compile it into SMT formulas. Later, we use these formulas to investigate the execution of the program. This step is the core heart of our analysis. It transforms IR operands into Z3 expressions to be used later by the z3 solver during the code's dynamic computation. First, every operand is expressed as a bit vector with a size that matches the target architecture. This operand is then passed to the Z3 solver in a form that implies the IR instruction's mathematical effect on the solution state. The SSA form facilitates the whole translation process as every SSA instruction is directly mapped to one SMT formula using array and bit-vector theories. For example, the LLIR SSA instruction R2 = R3 + 1 is translated as shown in equation 5.5 where a bit-vector variables are defined for the R2, R3 and the immediate value. Memory instructions are also interpreted in the same way. For example, the LLIR SSA instruction shown in 5.6 is computed as **select(mem,0x808)** where mem is the memory model and 0x808 is the load address. The translator performs the previous steps for all kinds of IR operations.

$$R2 = R3 + 1 \implies BitVec(R2, size) = BitVec(R3, size) + BitVec(1, size) \tag{5.5}$$

$$R2 = [data\_0x8080] \implies select(mem, 0x808) \tag{5.6}$$

- **Symbolic Execution Engine**

  A dynamic execution engine is built using Z3. Its goal is to execute the SMT formulas dynamically on the memory model. The engine has n number of states each of them captures the change in registers ($\Delta\Phi$), memory ($\Delta\Gamma$), or stack ($\Delta\beta$) status for the execution of a single formula. The total number of n is equal to the number of SSA instructions of the program under analysis. The concept of the states transformed our execution from a static to a dynamic symbolic execution. For example, during the translation of R2 = R3 +1, the translator first checks whether there are previous variables in the engine states for R3 and R2. In the case of already existed variables, the value of R3 is fetched from the engine and increased by one and then assigned to R2. If R3 has a previous value of 100, then the translation process is done as shown in equation 5.7. The same is correct for the memory instruction in equation 5.6. If the address 0x8080 has a value then R2 will be updated as shown in equation 5.8.

$$R2 = R3 + 1 \implies BitVec(R2, size) = BitVec(100, size) + BitVec(1, size) \tag{5.7}$$

$$R2 = [data\_0x8080] \implies 0xa080 \tag{5.8}$$

- **Initialization**

  For every loop l_i, initial state S_i is defined, which is a snap of the memory and registers before the loop's execution. This initial state is the equivalence of I_{init} and I_{in} in our symbolic execution model. In the case of **DELOOP DF**, the data-flow analysis is responsible for tracing back the registers' initial values ϕ_i. To avoid memory aliasing and to ensure that Γ_i is set correctly, we used the **-ffunction-sections** flag option during the building process. This option generates a separate ELF section for each function in the executable file. The output section for each function has the same name as the function that generates the section. In the case of **DELOOP FE**, the execution starts from the program entry point till the loop head to get the registers' ϕ_i and memory Γ_i initial values. The initial state S_i $<\phi_i,\Gamma_i>$ is then passed to the symbolic dynamic execution solver. All instructions are then executed until the final state S_f is reached. S_f is defined as the state when the loop exit condition is satisfied. This final state is the equivalence of I_{final} and I_{out} in our symbolic execution model. In other words, the SMT solver is given the inputs (loop initial conditions) and the outputs (loop exit condition) and is asked about the number of the needed transition phases to reach from the input to the output.

- **Execution**

  The SMT formulas are executed dynamically through the execution engine and the memory models. The execution engine checks the satisfiability of every formula and then updates the engine status with the effect of execution. For example, assuming a previous value of R3 from a previous engine state to be 100. The SMT formula's execution in 5.9 yields true and updates the value of R2 in the execution engine to 101. The same concept is true for memory-related SMT formulas. The execution of formula $\boldsymbol{i}$ changes the engine state from s_i to s_{i+1} which belongs to the loop overall state S_j. The resulting state from execution of a formula is $s_{i+1} = s_i + \Delta_k$ where $k \in \{\phi, \Gamma, \beta\}$ and depends on the formula type. For example, a memory formula implies Δ_ϕ while a PUSH formula implies Δ_β. After executing the loop instructions, the loop exit condition's satisfiability is checked to determine whether the loop should be re-executed or the maximum number of iterations is reached, i.e $S_j = S_f$.

$$BitVec(R2, size) = BitVec(R3, size) + BitVec(1, size) \tag{5.9}$$

  During execution, formulas are categorized into four main types, memory-related formulas, registers-related formulas, stack-related formulas and finally director formulas. Director formulas (branching instructions) are the ones responsible for setting the execution path for the solver. For example, the SMT formula for the LLIR instruction in listing 5.5 evaluates the branching condition r1 $>$ 0 and then sets the next basic block to be executed based on the evaluation result.

Listing 5.5: Branch Instruction (Director Formula)

```
If r1>0 then BB1 else BB2
```

Algorithm 1 shows how the symbolic dynamic execution is performed for SMT formulas. For every instruction I in a basic blocks B of the loop cyclic graph g_{loop} the satisfiability of its formula is checked, and the status s_i is updated based on its effect on the engine model. The loop instructions are executed by updating the engine state and evaluating the transition condition. If the instruction is a branch instruction that may yield to basic blocks B_x or B_y, the algorithm checks whether this basic block belongs to g_{loop} or not. This check is considered as an exit condition for the algorithm. The exit condition is only valid for **DELOOP DF**. For **DELOOP FE** the execution normally runs to the next basic block outside the loop till the exit function of the program. Through this, **DELOOP FE** explores all the loops in the program in a sequential way following the same execution technique mentioned in the algorithm. To detect the bounds, we investigate all basic blocks inside a loop to get the maximum loop iterations. The loop bound is $\max(n_i)$, where n is the basic block execution number, and i is the basic block index. The flow graph shown in Figure 5.4 is a simple example to explain the concept. Given the registers' and memory initial values, S_{init} is then set as the engine state before executing the loop. The first instruction R1=M[R0] is executed updating the engine state with R1 new value equal to five. At this step the satisfiability of the loop exit condition (R4<10) is checked resulting in jumping back to point a. The process is then repeated until satisfying the exit condition where the value of R4 becomes larger than 10. For the example purpose, we assume that every instruction execution leads to the loop repetition. From this, the analysis deduces that the loop is required to be executed three times to reach the point c.

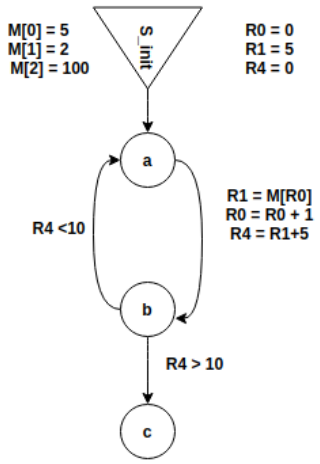


Figure 5.4: Loop Execution with Z3 Engine

Algorithm 1 Z3 Execution Engine

Input : g_{loop} : loop cyclic graph
Output : N : Number of loop iterations i
Initialize $S_i < \Phi_i\ ,\ \Gamma_i\ ,\ \beta_i >$
for B_i in g_{loop} **do**
  for I_i in B_i **do**
    if I_i is RegisterSet **then**
      $s_i \implies s_i + \Delta\ \Phi_i$
    if I_i is MemoryWrite **then**
      $s_i \implies s_i + \Delta\ \Gamma_i$
    if I_i is Push || Pop **then**
      $s_i \implies s_i + \Delta\ \beta_i$
    if I_i is conditional branch: B_x, B_y **then**
      $B_{next} \implies B_x \mid\mid B_y$
      if $B_{next} \notin g_{loop}$ **then**
        exit

5.4.4 Resolving Indirect Jumps

The state functionality of the execution engine kills two birds with one stone. It does not only bound loops, but it also resolves indirect jumps. During the dynamic execution of **DELOOP FE**, each formula's satisfiability is checked, and then the engine status is updated as demonstrated in algorithm 1. Updating the status means that the SMT solver yields SAT and yields the formula operation's actual value. Thus, before the execution of any indirect call, the engine's state contains the register's correct value. For example, during the execution of LLIR branch instruction call R3 in listing 5.6, the value of R3 is fetched from the engine where its replacement in the call instruction yields to the actual jump address. Resolving indirect calls is only available in **DELOOP FE**.

Listing 5.6: Resolving Indirect Calls

```
call R3 ⟹ call 0x808a
```

5.4.5 Proof of Correctness

DELOOP FE drives an automatic proof of correctness. The success of the analysis (loop bounding and resolving indirect jumps) must and has to lead the execution path to the CFG's exit function. Any failure during the analysis will lead to the unsatisfiability of the memory formulas during execution and thus the analysis failure. The memory formula's unsatisfiability may result from out-of-bound storing (store outside the memory model) or incorrect selecting (select uninitialized value/select from undefined address). Thus, reaching the exit of the CFG not only means the analysis end but it also means that the results are correct. By correctness we mean that the loops are bounded and the indirect jumps are resolved.

5.5 Primary Evaluation

We test our approach based on test-cases from the Mälardalen WCET benchmarks suit [43]. The Mälardalen benchmark was chosen as it has a common, open-source set of test programs for WCET methods and tools. The benchmark includes a broad set of program constructs for evaluating and testing WCET algorithms. For example, [18], [47], and [62] used the Mälardalen benchmark for performance evaluation. Although the Mälardalen is not as challenging as Tasking Framework use-cases because it includes trivial ANSI-C programs with mostly static loop bounds. We prefer to use it as a first evaluation step for two reasons:

- To compare our approach results with other already existed tools that only analyze ANSI-C code
- To decide on which of the proposed concepts to proceed (**DELOOP DF** or **DELOOP FE**) with Tasking Framework

The test-cases were built for **arm-none-eabi** tool chain for a single core **cortex-m3** as CPU with **armv7-m** architecture with the following option flags:

```
CCFLAG_Optmize: -o0 -ffunction-sections -fdata-sections
CCFLAG_Other:   -finline-limit=10000 -funsigned-char
                -funsigned-bitfields
                -fno-split-wide-types
                -fno-move-loop-invariants
                -fno-tree-loop-optimize
                -fno-unwind-tables -fshort-wchar
```

We assess the proposed analysis with two different: approaches **DELOOP DF** and **DELOOP FE**. In the second approach **DELOOP FE**, the test-cases were executed symbolically from the beginning till the end using only Z3. During this, the solver is responsible for detecting the initial condition S_i before evaluating the loop. The idea behind this evaluation is to explore both methods' strengths and weaknesses and to evaluate the efficacy of the data flow analysis in setting the initial conditions.

We have used 25 programs from the Mälardalen WCET Benchmark suite [43] to test our tool. We choose the most commonly used programs by other tools as comparison test-cases. The benchmarks are a diverse collection of programs designed to test different parts of the analysis. Our evaluation includes comparison with other tools as oRange [18], PAGAI [47] and SWEET [62]. Table 5.1 shows the comparison where BLT and %BLT represent the percentage and number of bounded loops out of 159 loops respectively. E and %E represent the percentage of exact bounding out of the same number of loops. For all conditions, DELOOP finds a bounding solution for the loop. The only exception is very large functions as **nischneu** where the BINARYNINJA fails to restore the CFG of the main function.

Table 5.2 shows the benchmarks results for DELOOP where #LC indicates the number of lines of codes, #L represents the number of loops in the program, E and %E represents the number and the percentage of exact bounded loops in the program. It highlights that DELOOP in the full execution mode has higher precision than the data-flow model. The solver correctly captured the loop bounds in all benchmarks As it follows a certain execution path that ensures correct initialization before loop execution. It was also able to bound recursive and nested loops as in **fac** and **ndes**. **DELOOP DF** misestimates some bounds. The reason behind its low precision is the

Table 5.1: Loop-Bounding Tools Comparison

Tool	BLT	% BLT	E	% E
DELOOP FE	158	99%	152	94%
oRange	133	84%	138	87%
PAGAI	116	73%	116	73%
SWEET LP	100	63%	81	51%
DELOOP DF	158	99%	72	47%

registers' initialization using data-flow. As it sometimes leads to multiple execution paths where registers' initial values can be found. This obliges the analysis to run the solver at different times with a different initial values to get the loop's upper bound. At this point, the loop bounds are defined as the result of the registers' initial value that leads to the highest number of iterations. This technique ensures a safer upper bound but lowers the precision of the results. The static data-flow analysis also fails if the register(s) under analysis has been updated during the execution, which means false results due to the static approach. On the contrary, the full-execution approach predicts correct values for registers before loop bounding.

Table 5.2: Benchmark Results

			DELOOP DF		DELOOP FE	
Program	***#LC***	***#L***	***E1***	***%E1***	***E2***	***%E2***
adpcm	879	27	10	37%	26	96%
bs	114	1	1	100%	1	100%
cnt	267	4	2	50%	4	100%
cover	640	3	3	100%	3	100%
crc	128	6	1	16%	6	100%
duff	86	2	0	0%	2	100%
edn	285	12	10	83%	12	100%
expint	157	3	1	33%	3	100%
fac	21	1	0	0%	1	100%
fdct	239	2	2	100%	2	100%
fft	219	30	8	27%	25	83%
fibcal	72	1	1	100%	1	100%
fir	276	2	1	50%	2	100%
inssort	92	2	2	100%	2	100%
jcomplex	64	2	0	0%	2	100%
ludcmp	147	11	8	72%	10	100%
matmult	163	7	4	57%	7	100%
ndes	231	12	0	0%	12	100%
ns	535	4	2	50%	4	100%
nsichneu	4253	1	0	0%	0	0%
prime	535	2	2	100%	2	100%
qsort-exam	121	6	3	50%	6	100%
qurt	166	3	1	33%	3	100%
select	114	4	0	0%	4	100%
ud	161	11	8	72%	11	100%

Table 5.3: Comparison between DELOOP DF and DELOOP FE

Criteria	DELOOP DF	DELOOP FE
Entry Point	Loop Header	Program Entry Point
Approach	Data-Flow + DSE	DSE
Indirect Jumps	BINARYNINJA API	Resolved accurately with SMT + DSE
Registers values before SDE	Non-deterministic	Accurate values
Memory model before SDE	Subjected to Aliasing	Accurate values

5.6 Conclusion

We presented a loop-bound tool based on dynamic symbolic execution and a mathematical solver. It proceeds by initializing the solver and then starts to run the loop code symbolically to evaluate the number of iterations. We showed two separate ways in which the analysis can be used and evaluated against other available tools. Table 5.3 shows a brief comparison between using DELOOP in a pure dynamic symbolic execution (DSE) mode and DELOOP as a hybrid methodology between data-flow and dynamic symbolic execution mode. Thanks to the full-execution technique of DELOOP, our approach can naturally resolve indirect jumps and determine the overhead resulting from the tool-chain. While the speed of our method is currently slower than other existing techniques, we believe that this can be improved in the future.

Chapter 6

Evaluation

In the previous chapters, we explored the challenges and the tools that may be useful to analyze a real industrial application as the Tasking Framework. In this chapter, we introduce our approach for calculating the WCET for a C++ application developed by the Tasking Framework using DELOOP and OTAWA.

6.1 Introduction

We showed in chapter 4 that OTAWA could compute the WCET for some functions in the Tasking Framework. Therefore, we decided to adopt OTAWA as a static analyzer. However, OTAWA needs loop bounds and it cannot resolve indirect jumps. For this end, we proposed a dynamic symbolic execution-based analysis than can compute the loop bounds and resolve indirect jumps using z3 solver.

The Tasking Framework is a static event-driven software development library used to develop data-flow-based applications. It generates a map of channels and tasks representing the developed application. The Tasking Framework's behavior is *deterministic* and its structure does not depend on values given at run-time. Loops in the Tasking Framework can be categorized into two types. Firstly, loops in the architecture of the framework depend only on the designed system's structure. Thus, they are known during the compile-time and will never change during the execution of the application. Secondly, loops exist in the developed tasks. These loops depend on the task being executed. In this thesis, we focus only on the binary input task-triggered events. In other words, the input events can be considered as on/off switches for the tasks. Through DELOOP, we can force the occurrence of events (inputs) and analyze even the executed tasks that have fixed execution paths.

We showed on the Mälardalen WCET Benchmark suite that running **DELOOP FE** overperforms **DELOOP DF** in terms of precision when bounding loops. Considering the comparison in table 5.3 in the previous chapter, we found that the full-execution approach fits our application. To resolve indirect jumps, the data-flow approach depends on BINARYNINJA APIs. The APIs uses a static analysis which fails to resolve the indirect jumps completely. DELOOP results are based mainly on the correctness of the initial state before loop execution. This correctness can not be ensured with the data-flow analysis because registers may have more than one initial value depending on the slicing results. Memory initial values are also subjected to aliasing. Memory aliasing means that a previous operation changed the memory content at a certain address and thus reusing the old-value is a wrong assumption. For the above reasons, we decided to adopt **DELOOP FE** to compute the flow facts and resolve the indirect jumps needed by OTAWA.

6.2 Experimental Setup

Figure 6.1 shows our proposed setup for measuring the WCET. The analysis is based primarily on OTAWA as a static analyzer and DELOOP as a flow facts generator. This setup expands the capabilities of OTAWA in estimating WCET for C++ code.

Given OTAWA [16] hardware description file for armv-7m, the WCET estimation starts with reconstructing the CFG. The results of the loop analysis performed by DELOOP are then passed to OTAWA for the WCET analysis. Analysis information from OTAWA is then collected in an execution graph which then gathers timing data for every basic block. The analysis is performed for a Bare-metal implementation with no caches. Caches are not part of our analysis as we are analyzing C-level safety-critical applications. Due to their unpredicted behavior as per ISO 17770 [70], caches are turned off. The setup gives the execution-time of every function in the binary input file.

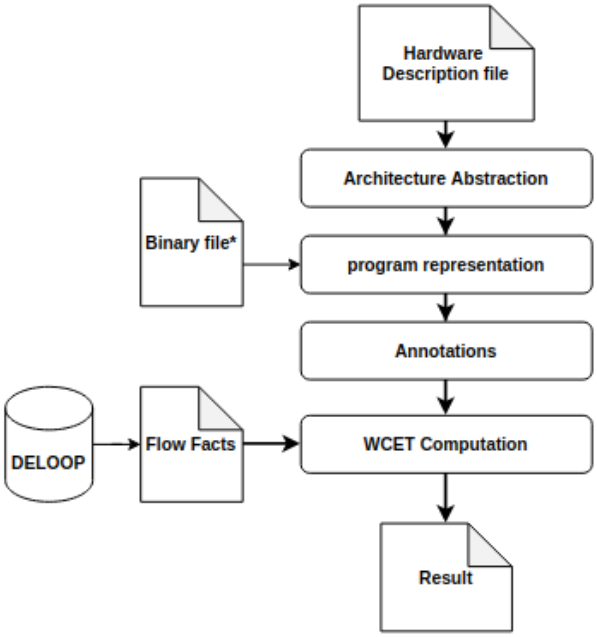


Figure 6.1: DELOOP with OTAWA

6.3 Loops Results

This section presents the results of DELOOP as tested on selected loops from the Tasking Framework use-cases mentioned in chapter 3. The loops in question are categorized as follows: Task related loops, Input related loops and Channel related loops. In the following subsections each loop category will be analysed, firstly, by explaining the functionality of the function to which the loops belong to. Secondly, we highlight the source code of the loops followed by its low level intermediate representation. Finally, we tabulate the results of DELOOP.

6.3.1 Inputs Related Loops

- **Input Array Provider**

  Input array provider is used to initialize a static memory for each input. Although there are no loops to bound from the source code shown in listing 6.1, DELOOP discovered a loop that iterates to initialize static memory for each task's input. Figure 6.2 shows the loop in the input array provider on the IR level. BINARYNINJA's GUI shows a loop through the blue arrow; however, its API does not have any loop analysis functionality and the loop detection is only possible using the loop detection algorithm of DELOOP.

Listing 6.1: Input Array Provider Constructor

```
1 template<size_t n> InputArrayProvider<n>::InputArrayProvider(void):
2 InputArray(inputMemory, n) {
3 }
```

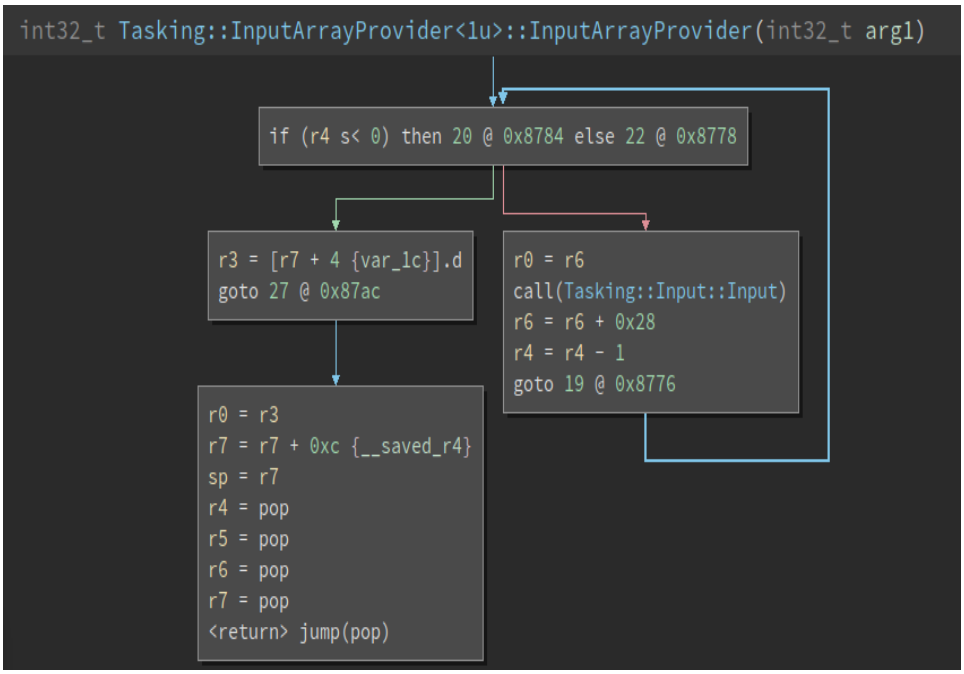


Figure 6.2: Loop in Input Array Provider

Table 6.1: Snippet of DELOOP Results For Input Array Provider

ID	Address	Function Name	Bound	Expected
Use-Case #1				
22	0x8754	18InputArrayProviderILj1EEC2Ev	1	1
23	0x84ae	18InputArrayProviderILj1EED2Ev	1	1
Use-Case #2				
22	0x8ad0	18InputArrayProviderILj1EEC2Ev	1	1
23	0x87e2	18InputArrayProviderILj3EED2Ev	3	3
Use-Case #3				
22	0x86b4	18InputArrayProviderILj1EEC2Ev	1	1
Use-Case #4				
32	0xc0dc	18InputArrayProviderILj1EEC2Ev	1	1

The table 6.1 shows a snippet of DELOOP results for the input array provider loop in the four use-cases. ID in the table refers to the loop ID, which is the ID of the last basic block in the loop body. ID numbers in BINARYNINJA restart from zero for every function, so we add every function's address to distinguish between basic blocks of the same IDs but in different functions. DELOOP reported 20 loops for the input provider functionality in the four use-cases covering all input array provider loops. Since most of the four use-cases tasks are only connected to one-input, DELOOP reported a bound of one indicating the initialization of the static memory for only one input. Except for the task **ILj3EED2Ev** in the second use-case, which relates to **navTask** on the source code level, DELOOP reported three iterations, which is the same number of inputs to be initialized for this task as shown in figure 3.7.

Detecting and bounding loops for the case mentioned in figure 6.2 is essential for WCET estimation. Consider the case when the developer goes directly to OTAWA to estimate the WCET for the Tasking::InputArrayProvider function. OTAWA will report unbounded WCET due to unbounded loop as shown in figure 6.3. Returning to the source code, the developer will find no loops. DELOOP facilitates catching such kind of behavior where the loops are hidden behind the constructor class.

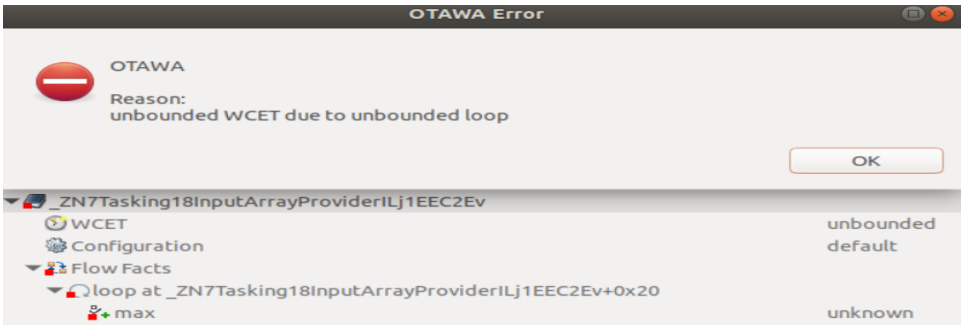


Figure 6.3: Unbounded Input Array Provider Loop in OTAWA

- **Input Array Connect Task and Reset**

  The loops in InputArray::conectTask in listing 6.2 iterates over the inputs to connect the input array to the task. The loop in InputArray::reset function in listing 6.3 iterates over the inputs to reset the input Array. Both loops depend mainly on the number of inputs of every task. DELOOP detects the loops on the IR level and reports correct bounds as shown in table 6.2. Same as in table 6.1, ID in the table 6.2 refers to the loop ID, which is the ID of the last basic block in the loop body and address refers to the function starting address. The table shows the bounds dependency on the number of tasks and the correct estimations from DELOOP. Since most of the four use-cases tasks are only connected to one-input, DELOOP reported a bound of one indicating the connection of one input array to the task or resetting the values of one input array. Except for the task **ILj3EED2Ev** in the second use-case, which relates to **navTask** on the source code level, DELOOP reported three iterations, which is the same number of inputs to be initialized for this task as shown in 3.7. The loops structure on the IR level are shown in figure **??**.

Listing 6.2: InputArray::connectTask

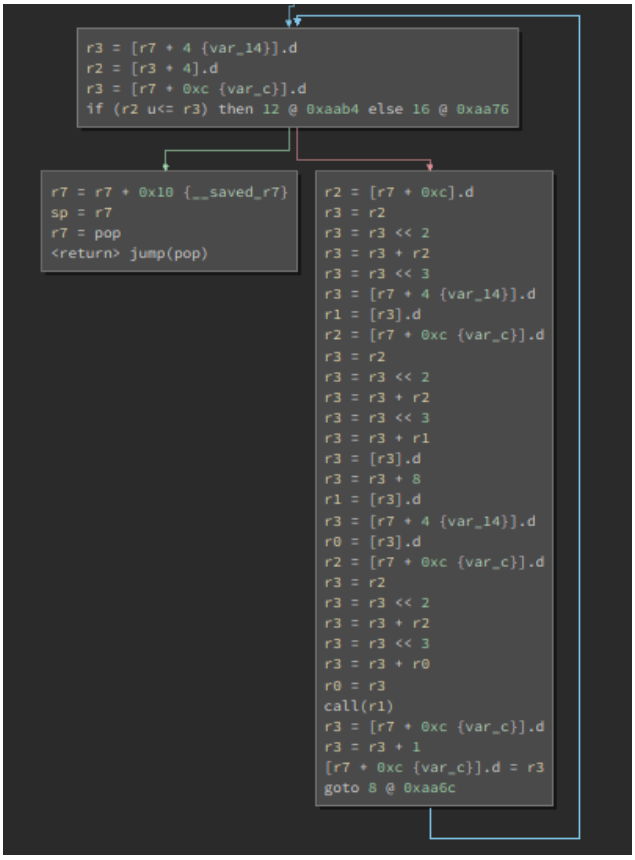
```
1 /* In function Tasking::InputArray :connectTask ( TaskImpl & task )
2 The loop iterates over all defined inputs
3 to connect the array of inputs to the designated task.
4 The number of inputs is defined in compile time
5 */
6  for (unsigned int i = 0u; i < impl.length; ++i) {
7         impl.inputs[i].connectTask(task);
8 }
```

Listing 6.3: Loop in InputArray::reset

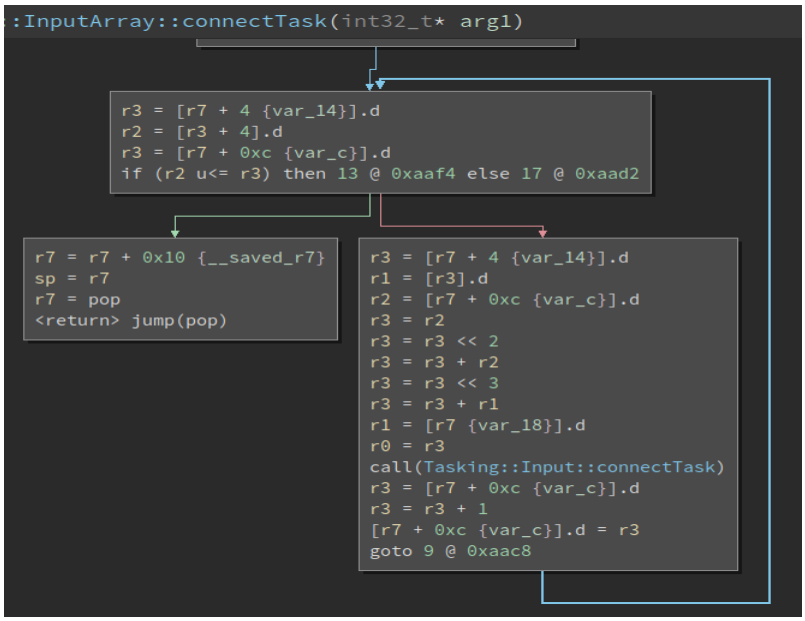
```
1 //In function Tasking::InputArray::reset(void)
2 //The loop iterates over inputs to reset them
3 for (unsigned int i = 0; i < impl.length; ++i) {
4         impl.inputs[i].reset();
5 }
```

Table 6.2: Snippet of DELOOP Results For the loops in Connect and Reset functions

ID	Address	Function Name	Bounds	Expected
Use-Case #1				
17	0xaf22	10InputArray11connectTaskERNS8TaskImplEEC2EV	1	1
17	0x820a	10InputArray11connectTaskERNS8TaskImplEED2EV	1	1
16	0xaa60	10InputArray5resetEvTaskERNS8TaskImplEED2EV	1	1
Use-Case #2				
17	0xaae6	10InputArray11connectTaskERNS8TaskImplEEC2EV	1	1
17	0x9a2c	10InputArray11connectTaskERNS8TaskImplEED2EV	3	3
16	0x82ae	10InputArray5resetEvTaskERNS8TaskImplEED2EV	3	3



(a) Loop in InputArray::reset



(b) Loop in InputArray::connectTask

Figure 6.4: reset and connectTask functions in InputArray

6.3.2 Tasks Related Loops

The loop in Scheduler::restart function in listing 6.4 iterates over tasks in the application to reset them at the end of the execution. DELOOP detected the loop on the IR level and reported correct bounds as shown in table 6.3. Same as in table 6.1, ID in the table 6.3 refers to the loop ID, which is the ID of the last basic block in the loop body and address refers to the function starting address. The table shows the bounds dependency on the number of tasks and the correct estimations from DELOOP. Since the use-cases from one to four have two, seven, six and five tasks, respectively, the detected loop bounds were exactly the same numbers.

Listing 6.4: Loop in Scheduler::restart

```
1 /* In function Scheduler::restart
2 The loop iterates over tasks in the application to reset them.
3 The number of tasks is known during compile time.
4 */
5 for (TaskImpl* task = impl.associatedTasks; (task != NULL);
6 task = task->nextTaskAtScheduler) {
7         // Doing a reset if option is set
8         task->parent.reset();
9 }
```

Table 6.3: DELOOP Results for the reset Loop in Scheduler::restart

ID	Address	Function Name	Bounds	Expected
Use-Case #1				
34	0x92a2	9Scheduler5restartEb	2	2
Use-Case #2				
34	0x970a	9Scheduler5restartEb	7	7
Use-Case #3				
34	0x91a2	9Scheduler5restartEb	6	6
Use-Case #4				
34	0x9050	9Scheduler5restartEb	5	5

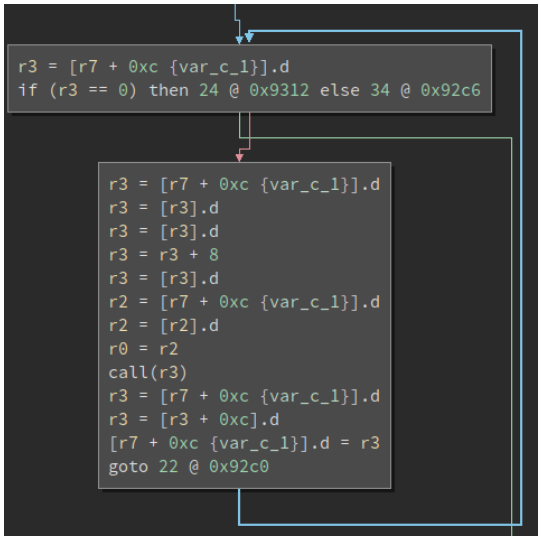


Figure 6.5: Loop in Scheduler::restart

6.3.3 Events Related Loops

The loop in Clock::dequeueAll function shown in listing 6.5 and in figure 6.6 iterates over all events in the application during termination. DELOOP detected the loop on the IR level and reported correct bounds as shown in table 6.4. Same as in table 6.1, ID in the table 6.3 refers to the loop ID, which is the ID of the last basic block in the loop body and address refers to the function starting address. The table shows the bounds dependency on the number of events and the correct estimations from DELOOP. Since the use-cases from one to four has one, two, one and five events respectively, the detected loop bounds were exactly the same numbers.

Listing 6.5: Clock::dequeueAll

```
1  /* In function Clock::dequeueAll(void)
2  The loop iterates over all events in the event queue.
3  It is called in case of terminating the application.
4  */
5  while (event != NULL) {
6          EventImpl* next = event->next;
7          event->queued = false;
8          event->next = NULL;
9          event->previous = NULL;
10         event = next;
11     }
```

Table 6.4: DELOOP Results For the loop in Clock::DequeueAll

ID	Address	Function Name	Bounds	Expected
Use-Case #1				
29	0x8d52	5Clock10dequeueAllEv	1	1
Use-Case #2				
29	0x91ba	5Clock10dequeueAllEv	2	2
Use-Case #3				
29	0x91a2	5Clock10dequeueAllEv	1	1
Use-Case #4				
29	0x9ba4	5Clock10dequeueAllEv	5	5

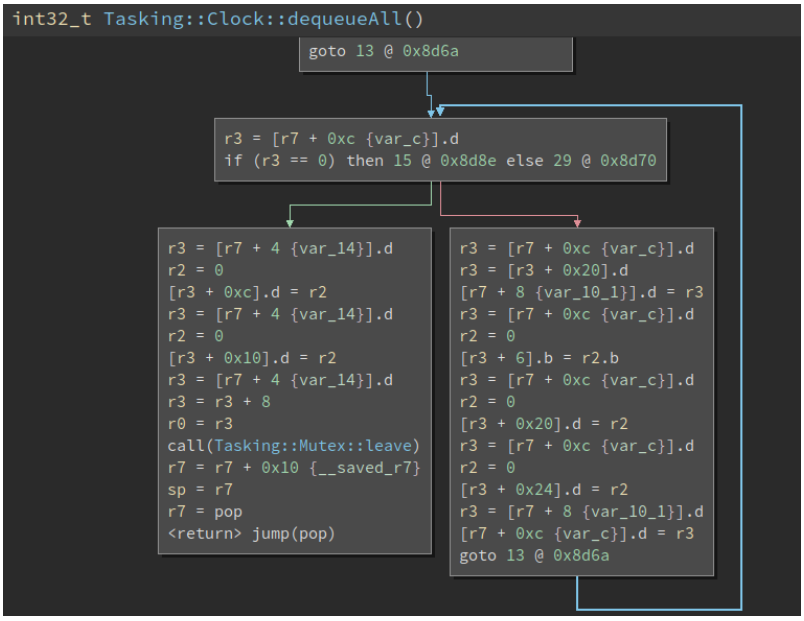


Figure 6.6: Loop in Clock::dequeueAll

6.3.4 Channel Related Loops

The channel related loops shown in listing 6.6, 6.7 and 6.8 depend on the number of inputs that are connected to the channel. Since in the use-cases, all the channels are connected to one input, we expect the execution of these loops only one time, which is proved by DELOOP as shown in table 6.5.

Table 6.5: Channel Related Loops

Address	Function Name	Bounds	Expected
Use-Case #1			
0x9a8a	7Channel4PushEEC2Ev	1	1
0x9826	16SynchronizeStartEEC2Ev	1	1
0x9878	16SynchronizeEndEEC2Ev	1	1

Listing 6.6: Loop in Channel::push

```
1 // In fucntion Channel::push(void)
2 //The loop iterates over the associated inputs to notify the channel.
3     for (InputImpl* i = m_inputs; i != NULL; i = i->channelNextInput) {
4         i->notifyInput();
5     }
```

Listing 6.7: Loop in TaskImpl::synchronizeStart

```
1 /* In function TaskImpl::synchronizeStart(void)
2 iterate over the inputs to start synchronizing the
3 access to a shared channel
4 */
5 for (unsigned int i = 0; (i < inputs.size()); i++) {
6         static_cast<ProtectedInputAccess&>(inputs[i]).synchronizeStart();
7     }
```

Listing 6.8: Loop in TaskImpl::synchronizeEnd

```
1 /* In function TaskImpl::synchronizeEnd(void)
2 iterate over the inputs to finish synchronizing the
3 access to a shared channel
4 */
5 for (unsigned int i = 0; (i < inputs.size()); i++){
6         static_cast<ProtectedInputAccess&>(inputs[i]).synchronizeEnd();
7     }
```

6.3.5 Application Related Loops

The loop presented in listing 6.9 is not a part of the Tasking Framework API and its bounding depends on the occurrence of an external event (such as a timer). Since the execution of the loop depends on the event activation, we are not able to analyze this loop or proceed in the analysis unless this event takes place. To overcome this problem, we simulate the occurrence of the event by forcing the symbolic execution to evaluate the event.isActivated() check to true. This simple modification allows the continuity of the analysis, especially for applications like the ones considered in this thesis, whose execution path does not depend on the input data. As a result, DELOOP bounds this loop to the maximum number, which is the total number of tasks.

Listing 6.9: Loop in SchedulerExecutionModel::Executor::run

```
1  /* In function SchedulerExecutionModel::Executor::run
2  This loop is not part of the API, it is part of the execution platform.
3  This is an example of loops that depend on the input-data
4  */
5  for (Tasking::TaskImpl* task = schedulerImpl->policy.nextTask(); (task != nullptr);
   task = schedulerImpl->policy.nextTask()){
6
7      if (task->event.isActivated()) {
8              schedulerImpl->execute(*task);;
9          }
10 }
```

6.3.6 DELOOP Bounding Results

DELOOP bounding results are generated typically as the results shown in table 6.6. The tool shows its ability to detect loop bounds for tool-chain functions as memset and libc-initarray. Tasking Framework allows only static memory allocation based on MISRA standards [3]. Thus, all memory allocations functionalities as memset are bounded and known during the compile time.

Table 6.6: Complete DELOOP Results For The first Use-Case

ID	Address	func. Name	Bounds
49	0xd970	memset	30
44	0xd970	memset	123
41	0xd970	memset	495
32	0xd2dc	libc-initarray	4
22	0x8754	18InputArrayProviderILj1EEC2Ev	1
17	0xaaba	10InputArray11connectTaskERNS8TaskImplE	1
34	0x92a2	9Scheduler5startEb	2
16	0xaa60	10InputArray5resetEv	1
15	0x9a8a	7Channel4PushEEC2Ev	1
28	0x9826	16SynchronizeStartEv	1
28	0x9878	14SynchronizeEndEv	1
49	0xaf42	23SchedularExecution6SignalEv	1
29	0x8d52	5Clock10dequeueAllEv	1
76	0xec14	call-exitprocs	7
88	0xec14	call-exitprocs	7
23	0x84ae	18InputArrayProviderILj1EED2Ev	1
86	0xec14	call-exitprocs	7
95	0xec14	call-exitprocs	7
69	0xf07c	call-exitprocs	7
18	0xd260	libc-fini-array	1

6.4 Re-construing the Control Flow Graph

Indirect jumps in Tasking Framework occurs mainly due to virtual methods. Virtual methods lead to the indirect jumps on the assembly level and hinder the ability to reconstruct the control-flow graph. Through its dynamic symbolic execution engine DELOOP is able to reconstruct the control-flow graph. For example, the graph in appendix B shows the reconstructed CFG on the function-level resulted from DELOOP for the first use-case shown in figure 3.6. In the graph in appendix B, the numbers inside the graph nodes indicate the function addresses, the red arrows joining nodes indicates call operation resulting from indirect branch and the numbers on the arrows refer to the order of the function-call execution.

6.4.1 Resolving Indirect Jump Example

Estimating the WCET time of the TaskImpl::synchronizeEnd function in figure 6.7 is challenging due to the indirect call instruction "call(r3)". From the snippet of the symbolic execution graph in figure 6.8, we can easily detect that the indirect branch instruction calls Input::synchronizeEnd function shown in figure 6.9 which itself has indirect call that resolves to Channel::synchronizeEnd function. In figure 6.8 the red arrows indicate indirect jumps, the numbers before the function name indicate the function address and the numbers on the arrows indicate the execution order of the function. Following the sequence diagram in figure 3.4 for the synchronization start and end functions, we can ensure the correctness of the indirect jump resolution.

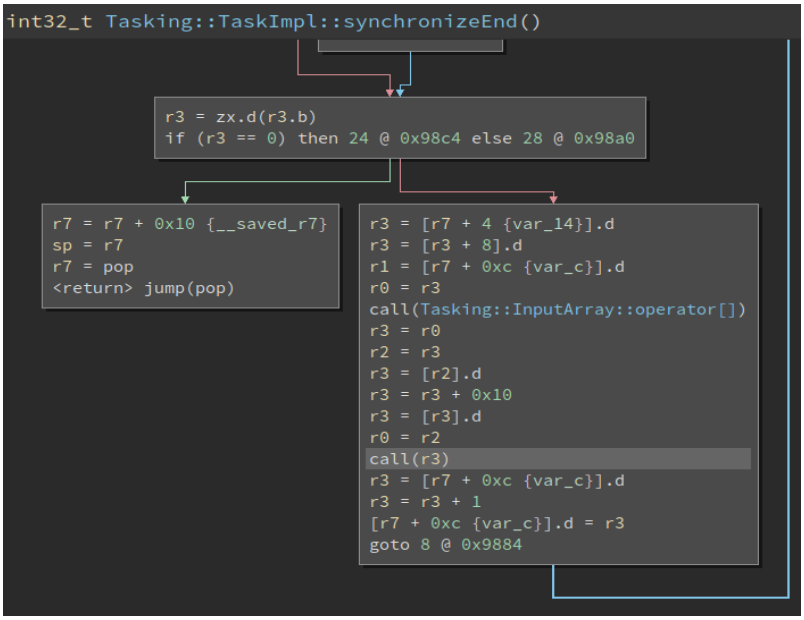


Figure 6.7: Indirect Jump in TaskImpl::synchronizeEnd Function (address:0x9878)

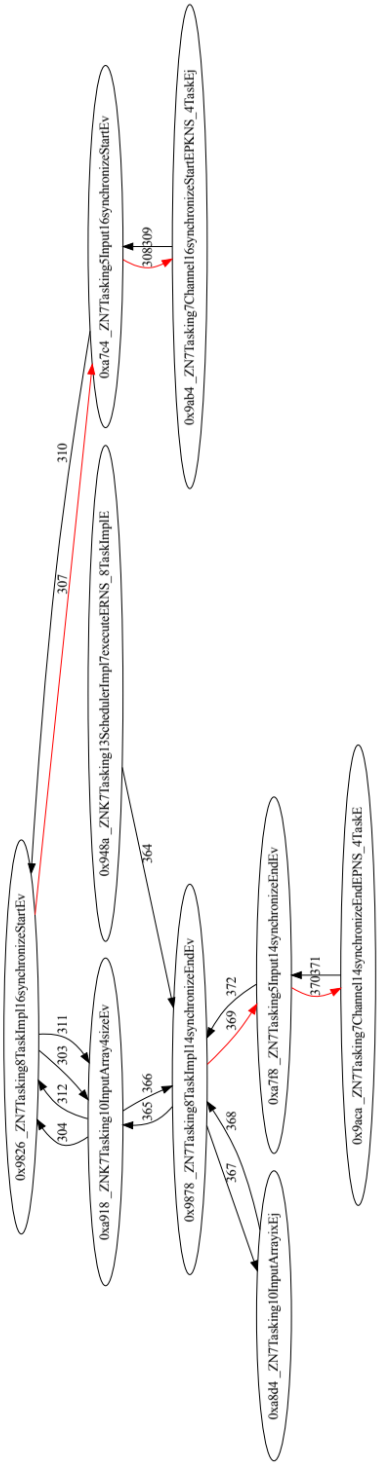


Figure 6.8: Snippet from the Symbolic Execution Graph in Appendix B

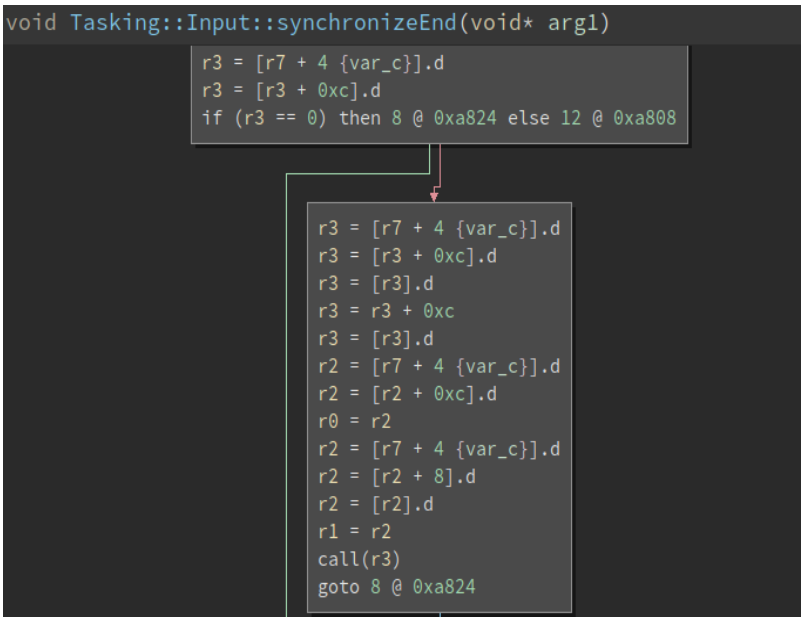


Figure 6.9: Indirect Jump in Input::synchronizeEnd Function (address:0xa7f8)

6.5 WCET Estimation

In this section, we present the WCET results for our proposed setup. Table 6.7 shows the static WCET resulting from integrating DELOOP with OTAWA for selected functions from Tasking Framework. The Scheduler::restart function has an indirect branch which is resolved and leads to another 75 function calls. This explains the large execution over-head resulting from this function call.

Table 6.7: WCET Results from Integrating DELOOP with OTAWA

Address	Function Name	# Loops	Indirect Jump	WCET (cycles)
Use-Case #1				
0x8754	18InputArrayProviderILj1EEC2Ev	1	No	670
0xaf22	10InputArray11connectTaskERNS8TaskImplE	1	No	245
0x8d52	5Clock10dequeueAllEv	1	No	350
0x92a2	9Scheduler5restartEb	2	Yes	11,200
Use-Case #2				
0x87e2	18InputArrayProviderILj3EED2Ev	3	No	1000
0x8ad0	18InputArrayProviderILj1EEC2Ev	1	No	670
0x9a2c	10InputArray11connectTaskERNS8TaskImplE	3	No	560
0x91ba	5Clock10dequeueAllEv	2	No	440
0x970a	9Scheduler5restartEb	7	Yes	38,310
Use-Case #3				
0x8650	18InputArrayProviderILj1EEC2Ev	1	No	670
0xaab2	10InputArray11connectTaskERNS8TaskImplE	1	No	245
0x91a2	5Clock10dequeueAllEv	1	No	350
0x820a	18InputArrayProviderILj1EED2Ev	1	No	270
0xd99c	9Scheduler5restartEb	6	Yes	32,470
Use-Case #4				
0xc0dc	18InputArrayProviderILj1EEC2Ev	1	No	670
0xaab2	10InputArray11connectTaskERNS8TaskImplE	1	No	245
0x9ba4	5Clock10dequeueAllEv	5	No	1025
0x9050	9Scheduler5restartEb	5	Yes	28,150

Measurement analysis was performed to measure the execution time of some of the functions in order to verify the efficacy of our proposed setup. The measurement was done using STM development board model STM32L152RET6 with cortex M3 processor connected with logic analyzer to measure the execution time. To measure the execution time of a function, an output pin is initialized to low just before calling the function to be measured. Then this pin is set to high at the beginning of the function and to a low state at the end of the function. The pin is connected to the logic analyzer and the execution time is then calculated. Given the CPU frequency we are able to calculate the CPU cycles for every function. The measurements were repeated 50 times and the average execution time was recorded in graph 6.10.

Figure 6.10 shows the comparison between the static computed WCET and the average measured execution time for selected functions. The number before the function name in the figure indicates the use-case number. For all the functions, our proposed setup results in a safe upper bound for the WCET. The graph also shows the difference in the overhead of some of the framework's functions. The execution time of some functions as InputArray::connectTask or Clock::dequeueAll can be nearly rounded to zero while it should be accurately examined for function as Scheduler::restart, especially when calculating the WCRT of the developed tasks. Although we showed the results of some selected functions, we can calculate the WCET for **all** the Tasking Framework functions thanks to our proposed analysis.

6.6 Performance

Table 6.8 shows the performance results during the analysis of the 4 use-cases. The analysis was executed on a workstation with a linux operating system, i7-9750H processor and 16GB RAM. Our analysis is slow compared to other analysis as oRange [18] or PAGAI [47] however this is irrelevant for our goal as the analysis is executed offline during the design verification and validation phase.

Table 6.8: Performance Results

Use-Case	Binary Size (Kbyte)	%CPU	Average Memory (MiB)	Execution Time (sec)
1	630	24 %	335	15
2	664	25%	640	81
3	683	25%	1200	103
4	687	26%	1300	114

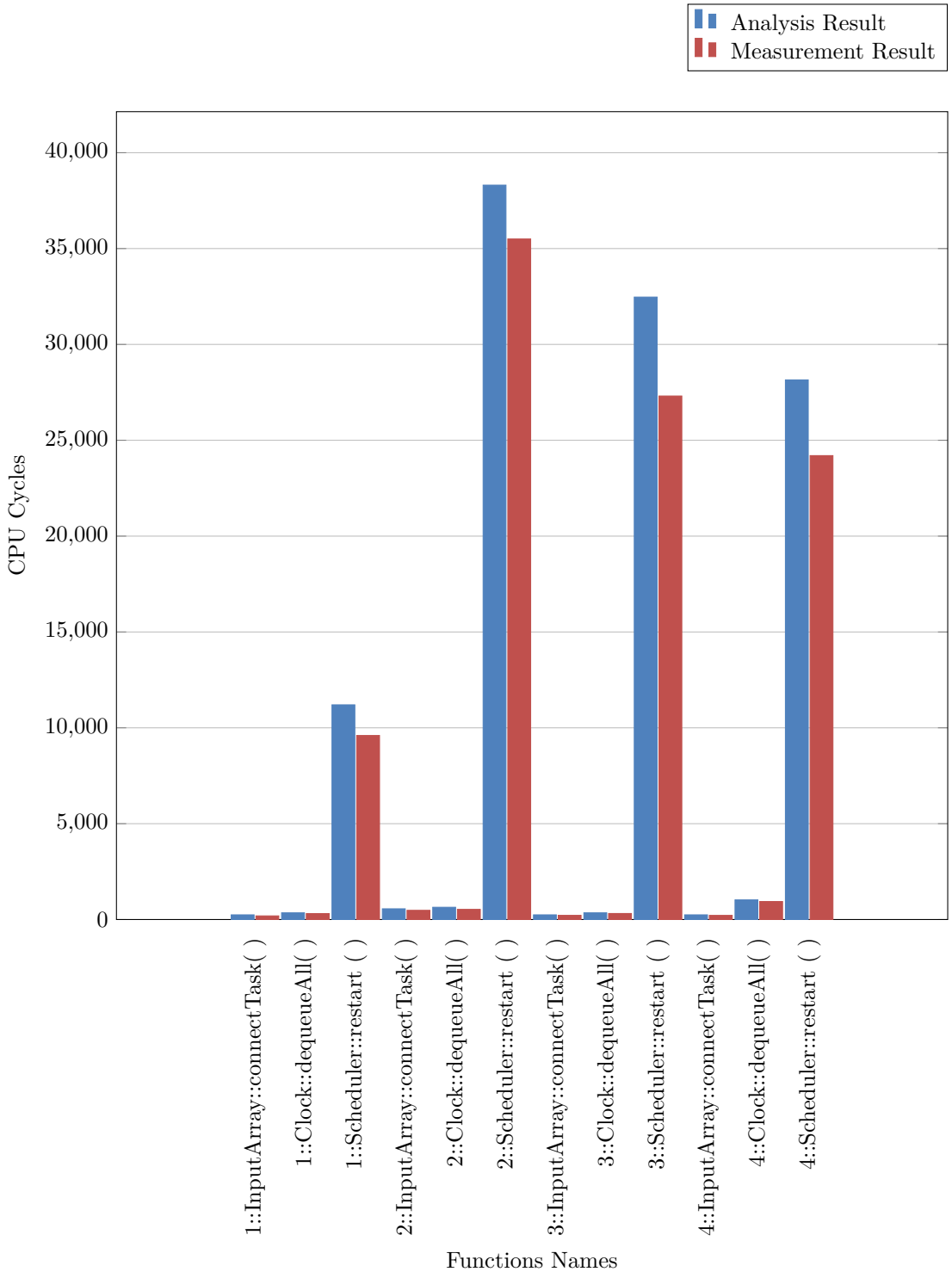


Figure 6.10: WCET for Selected Functions from the Use-Cases

Chapter 7

Discussion

7.1 Conclusion

On-board embedded systems play a critical role in space exploration. Satellites and probes are equipped with such systems to collect a massive amount of data continuously. These applications, which are used to process on-board data, follow the data-flow programming, and they have similar software structures. Developing software libraries that facilitate the development of such systems reduce the development efforts and improve the re-usability. However, they need a programming language that is both machine-oriented and abstraction capable for a better software development practices. Machine oriented means that the language should be supported with static memory implementation capabilities to meet the embedded system requirement of on-board data handling applications. C++ is the programming language that offers both features. The DLR Institute for Software Systems took the initiative and developed a C++-based software library to develop an on-board data application. The library is called Tasking Framework.

This thesis investigates the opportunities to perform a static worst-case execution analysis on applications developed using the Tasking Framework. The goal is twofold: compute to the WCET to perform a schedulability analysis and to enable the software architect to make a design decision on a budget of the computation blocks. The majority of static analyses are dedicated to C-based embedded software or can work on very simple C++ code. This work explors the available open-source tools such as SWEET and OTAWA and defines the main challenges to perform a static WCET analysis, namely the indirect jumps and loop bounds.

We found that OTAWA, which works on the binary level, is promising and can perform some functionalities of Tasking Frameworks. We proposed an analysis based on dynamic symbolic execution to overcome the two main challenges, indirect jumps, and loop bounds. We developed a tool called DELOOP to perform the proposed analysis that works on an intermediate representation level using the Z3 SMT solver to detect the loops, compute their bounds, and resolve indirect jumps. The tool shows high precision when bounding loops of the Mälardalen WCET Benchmark suite. To perform the WCET analysis on the Tasking Framework, we feed OTAWA with the loop bounds flow facts, and the function addresses flow facts obtained from DELOOP to resolve loop bounds and indirect jumps. Experimental results show that the proposed solution is practical and capable of evaluating the WCET of tasks developed using the Tasking Framework.

Using DELOOP with tasks developed by Tasking Framework eases the WCET analysis. From one side, tasks developed by Tasking Framework do not face code translation challenges as they are developed in C++ following MISRA standard [3]. Tasks developed by other data-flow frameworks as MATLAB require a specialized translator to transform the MATLAB model into analyzable assembly code. This translation process is difficult due to the high level of abstraction of the MATLAB models [55].

On the other side, analyzing data-flow tasks is challenging as each task has its operation modes and signals that conditionally eliminate each other. Depending on the current input condition/-mode, different computation blocks are triggered. Static timing analyzers as aiT [5] do not comprehend conditions. Thus, it takes the longest execution path resulting in over-approximation of the WCET. This is not the case with DELOOP, through which the symbolic dynamic execution can be tuned to examine the program behavior for a certain path/condition resulting in reasonable WCET. Through this thesis, symbolic dynamic execution proved a lot of benefits when it comes to examining the timing behavior of model-based applications.

7.2 Limitations

- One limitation of our proposed analysis is its disability to analyze parallel executing threads sharing the same resources. The single-path symbolic execution technique analyzes just one control-flow path in the program under analysis, which makes it hard to predict the behavior of parallel executing binaries.
- Additionally, the tool reports only loop bounds for a given control flow graph with a given input. Such behavior limits its ability to bound loops for state machine-designed programs, where the number of loops differs based on the state.
- Another limiting factor for the proposed tool is that it is built based on the BINARYNINJA framework. Despite its strong capabilities in lifting arm instructions, the generated IR are not consistent. Some instructions produce unpredictable variables during the analysis, which makes it hard to capture during the analysis. Besides, it still has a limited number of un-lifted instructions.

7.3 Future Work

To the best of our knowledge, BINARYNINJA was the only available tool for us to perform correct lifting. Other tools as [11] requires integration with IDA PRO [7], which is outside the budget of this thesis. We recommend that the next step is to develop a generic lifter that can integrate different instruction sets. We suggest that whether the developed lifter should lift the instruction to the generic LLVM IR or to a reverse engineering IR as REIL IR [31]. Lifting to LLVM IR facilitates the integration of LLVM analysis as value analysis and data flow analysis. On the other hand, lifting to reverse engineering IR like REIL simplifies the whole analysis as it only includes 15 IR instructions. Integration with other lifters affects both the performance and the precision of the tool. Another important next step in this project is introducing parallel executing SMT solver threads that are synchronized to perform symbolic-execution for parallel-threads architectures.

Appendix A

Hello World Example from Tasking Framework

Listing A.1: Hello world example from Tasking Framework.

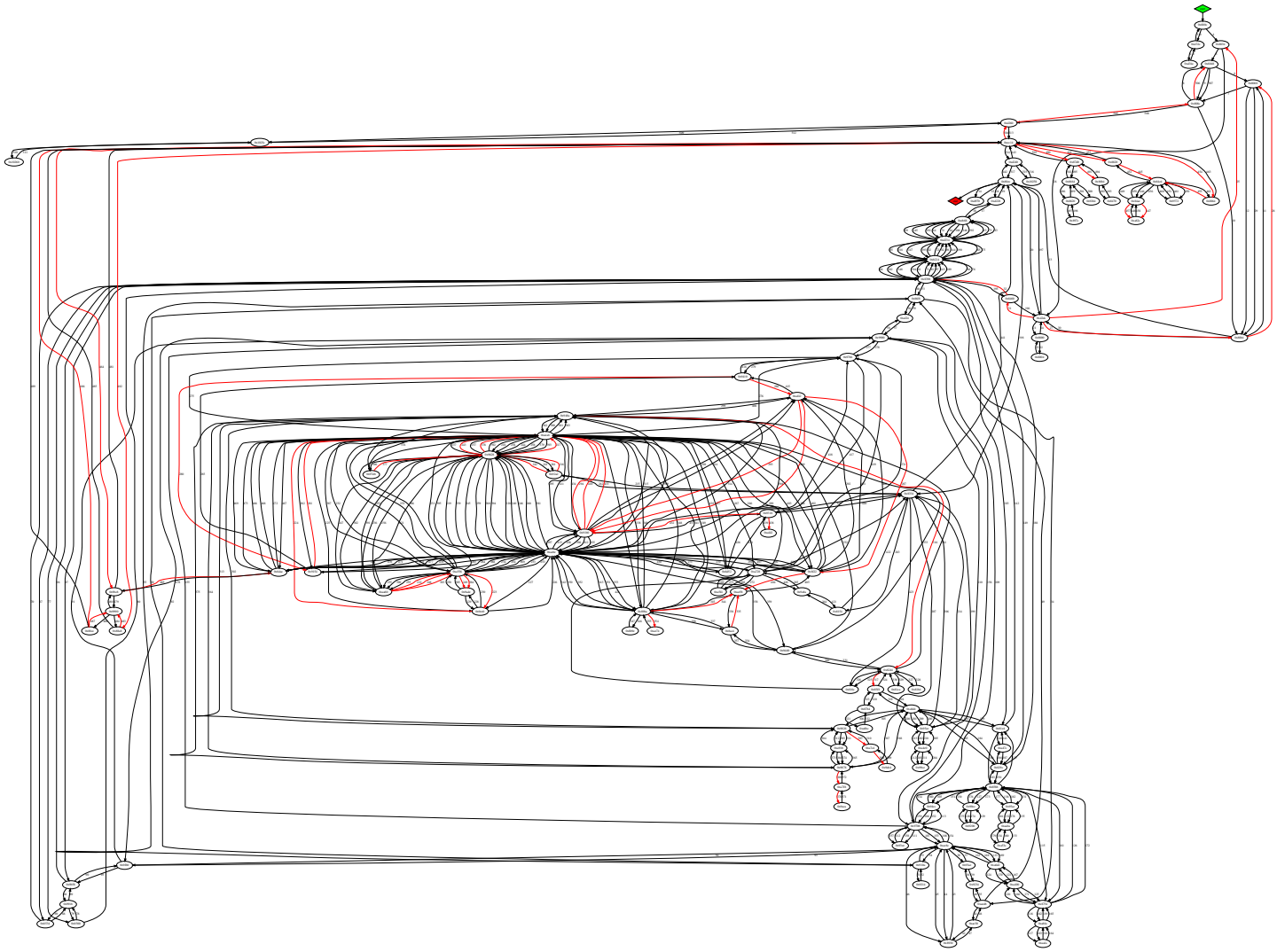
```
1  #include <schedulerProvider.h>
2  #include <schedulePolicyFifo.h>
3  #include <taskChannel.h>
4  #include <taskEvent.h>
5  #include <task.h>
6  class InputChannel : public Tasking::Channel {
7  public:
8      void doPush(void);
9      const int& getValue(void) const;
10     void setValue(int);
11 protected:
12     int sensValue = 10;
13 };
14 void InputChannel::doPush(void){push();}
15 const int&InputChannel::getValue(void) const { return sensValue;}
16 void InputChannel::setValue(int value){sensValue = value;}
17
18 class OutputChannel : public Tasking::Channel {
19 public:
20     void doPush(void);
21     const int& getProcessedValue(void) const;
22     void setProcessedValue(int);
23 protected:
24     int sensProcessedValue;
25 };
26
27 void OutputChannel::doPush(void) {push();}
```

```
28 const int& OutputChannel::getProcessedValue(void) const {return sensProcessedValue;}
29 void OutputChannel::setProcessedValue(int value) sensProcessedValue = value; }
30
31 // ------------------------------
32 class HandleInput : public Tasking::TaskProvider<1u, Tasking::SchedulePolicyFifo> {
33 public:
34     HandleInput(Tasking::Scheduler& scheduler, OutputChannel& outChannel,
35             Tasking::Event& outTime);
36     virtual void execute(void);
37 private:
38     OutputChannel& out;
39     Tasking::Event& outTrigger;
40 };
41
42 HandleInput::HandleInput(Tasking::Scheduler& scheduler, OutputChannel& outChannel,
43             Tasking::Event& outTime) :
44     TaskProvider(scheduler),
45     out(outChannel),
46     outTrigger(outTime) {
47     inputs[0u].configure(1u);
48 }
49
50 void HandleInput::execute(void) {
51     int sensValue =
52             getChannel<InputChannel>(0u)->getValue() + out.getProcessedValue() + 10;
53     out.setProcessedValue(sensValue);
54     // Wait 5 seconds
55     outTrigger.trigger(5000u);
56 }
57 // ------------------------------
58 class ModifiyValue : public Tasking::TaskProvider<1u, Tasking::SchedulePolicyFifo> {
59 public:
60     ModifiyValue(Tasking::Scheduler& scheduler, InputChannel& inChannel,
61             OutputChannel& outChannel);
62     virtual void execute(void);
63 private:
64     InputChannel& in;
65     OutputChannel& out;
66 };
67 ModifiyValue::ModifiyValue (Tasking::Scheduler& scheduler,InputChannel& inChannel,
68             OutputChannel& outChannel) :
69     TaskProvider(scheduler),
70     in(inChannel),
71     out(outChannel){
72     // Start when it is trigger on input
73     inputs[0u].configure(1u);
```

```
74 }
75 voidModifiyValue::execute(void) {
76     int sensValue = out.getProcessedValue() + 3;
77     if (sensValue < 247){
78         in.setValue(sensValue);
79         in.doPush();
80     } // else end of the program
81 }
82 // <<<<<<== instances ==>>>>>>
83 /// Instantiate the input channel
84 Tasking::SchedulerProvider<1u, Tasking::SchedulePolicyFifo> scheduler;
85 InputChannel inChannel;
86 OutputChannel outChannel;
87 Tasking::Event modifyTrigger(scheduler);
88 HandleInput handleTask(scheduler, outChannel, modifyTrigger);
89 ModifiyValue modifyTask(scheduler, inChannel, outChannel);
90
91 // <<<<<< == program code == >>>>>>
92
93 int main(void) {
94     // Connect tasks to channels
95     handleTask.configureInput(0u, inChannel);
96     modifyTask.configureInput(0u, modifyTrigger);
97     // Start Tasking scheduler
98     scheduler.start();
99     // Make the first push to notify handleTask
100    inChannel.doPush();
101    // Stop Tasking scheduler
102    scheduler.terminate(true);
103    return 0;
104 }
```

Appendix B

Symbolic Execution Graph



References

[1] https://ecss.nl/standard/ecss-q-st-40c-rev-1-safety-15-february-2017/ (cit. on p. 10).

[2] https://en.cppreference.com/w/ (cit. on pp. 16, 18).

[3] https://www.misra.org.uk/Activities/MISRAC/tabid/171/Default.aspx (cit. on pp. 17, 75, 84).

[4] http://www.bound-t.com/ (cit. on pp. 18, 21).

[5] https://www.absint.com/ait/ (cit. on pp. 20, 22, 84).

[6] https://llvm.org/ (cit. on pp. 26, 40).

[7] https://www.hex-rays.com/products/ida// (cit. on pp. 27, 84).

[8] https://solarsystem.nasa.gov/missions/rosetta-philae/in-depth/ (cit. on p. 29).

[9] https://github.com/avast/retdec (cit. on p. 40).

[10] https://binary.ninja/ (cit. on p. 52).

[11] https://github.com/lifting-bits/mcsema (cit. on p. 84).

[12] Roberto Amadini et al. "Abstract Interpretation, Symbolic Execution and Constraints". In: *Recent Developments in the Design and Implementation of Programming Languages*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2020 (cit. on p. 46).

[13] Andrew W Appel. *Modern compiler implementation in C*. Cambridge university press, 2004 (cit. on p. 49).

[14] Domagoj Babić et al. "Statically-directed dynamic automated test generation". In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 2011, pp. 12–22 (cit. on p. 28).

[15] Clément Ballabriga, Julien Forget, and Giuseppe Lipari. "Abstract interpretation of binary code with memory accesses using polyhedra". In: *arXiv preprint arXiv:1711.07257* (2017) (cit. on pp. 15, 23, 25, 42).

[16] Clément Ballabriga et al. "OTAWA: an open toolbox for adaptive WCET analysis". In: *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*. Springer. 2010, pp. 35–46 (cit. on pp. 20, 21, 42, 64).

[17] Clément Ballabriga et al. "Static Analysis Of Binary Code With Memory Indirections Using Polyhedra". In: *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer. 2019, pp. 114–135 (cit. on p. 25).

[18] Armelle Bonenfant, Marianne de Michiel, and Pascal Sainrat. "oRange: A tool for static loop bound analysis". In: *Workshop on Resource Analysis, University of Hertfordshire, Hatfield, UK*. Vol. 9. 09. 2008, p. 08 (cit. on pp. 25, 40, 59, 80).

[19] Aaron R Bradley and Zohar Manna. *The calculus of computation: decision procedures with applications to verification*. Springer Science & Business Media, 2007 (cit. on p. 48).

[20] Robert Brummayer. *Efficient SMT solving for bit-vectors and the extensional theory of arrays*. Trauner, 2010 (cit. on p. 48).

[21] Randal E Bryant et al. "Deciding bit-vector arithmetic with abstraction". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2007, pp. 358–372 (cit. on p. 49).

[22] Stefan Bygde. "Static WCET analysis based on abstract interpretation and counting of elements". PhD thesis. Mälardalen University, 2010 (cit. on p. 23).

[23] Pavel ČADEK. "Symbolic Loop Bound Analysis". PhD thesis. Masarykova univerzita, Fakulta informatiky, 2015 (cit. on p. 46).

[24] Po-Yung Chang, Eric Hao, and Yale N Patt. "Target prediction for indirect jumps". In: *ACM SIGARCH Computer Architecture News* 25.2 (1997), pp. 274–283 (cit. on pp. 16, 27).

[25] Cristina Cifuentes and Antoine Fraboulet. "Intraprocedural static slicing of binary executables". In: *1997 Proceedings International Conference on Software Maintenance*. IEEE. 1997, pp. 188–195 (cit. on p. 27).

[26] Cristina Cifuentes and Mike Van Emmerik. "Recovery of jump table case statements from binary code". In: *Science of Computer Programming* 40.2-3 (2001), pp. 171–188 (cit. on p. 27).

[27] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1977, pp. 238–252 (cit. on pp. 15, 22).

[28] Christoph Cullmann and Florian Martin. "Data-flow based detection of loop bounds". In: *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2007 (cit. on p. 22).

[29] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340 (cit. on pp. 47, 49, 51, 52).

[30] Marco Di Natale and Haibo Zeng. "Task implementation of synchronous finite state machines". In: *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2012, pp. 206–211 (cit. on p. 10).

[31] Thomas Dullien and Sebastian Porst. *REIL: A platform-independent intermediate representation of disassembled code for static code analysis*. 2009 (cit. on p. 84).

[32] Khaled ElWazeer. "Deep Analysis of Binary Code to Recover Program Structure". PhD thesis. 2014 (cit. on p. 27).

[33] Andreas Ermedahl. "A modular tool architecture for worst-case execution time analysis". PhD thesis. Acta Universitatis Upsaliensis, 2003 (cit. on pp. 13, 19, 23).

[34] Andreas Ermedahl et al. “Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis”. In: *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2007 (cit. on pp. 15, 22, 23).

[35] Heiko Falk. *Lecture notes in Compiler for embedded systems* (cit. on p. 23).

[36] Christian Ferdinand. *Cache behavior prediction for real-time systems*. Pirrot, 1997 (cit. on p. 15).

[37] Christian Ferdinand and Reinhold Heckmann. “ait: Worst-case execution time prediction by static program analysis”. In: *Building the Information Society*. Springer, 2004, pp. 377–383 (cit. on p. 13).

[38] Christian Ferdinand et al. “Combining a high-level design tool for safety-critical systems with a tool for WCET analysis of executables”. In: *Proc. of the 4th European Congress on Embedded Real Time Software (ERTS), Toulouse*. 2008 (cit. on p. 10).

[39] Ian Foster. *Designing and building parallel programs: concepts and tools for parallel software engineering*. Addison-Wesley Longman Publishing Co., Inc., 1995 (cit. on p. 30).

[40] Jorge Garrido et al. “Analysis of WCET in an experimental satellite software development”. In: *12th International Workshop on Worst-Case Execution Time Analysis*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2012 (cit. on p. 10).

[41] Jan Gustafsson, Andreas Ermedahl, and Björn Lisper. *ALF (ARTIST2 language for flow analysis) specification*. 2008 (cit. on p. 19).

[42] Jan Gustafsson et al. “ALF-a language for WCET flow analysis”. In: *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2009 (cit. on p. 22).

[43] Jan Gustafsson et al. “The Mälardalen WCET benchmarks: Past, present and future”. In: *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2010 (cit. on pp. 25, 59).

[44] Zain Alabedin Haj Hammadeh et al. “Event-Driven Multithreading Execution Platform for Real-Time On-Board Software Systems”. In: *Proceedings of the 15th annual workshop on Operating Systems Platforms for Embedded Real-time Applications*. 2019, pp. 29–34 (cit. on pp. 10, 30, 31, 33, 36).

[45] Brian Hale. “Cyberphysical Aircraft Development and Test using Industrial Linux Servers”. In: *AIAA Scitech 2020 Forum*. 2020, p. 2119 (cit. on p. 10).

[46] Christopher Healy et al. “Bounding loop iterations for timing analysis”. In: *Proceedings. Fourth IEEE Real-Time Technology and Applications Symposium (Cat. No. 98TB100245)*. IEEE. 1998, pp. 12–21 (cit. on p. 22).

[47] Julien Henry, David Monniaux, and Matthieu Moy. “Pagai: A path sensitive static analyser”. In: *Electronic Notes in Theoretical Computer Science* 289 (2012), pp. 15–25 (cit. on pp. 26, 40, 59, 80).

[48] Hajer Herbegue et al. “Hardware architecture specification and constraint-based WCET computation”. In: *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*. IEEE. 2013, pp. 259–268 (cit. on p. 14).

[49] Niklas Holsti, Thomas Långbacka, and Sami Saarinen. "Worst-case execution time analysis for digital signal processors". In: *2000 10th European Signal Processing Conference*. IE1EE. 2000, pp. 1–4 (cit. on p. 41).

[50] Niklas Holsti and Sami Saarinen. "Status of the Bound-T WCET tool". In: *Space Systems Finland Ltd* (2002) (cit. on p. 18).

[51] M Irlbeck et al. "Deconstructing dynamic symbolic execution". In: *Dependable Software Systems Engineering* 40 (2015), p. 26 (cit. on p. 46).

[52] Donald B Johnson. "Finding all the elementary circuits of a directed graph". In: *SIAM Journal on Computing* 4.1 (1975), pp. 77–84 (cit. on p. 52).

[53] Daniel Kästner and Stephan Wilhelm. "Generic control flow reconstruction from assembly code". In: *Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems*. 2002, pp. 46–55 (cit. on p. 14).

[54] Daniel Kästner et al. "Safety-Critical Software Development in C++". In: *International Conference on Computer Safety, Reliability, and Security*. Springer. 2020, pp. 98–110 (cit. on p. 16).

[55] Raimund Kirner et al. "Fully automatic worst-case execution time analysis for Matlab/Simulink models". In: *Proceedings 14th Euromicro Conference on Real-Time Systems. Euromicro RTS 2002*. IEEE. 2002, pp. 31–40 (cit. on pp. 10, 84).

[56] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. "r-TuBound: Loop bounds for WCET analysis (tool paper)". In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer. 2012, pp. 435–444 (cit. on pp. 13, 25).

[57] Christoph Krause and Florian Holzapfel. "Implementing a multi-level finite state machine with MATLAB Simulink and Stateflow in the environment of high-integrity aircraft controller software". In: *2018 4th International Conference on Control, Automation and Robotics (ICCAR)*. IEEE. 2018, pp. 147–151 (cit. on p. 10).

[58] Bal Krishna Nyaupane. *Testing a Timing Analysis tool: SWEET*. 2013 (cit. on p. 19).

[59] Marc Langenbach, Stephan Thesing, and Reinhold Heckmann. "Pipeline modeling for timing analysis". In: *International Static Analysis Symposium*. Springer. 2002, pp. 294–309 (cit. on p. 15).

[60] Xianfeng Li et al. "Chronos: A timing analyzer for embedded software". In: *Science of Computer Programming* 69.1-3 (2007), pp. 56–67 (cit. on pp. 19, 21).

[61] Ingela Lind and Henric Andersson. "Model Based Systems Engineering for Aircraft Systems– How does Modelica Based Tools Fit?" In: *8th International Modelica Conference, March 20th-22nd, Technical Univeristy, Dresden, Germany*. Linköping University Electronic Press. 2011, pp. 856–864 (cit. on p. 10).

[62] Björn Lisper. "SWEET–a tool for WCET flow analysis". In: *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer. 2014, pp. 482–485 (cit. on pp. 19, 39, 59).

[63] Paul Lokuciejewski et al. "A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models". In: *2009 International Symposium on Code Generation and Optimization*. IEEE. 2009, pp. 136–146 (cit. on pp. 15, 22, 23).

[64] Olaf Maibaum and Ansgar Heidecker. “Software Evolution from TET-1 to Eu:CROPIS”. In: *10th International Symposium on Small Satellites for Earth Observation*. Ed. by Rainer Sandau, Hans-Peter Röser, and Arnoldo Valenzuela. Wissenschaft & Technik Verlag, Apr. 2015, pp. 195–198. URL: https://elib.dlr.de/100859/ (cit. on p. 33).

[65] Olaf Maibaum, Daniel Lüdtke, and Andreas Gerndt. “Tasking framework: parallelization of computations in onboard control systems”. In: (2013) (cit. on pp. 29–31).

[66] Kewen Meng and Boyana Norris. “Mira: A framework for static performance analysis”. In: *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2017, pp. 103–113 (cit. on pp. 23, 24).

[67] Tim Menzies et al. *Sharing data and models in software engineering*. Morgan Kaufmann, 2014 (cit. on p. 28).

[68] Minh Hai Nguyen et al. “A hybrid approach for control flow graph construction from binary code”. In: *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*. Vol. 2. IEEE. 2013, pp. 159–164 (cit. on p. 28).

[69] Richard J Orgass. “McCarthy J.. Towards a mathematical science of computation. Information processing 1962, Proceedings oflFIP Congress 62, organized by the International Federation for Information Processing, Munich, 27 August-1 September 1962, edited by Popplewell Cicely M., North-Holland Publishing Company, Amsterdam 1963, pp. 21–28. McCarthy John. Problems in the theory of computation. Information processing 1965, Proceedings of IFIP Congress 65, organized by the International Federation for Information Processing, New York City, May 24–29, 1965, Volume I, edited by Kalenich Wayne A., Spartan Books, Inc., Washington, DC, and Macmillan and Co., Ltd., London, 1965, pp. 219–222”. In: (1971) (cit. on p. 48).

[70] Dwi Risdianto. “Pengembangan Standar Nasional Indonesia Produk Pelontar Satelit Kubus Berbasis Tab Berdasarkan ISO 17770: 2017”. In: *Pertemuan dan Presentasi Ilmiah Standardisasi*. Vol. 2019. Badan Standardisasi Nasional. 2019, pp. 45–52 (cit. on p. 64).

[71] Eicke-Alexander Risse et al. “Guidance, Navigation and Control for Autonomous Close-Range-Rendezvous”. In: *Deutscher Luft- und Raumfahrtkongress 2020*. Oct. 2020. URL: https://elib.dlr.de/137654/ (cit. on p. 33).

[72] Selma Saidi. *Lecture notes in Real-time systems* (cit. on p. 15).

[73] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. “Disassembly of executable code revisited”. In: *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* IEEE. 2002, pp. 45–54 (cit. on p. 27).

[74] Kurt Schwenk and Daniel Herschmann. “On-Board Data Analysis and Real-Time Information System”. In: *Deutscher Luft- und Raumfahrtkongress 2020*. Oct. 2020. URL: https://elib.dlr.de/137250/ (cit. on p. 33).

[75] Venkatesh Srinivasan and Thomas Reps. “An improved algorithm for slicing machine code”. In: *ACM SIGPLAN Notices* 51.10 (2016), pp. 378–393 (cit. on p. 53).

[76] Lili Tan. “Improving timing analysis for Matlab Simulink/Stateflow”. In: 2000 (cit. on p. 10).

[77] Lili Tan. “The worst case execution time tool challenge 2006: The external test”. In: *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006)*. IEEE. 2006, pp. 241–248 (cit. on p. 19).

[78] Stephan Theil et al. "ATON (Autonomous Terrain-based Optical Navigation) for exploration missions: recent flight test results". In: *CEAS Space Journal* 10.3 (2018), pp. 325–341 (cit. on pp. 10, 33, 37).

[79] Henrik Theiling. "Extracting safe and precise control flow from binaries". In: *Proceedings Seventh International Conference on Real-Time Computing Systems and Applications*. IEEE. 2000, pp. 23–30 (cit. on p. 27).

[80] Carl Johann Treudler et al. "ScOSA - Scalable On-Board Computing for Space Avionics". In: *IAC 2018*. Oct. 2018. URL: `https://elib.dlr.de/122492/` (cit. on p. 33).

[81] Antoine Varet and Nicolas Larrieu. "Design and development of an embedded aeronautical router with security capabilities". In: *2012 Integrated Communications, Navigation and Surveillance Conference*. IEEE. 2012, E1–1 (cit. on p. 10).

[82] Rick Veens. "Adding support for static-WCET analysis to LLVM". In: () (cit. on pp. 23, 39, 41).

[83] Alexey Vishnyakov et al. "Sydr: Cutting Edge Dynamic Symbolic Execution". In: *arXiv preprint arXiv:2011.09269* (2020) (cit. on p. 28).

[84] Reinhard Von Hanxleden et al. "WCET tool challenge 2011: Report". In: *Procs 11th Int Workshop on Worst-Case Execution Time (WCET) Analysis*. 2011 (cit. on p. 20).

[85] Mark Weiser. "Program slicing". In: *IEEE Transactions on software engineering* 4 (1984), pp. 352–357 (cit. on p. 22).

[86] I. Wenzel et al. "Automatic timing model generation by CFG partitioning and model checking". In: *Design, Automation and Test in Europe*. 2005, 606–611 Vol. 1. DOI: `10.1109/DATE.2005.76` (cit. on p. 10).

[87] I. Wenzel et al. "Measurement-based worst-case execution time analysis". In: *Third IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'05)*. 2005, pp. 7–10. DOI: `10.1109/SEUS.2005.12` (cit. on p. 10).

[88] Liang Xu, Fangqi Sun, and Zhendong Su. "Constructing precise control flow graphs from binaries". In: *University of California, Davis, Tech. Rep* (2009) (cit. on p. 28).

[89] Kai-long ZHU et al. "Construction approach for control flow graph from binaries using hybrid analysis". In: *Journal of ZheJiang University (Engineering Science)* 53.5 (2019), pp. 829–836 (cit. on pp. 16, 28).

[90] Kailong Zhu et al. "Constructing More Complete Control Flow Graphs Utilizing Directed Gray-Box Fuzzing". In: *Applied Sciences* 11.3 (2021), p. 1351 (cit. on p. 27).