Technische Universität Hamburg

Master's Thesis

Dynamic Symbolic Execution for Enhanced Intermediate Representation of Data Flow Space Applications

Author:
Hany Abdelmaksoud

Degree Program:
M.Sc. Mechatronics

First Examiner:
Prof. Dr.-Ing. Görschwin Fey

Second Examiner:
Prof. Dr.-Ing. Thorsten A. Kern

Supervisor:
Dr.-Ing. Zain A. H. Hammadeh
German Aerospace Center (DLR)

March 23, 2022

TUHH
Hamburg University of Technology

DLR Deutsches Zentrum für Luft- und Raumfahrt
German Aerospace Center

ABSTRACT

Verifying the safety and security requirements of embedded software requires a code analysis. Many software systems are developed based on software development libraries; therefore, code specifications are known at compiling time. Hence, many source-code analyses will be excluded, and low-level intermediate representations (LLIRs) of the analyzed binaries are preferred. Improving the expressiveness of the LLIR and enhancing it with more information from the binaries will improve the tightness of the applied analyses. This work is interested in developing a lifter that lifts binaries into an enhanced LLIR and can resolve indirect jumps. LLVM is used as the LLIR.

Our proposed lifter, which we call DEL (Dynamic symbolic Execution Lifter), combines both static and dynamic symbolic execution and strives to fully recover the analyzed program's control flow. DEL consists of an API to translate ARMv7-M assembly instructions into static single assignment LLVM instructions, an LLIR to Z3 expressions parser, a memory model, a register model, and a specialized condition flags handler. This work used a case study based on a software development library for onboard data-handling applications developed at the German Aerospace Center (DLR), which is called the Tasking Framework. DEL demonstrated high accuracy of around 93% in resolving indirect jumps in our case study.

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.

Hany Abdelmaksoud

Hamburg, March 23, 2022

Contents

List of Figures

List of Tables

Acronyms

AOCS	Attitude and Orbit Control System
APSR	Application Processor Status Register
ARGV	Argument Vector
ASAP	As Soon As Possible Scheduling
ATON	Autonomous Terrain-based Optical Navigation
BIRD	Bi-spectral Infrared Detection Representation
CFG	Control Flow Graph
CPSR	Current Processor Status Register
DEL	Dynamic symbolic Execution Lifter
DLR	Deutsches Zentrum für Luft- und Raumfahrt
DSE	Dynamic Symbolic Execution
DSEIR	Dynamic Symbolic Executable Intermediate Representation
Eu:CROPIS	Euglena Combined Regenerative Organic Food Production In Space
ICFTS	Indirect Control Flow Targets
IR	Intermediate Representation
ISA	Instruction Set Architecture
ODARIS	On-board Data Analysis and Real-time Information System
OSRA	Offset Shifted Range Analysis
PIC	Position Independant Code
ScOSA	Scalable On-Board Computing for Space Avionics
SET	Simple Expression Tracker
SSA	Static Single Assignment
SSE	Static Symbolic Execution
UCSE	Under Constrained Symbolic Execution
WCET	Worst Case Execution Time

Chapter 1

Introduction

Designing embedded systems for space applications is a complicated process. The modernization of aerospace systems has given rise to more-electric technologies and tightly interconnected architectures, contributing to a considerable increase in design complexity. Various architectural design approaches develop these systems effectively and efficiently. Because of its capacity to address the complexity of systems, the model-based approach is amongst the most ubiquitous design techniques [10]. The model-based approach entails creating models as rudimentary blocks, which create the entire embedded system's software using code that is generated automatically [10]. This approach enhances productivity and guarantees the correctness of the software as the applications are implemented in a structured and error-proof manner [10]. There are numerous software analysis techniques that can be applied on the developed models including data flow analysis [12, 72], worst-case execution time analysis (WCET) [80], input/output analysis [54], and security analysis [33]. Nevertheless, their analysis becomes increasingly challenging because the developed models are intricate and dynamic.

The Tasking Framework is a model-based framework developed by the German Aerospace Center's Institute for Software Technology (DLR). This software library supports the scheduling of embedded software space systems. Tasks are represented in the Taking Framework as graphs of tasks with arbitrary activation patterns. It is implemented in C++ and follows an event-driven paradigm. The framework's capabilities have been applied to a wide range of non-safety critical aerospace applications since its inception, including [76] and [45]. The Tasking Framework must be certified for use in safety-critical applications, which can be indeed an arduous process. The ECS-Q-ST-40C is the most commonly used certification standard for aerospace-embedded programs and the standard for validating implementation and verification tools [1]. It defines five levels of design assurance, varying between E-Level, which requires the least amount of testing and verification, to A-Level, which necessitates significant testing and verification. The standard demands the Tasking Framework at the very least to demonstrate functional correctness and the absence of dangers in the software to get qualified for the C-Level certification. The presented proofs must not be vulnerable to any logical or reasonable objections. It is required to show that all real-time tasks are completed on time or that missing the deadlines will not jeopardize the system's safety [10].

The computation of the WCET helps establish deadline correctness. A precise WCET analysis requires detailed architectural knowledge. Performing a WCET analysis to the source code of the Tasking Framework makes it language-dependent, while analyzing its binaries is hardware-dependent. Consequently, analyzing at the intermediate representation (IR) level proves a more viable option. An IR is the data format used inside a compiler or virtual machine to represent source code. It is designed to be suitable for post-processing, for instance, optimization and translation. It should accurately represent the source code with no loss of information – and independent of any specific source or target language. Strictly speaking, there are two approaches to acquiring the IR of a program, as shown in Figure 1.1, is either through compiling source code or through lifting binaries.

Nevertheless, the high-level IR acquired from compilers lacks information about the memory model and ignores linking effects. As a result, low-level IRs acquired through binary lifting have become an increasingly attractive alternative for performing software analyses at the intermediate level of a program. Even so, due to underlying factors such as the handling of condition flags and resolving indirect control flow targets, it is pretty challenging to generate a highly accurate and 100% representative IR.

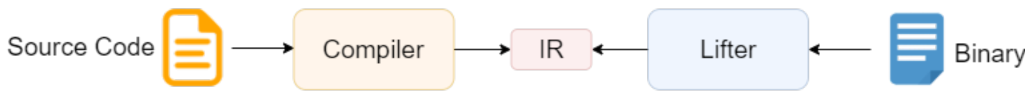


Figure 1.1: The two approaches to get an IR, compiling source code and lifting binaries.

An IR should be able to capture the control flow of the original program. A control flow graph (CFG) illustrates such flow. Figure 1.2 showcases the CFG of a simple Python program. A CFG represents all possible paths a program takes during execution in program analysis. The CFG consists of nodes exhibiting blocks of instructions and directed edges exhibiting control flow jumps [90]. The CFG is the cornerstone of numerous program analysis techniques, such as taint analysis [44, 84] and symbolic execution [53, 73]. The CFG is also prominent in program verification [42, 70], malware detection [28, 47], code similarity analysis [63, 74], and software vulnerability detection [49, 87]. Consequently, implementing the right approach to generate a complete and accurate CFG while lifting to an IR is imperative [90].

Nevertheless, indirect jumps present a challenge when constructing complete CFGs [31]. We can classify a jump instruction as either direct or indirect. A direct jump has a statically determined target which refers to a specific location in the program; however, for an indirect jump, the jump target is execution-dependant and is only known at run-time [90]. In most cases, indirect branches provide dynamic programming behaviors by implementing standard programming constructs such as function pointers and virtual function calls [90]. While indirect jumps are ubiquitous and helpful, a purely static analysis often fails to resolve an indirect jump's target due to its dynamic nature, which poses intrinsic issues when lifting into an IR module that mirrors a complete CFG.

There are two lifting solutions available today: static lifting and dynamic lifting. Static techniques do not require executing the target programs; instead, they only need to examine their code structure. These approaches offer high code coverage at a low time cost. As a result, static lifting tools like McSema [2] are used extensively in a wide range of analyses. Nevertheless, static techniques lack completeness because of their inability to resolve indirect jump relations [90]. A dynamic lifter such as BinRec [14] on the other hand, runs programs on a set of test suites and acquires control flow information from the traces of the execution while lifting. This dynamic approach is capable of resolving several indirect jumps. However, the precision of the CFG constructed by it is determined by how well the test cases cover indirect jumps. Xu et al. [85] proposed forcing a program's execution to investigate both possible paths of each conditional branch in order to increase test case coverage. Although forced execution is a powerful tool for analyses, it still lacks sufficient coverage for large-scale programs.

In this thesis, we propose a novel hybrid static-dynamic symbolic execution lifter (DEL) that

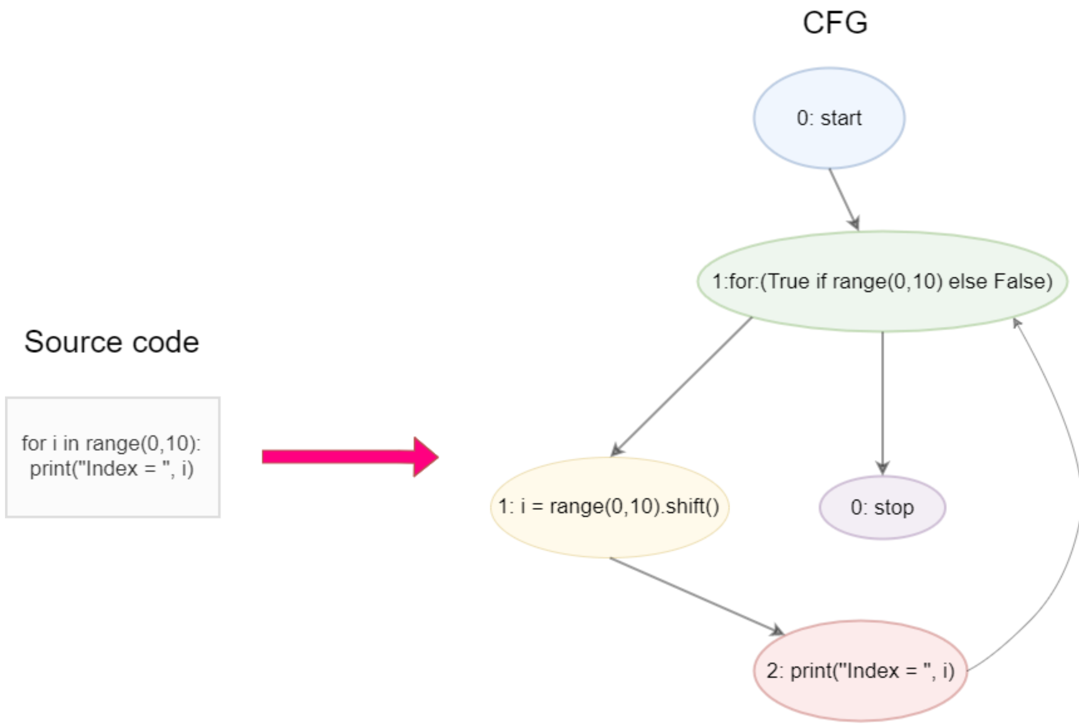


Figure 1.2: From source code to CFG.

combines both dynamic and static techniques while lifting to a low-level virtual machine (LLVM) [3] IR. Our approach is motivated by the fact that each indirect jump in the program can have multiple potential-jump targets. Each indirect jump's target calculation depends on an input-based potential execution path starting from the program's entry point up to the basic block that terminates with the indirect jump in question.

Our key insight is to combine static and dynamic symbolic execution while lifting to resolve all potential-jump targets of each indirect branch instruction. More specifically, The static part of our approach aims to construct a preliminary IR module while generating a mathematical expression for each potential-jump target address of each indirect jump detected in the binary we aim to lift. The dynamic part then performs the dynamic symbolic execution (DSE) [78] of the statically generated IR module to resolve all individual expressions of each potential-jump target of all indirect jumps to a concrete value. DEL uses Microsoft's Z3 [37] solver for its DSE engine. We suggested an iterative approach of varying the program's input during the dynamic analysis continuously. By varying the program input, we aim to explore all possible execution paths leading to an indirect jump instruction's basic block and resolve all possible potential-jump targets for each indirect jump detected. As the final output, we consider the IR constructed through multiple iterations.

In order to lift into a more complete and representative IR for the Tasking Framework, DEL implements its memory and register model. The lifted IR module captures the effects of each instruction on the state of the memory and register model and the effects of each instruction on the state of the condition flags.

1.1 Contribution

Nowadays, binary lifting tools rely heavily on static disassembly techniques and heuristics to disassemble binaries, an approach that fails to identify indirect control-flow targets, accurately distinguish between data constants and code pointers, and correctly interpret instruction and data byte boundaries. In this thesis, we present DEL, a new optimized approach of dynamic binary lifting to a low-level IR. DEL makes it possible to make use of current IR-level compiler analyses on binaries where static lifting falls short. DEL integrates symbolic execution into the lifting process to generate an enhanced IR that models the state of the memory, registers, and condition flags of the program as it executes. We divided our work into the following tasks.

- **Overview of the state-of-the-art lifting tools and techniques**

  We present an overview of the available lifting tools and techniques.

- **Introducing a new hybrid approach for lifting binaries into an enhanced intermediate representation**

  We introduce a new hybrid lifting approach that tackles issues related to existing lifting tools.

- **Comprehensive Evaluation**

  We evaluate the percentage of indirect control flow targets resolved by our lifting approach for a given test case of the Tasking Framework.

1.2 Structure

Following is an outline of this thesis. Chapter two discusses program analysis at the IR level and presents an overview of the state-of-the-art lifting tools and techniques. Chapter three presents the Tasking Framework and its relevance for IR analysis. Chapter four presents our new dynamic lifting tool DEL and its enhanced generated IR module features. Chapter five discusses the results of our approach. Finally, Chapter six presents our discussion and future steps.

Chapter 2

State of the art

Many safety and security analyses can be performed on the IR of a program. An application could have one of two IRs, one that is spawned from binary lifters, the other from compilers. Both IRs possess expressive capabilities that set them apart. IRs obtained from source code exhibit high-level language constructs such as loops and functions. Alternatively, IRs obtained from a binary lifter do not have to take into account these language abstractions in their underlying syntax tree. Using a binary lifter to derive the IR of a program could be especially useful when the analyses require information that cannot be extracted from the source code. This thesis focuses on IRs acquired from binary lifters and does not consider the work of compilers in IR generation.

In an IR code analysis, various analysis techniques are typically used to model data types, flows, and control paths of the program being analyzed. The refined model can then be evaluated to identify well-known security issues. The results can be compiled into comprehensive vulnerability reports with effective practical countering actions to tackle such vulnerabilities.

There are two approaches when analyzing at the IR level: static and dynamic analyses.

- **Static analysis**

  A static analysis examines programs to obtain specific code characteristics and behaviors before it is run. It is extensively utilized in many compiler optimizations and program analyses. It gives the chance to collect information about programs without executing them, thus acquiring a minimal or zero runtime cost [88]. A static analysis typically identifies bugs preceding the execution of a program (e.g., between coding and unit testing).

- **Dynamic analysis**

  A dynamic analysis entails examining a particular program as it runs. Various tools are available for dynamic analysis, including profilers, checkers, and execution visualizers. A program could have code for analysis incorporated fully inline or external routines that are invoked by the inline analysis code. This code runs in the background, not disrupting the program's normal execution (other than maybe slugging it down), but instead carries out additional work (during the analysis session), such as checking for bugs or assessing performance [61]. A dynamic analysis pinpoints potential bugs that may appear when a program is run (e.g., during unit testing).

Both strategies complement one another. A static analysis is generally reliable, as it considers all execution paths in a program. A dynamic analysis, however, is usually less pessimistic than a static analysis because it employs real values "in the perfect light of runtime" [38]. However, it lacks sound reasoning, as it only examines a single execution path [41]. Consequently, in reality, a dynamic analysis tends to be far less complex than a static analysis.

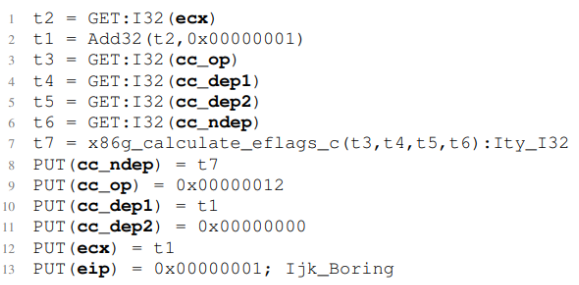
As we move forward, we examine the first step of analyzing at the IR level: the generation of an IR module through binary lifting.

2.1 Binary lifting

Binary lifting is transforming a binary executable into a higher-level intermediate language. A crucial part of binary translation, analysis, and instrumentation applications is the translation of low-level machine instructions into higher-level IR [52]. A mapping table between machine instructions and IR is usually manually created in these systems. The mapping table designates a single or a set of IR instructions to each assembly code instruction in the Instruction Set Architecture (ISA). ISA cross-compatibility is typically achieved with this method. A formal definition of binary lifting adapted from [52] is as follows:

Definition 1 (Binary Lifting).
Binary lifting is a function $\uparrow^{tar}_{ins}$: $I^{ins}_{ISA} \rightarrow I^{tar}_{IR}$, where I_{ISA} is an instruction from a specific ISA, I_{IR} is an IR instruction, *ins* is the name of an ISA and *tar* is the target IR we would like to lift to.

For example, I^{x86}_{ISA} means x86 assembly language, and I^{VEX}_{IR} is VEX IR. $\uparrow^{vex}_{x86}$ is a function to which an x86 binary code is given as input and outputs a translated VEX instance. So the expression $\uparrow^{x86}_{vex}$(0x41) lifts the binary instruction 0x41 into a VEX IR instance as highlighted in Figure 2.1, 0x41 is the inc ecx when decoded. A tool that conducts this process of lifting binaries is called a binary lifter.



```
t2 = GET:I32(ecx)
t1 = Add32(t2,0x00000001)
t3 = GET:I32(cc_op)
t4 = GET:I32(cc_dep1)
t5 = GET:I32(cc_dep2)
t6 = GET:I32(cc_ndep)
t7 = x86g_calculate_eflags_c(t3,t4,t5,t6):Ity_I32
PUT(cc_ndep) = t7
PUT(cc_op) = 0x00000012
PUT(cc_dep1) = t1
PUT(cc_dep2) = 0x00000000
PUT(ecx) = t1
PUT(eip) = 0x00000001; Ijk_Boring
```

Figure 2.1: A lifted IR instance of Valgrind [62].

The term Binary-Based IR was first introduced by [52] to differentiate between two types of IRs: one derived from binary lifters, the other from compilers. The main distinguishing feature between Binary-Based IRs and IRs from compiler theory [11] is their expressive ability [52]. IRs generated from source code exhibit high-level language constructs such as loops and functions. However, Binary-Based IRs do not need to take such language components into account in their abstract syntax tree [52]. Binary analysis tools like Valgrind and bap create their own Binary-Based IRs to convey the semantics of binary code at a low level.

Strictly speaking, Binary-Based IRs have two main properties: explicitness and self-containment [52]. A Binary-Based IR is said to be explicit if it updates only a single variable in the execution context. On the other hand, the self-containment property of a Binary-Based IR basically demonstrates whether or not it fully reflects the relevant binary code semantics. For instance, In QEMU [24], the semantics of binary instructions are often expressed with external functions. Here, is an example from [52] of a logical AND instruction in x86: pand xmm0, xmm1. Upon lifting the instruction to the Binary IR of QEMU (TCG), the IR instance directly forwards both operands to an external function named pandxmm rather than explicitly defining its operation within the IR's semantics. In this scenario, [52] argued that the IR instance is not self-contained since it has a side-effect.

Typically, in IR analysis, The explicitness helps perform control- and data-flow analyses; however, self-containment makes it possible to conduct analyses without unwanted over-approximation [52].

2.1.1 General Phases of Binary Lifting

This section highlights the common steps that a binary lifting tool goes through to transform binaries to a higher-level intermediate representation. Figure 2.2 illustrates these steps according to the logical order of their application to low-level code.

The first step in lifting binary code is to disassemble it. In the next section, we will discuss the different disassembly methods currently used in practice.

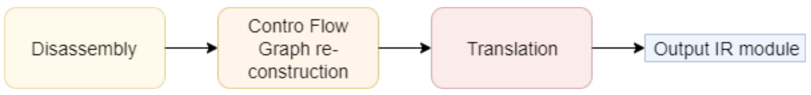


Figure 2.2: Binary lifting stages.

- **Disassembly**

  Disassembly is the translation of a program from machine code into assembly language [55]. Next, we highlight the two different disassembly techniques currently employed by existing binary analysis tools.

  - **Disassembly methods**
    - **Static disassembly**
      A static disassembler reads the binary from a file and parses the headers and section contents to disassemble it. This technique has zero runtime overhead because all of the work takes place offline. When utilized by tools like profilers and binary rewriters, the output of a static disassembler can boost performance [20]. The GNU objdump utility is a good example of a static disassembler.
      By far, the most well-known disassembler for static analysis and reverse engineering is IDA pro [4]. To find function start addresses, IDA Pro utilizes a depth-first call-graph traversal. The disassembler can accurately identify only functions that are directly called. For indirect function calls, however, it uses heuristics like scanning for conventional function prologue patterns. Nevertheless, the applied heuristics are not portable to other architectures and are complex to implement [40]. Even with the high static disassembly coverage of IDA Pro, it cannot be used in analyses that have no tolerance for intermittent errors in the disassembly output [20].
    - **Dynamic disassembly**
      A dynamic disassembler interacts with the software to be disassembled. Each instruction is deconstructed before it is executed as the software runs. The key benefit of this method is that data and code can be differentiated because the

disassembler only disassembles the instructions that will be executed. As instructions are deconstructed and executed, it becomes possible to use dynamic disassemblers with self-modifying code [20].
Because control must be passed to the disassembler before each instruction can be carried out, the performance of dynamic disassemblers is their worst flaw. In other words, the application runtime is significantly slowed down since control must be handed to the disassembler prior to the execution of each instruction. Moreover, disassemblers that use dynamic disassembly do not provide full code coverage since such a technique only disassembles specific program paths, which are executed given a predefined program input [20].
Despite the runtime overhead and low coverage of dynamic disassembly, [20] argued that the approach's ability to resolve indirect control flow targets makes it very useful for much current instrumentation and binary analysis tools, including Pin [58] and Valgrind [62].

```
804964a: bf 00                 nop
804964c: 55                 push%ebp
804964d: 89 e5              mov %esp,%ebp
804964f: 53                 push%ebx
8049650: 83 ec 04           sub $0x4.%esp
8049653: eb 04              jmp 0x8049658
8049655: e6 02 04           <junk>
8049658: be 05 00 00 00     mov 1$0x5,%esi
```

Listing 1: Dynamic Disassembly Output.

```
804964c: 55                 push%ebp
804964d: 89 e5              mov %esp,%ebp
804964f: 53                 push%ebx
8049650: 83 ec 04           sub $0x4.%esp
8049653: eb 04              jmp 0x8049658
8049655: e6 02              out 0x2, al
8049657: 04 be              add al, 0xbe
8049659: 05 00 00 00 12     add eax,0x12000
```

Listing 2: Linear Disassembly Output.

- **Disassembly algorithms**
  - **Linear sweep**
    Utilizing the linear sweep algorithm is the easiest and quickest way to disassemble binaries [20]. The GNU disassembler, objdump, is based on such algorithm [5]. The disassembly commences from the entry point found in the binary's header in virtually every binary. Each consecutive instruction is disassembled from the subsequent position, which is determined by adding the current instruction's length to its start address.
    Linn and Debray's publication [57] is the foundation for the linear sweep method. Algorithm 1 from [57] below is a pseudocode depiction of the linear disassembly approach's theoretical implementation.
    The linear sweep algorithm, however, has its shortcomings. Its main flaw is that it cannot differentiate between data and code. Any data contained in the code is disassembled incorrectly [20]. Above is a sample log from [20] that includes the attested disassembled output using a dynamic disassembler shown in Listing 1 and that of objdump shown in Listing 2. [20] demonstrated through the runtime disassembler output that some garbage bytes are stored following the jump instruction. The jump target follows the current instruction, 0x8049658, by 0x4 bytes. They argued that it is possible that the garbage bytes are perhaps data or merely alignment bytes. When using a linear disassembler, following the rendering of the two-byte jump instruction that appears at address 0x8049653, the disassembler proceeds with decoding at address 0x8049655, which is most likely not code. As a result, the actual jump destination is wrongly deconstructed, and

the output is a jump in the middle of the instruction [20].

Algorithm 1 Linear Sweep Algorithm

```
Require: $startAddress, endAddress$
  procedure LINEARSWEEP(addr)
    while $startAddress \leq addr < endAddress$ do
        $I \leftarrow$ decode instruction at address (addr)
        $addr = addr + length(I)$
  procedure MAIN
    $startAddress \leftarrow$ address of the first executable byte
    $endAddress \leftarrow$ address of the last executable byte
    linear sweep(entry point)
```

- ∗ **Recursive traversal**
  The recursive traversal algorithm traverses through one starting address to one end address in a sequential manner for every recursive traversal call. If the algorithm has visited an address already, the procedure will return. Otherwise, the algorithm decodes the current address instruction and checks whether it is a jump or a call instruction. In this approach, potential branches and function calls are followed to identify new controlling edges.
  Algorithm 2 from [57] highlights the pseudocode characterization of a recursive traversal algorithm. Because it considers the control flow in the binary, the recursive traversal approach has several advantages over the linear sweep [20]. For example, data is not falsely identified as code. As a jump instruction is disassembled, the disassembler decodes the jump target rather than heedlessly disassembling the next instruction. However, code accessed by indirect control flow transfers is not disassembled by a recursive disassembly algorithm [20].

Algorithm 2 Recursive Traversal Algorithm

```
Require: $startAddress, endAddress$
  procedure RECURSIVETRAVERSAL(addr)
    while $startAddress \leq addr < endAddress$ do
          if $addr$ has already been visited then return
          $I \leftarrow$ decode instruction at address (addr)
          markaddr as visited
          if I is branch or function call then
              for all possible targets $t$ of $I$ do  recursiveTraversal(t)
              return
          else
              $addr = addr + length(I)$
  procedure MAIN
    $startAddress \leftarrow$ address of the first executable byte
    $endAddress \leftarrow$ address of the last executable byte
    linear sweep(entry point)
```

- **Control flow graph re-construction**

  For binary analyses, a Control Flow Graph (CFG) is indispensable. It is a graph that illustrates all paths that could potentially be taken throughout the execution of a program. Figure 2.3 shows a sample CFG built for the Mälardalen WCET crc benchmark binaries [6]. In this example, there are three functions, icrc, icrc1, and main. Each function in the source code is viewed as a cluster of interlinked basic blocks. Each basic block is represented by a node containing the address of the first instruction in that basic block.

A CFG is required for conducting an accurate IR analysis. For most IR analysis algorithms, the flow of the program being analyzed is a crucial consideration, and hence such algorithms indeed require a CFG. Following the disassembly of binary code, it is necessary to create a CFG or build on the premise of one created before disassembly.

There are numerous algorithms to choose from for building CFGs. A disassembler can readily determine the targets of the edges caused by direct branches and call instructions and append the edges for them to the CFG. Like any static disassembler, IDA Pro, for example, creates a CFG with only the direct branch and call instructions as edges without considering indirect jumps.

The traditional method for creating a CFG is to begin at the start of a function and continue through instructions. At first, the CFG has neither nodes nor edges. The algorithm begins at the point of entry, and whenever a jump command is found, the current basic block ends. Generally, a basic block contains instructions devoid of branching instructions or targets of branching instructions between them. In other words, in a basic block, an instruction is executed prior to the instructions in subsequent addresses in the same basic block, with no instruction being executed in the middle [88].

For example, in the code snippet of Listing 3 of the function icrc1 in Mälardalen WCET crc benchmark, Figure 2.4 shows that the icrc1 function is constructed of 7 basic blocks. Each basic block ends with a branching instruction such as the basic block labeled BB1 or ends with an assembly instruction immediately preceding an instruction targetted by a jump instruction. A good example for the latter case would be BB6, where its last instruction of address 0x00008084 directly precedes the load instruction of address 0x00008088, which is the target address of the unconditional branching instruction 0x00008078 of BB5.

```c
unsigned short icrc1(unsigned short CRC, unsigned char onech)
{
        int i;
        unsigned short ans=(crc^onech << 8);

        for (i=0;i<8;i++) {
                if (ans & 0x8000)
                        ans = (ans <<= 1) ^ 4129;
                else
                        ans <<= 1;
        }
        return ans;
}
```

Listing 3: Icrc1 source code from Mälardalen WCET crc benchmark [6].

Even though the standard approach for CFG construction mentioned above is frequently used in analyzing the control flow of both the source and intermediate level representations generated by compilers, it cannot be applied in the opposite direction when binaries are statically lifted into IR. There are indirect calls and jumps where the targets can only be found in registers or memory. As a result, it is not always feasible to determine the destination of indirect calls and jumps statically. The targets of indirect calls and jumps can result from data segments that are globally initialized, such as function tables and jump tables [88]. They could rely on the input set, which is difficult to establish statically. In light of this, existing static analyses can either be cautious, reasoning that an indirect jump can leap to any basic block, any instruction, or in the middle of an instruction, or perhaps arbitrarily supposing that every indirect jump can only step into a limited number of targets [88].

Balakrishnan et al. devised the Value Set Analysis (VSA) algorithm for statically analyzing the memory contents in binary code [21]. On-the-spot detection of control flow boundaries caused by indirect calls is possible using this method. Here, the aim is primarily to create an IR for binary code analogous to the IR produced by a compiler from the source code. Firstly, the algorithm used in this technique takes as input the assembly code provided by IDA Pro, which includes procedure boundaries and an incomplete CFG. Secondly, a value set analysis is conducted to develop a complete CFG. With its coupled numeric and pointer analysis algorithm, VSA calculates a rounded-off set of values or addresses that could be stored in each register and memory location [22]. Generally speaking, VSA can be helpful when analyzing indirect jump targets or even analyzing the potential targets of ”read” and ”write” operations in memory. However, due to failed branch conditions tracking, value set analysis can suffer from a lack of accuracy [56].

In light of the approaches mentioned above, it becomes clear that a strictly static approach hinders the accurate reconstruction of a CFG from binary code. Consequently, resorting to the dynamic execution of the program being analyzed has become a more appealing option. The goal is to run programs against a suite of test cases and acquire the control-flow data from the traces of the execution [90]. This method can resolve indirect jumps and capture an accurate control flow. However, the ability of the test cases to cover all indirect jumps determines the completeness of the CFG created using this method [10]. Conventional dynamic analysis tools handle only a limited part of the program execution routes. In light of this, [85] has implemented forced execution to increase the code coverage. In forced execution, the code is run symbolically to examine both pathways at each branch point, and the indirect branches' targets are retrieved in a scale-able manner at run time. Using the same rationale to resolve indirect jumps, Syder [78] implemented the dynamic symbolic execution (DSE), which is a method for determining the program's execution based on a particular input value.

In this study, DEL aims to integrate both static and dynamic symbolic execution into the lifting process itself. Its goal is to provide an enhanced intermediate representation of C++ programs by combining some of the above-mentioned static and dynamic approaches. Firstly, DEL statically disassembles the binary it aims to lift and then constructs a preliminary CFG using the standard approach for CFG reconstruction discussed above. DEL then translates all the assembly instructions generated by its disassembler into an LLVM IR module. Embedded inside this module is the preliminary CFG constructed from step one. DEL then performs a static symbolic execution in an effort of formulating each potential target of the indirect jumps into individual Z3 formulas. DEL then proceeds with dynamically symbolically executing the preliminary IR module to resolve the Z3 formulas generated for each indirect jump detected and finally output an enhanced IR module that can accurately represent the control flow of the program being analyzed.

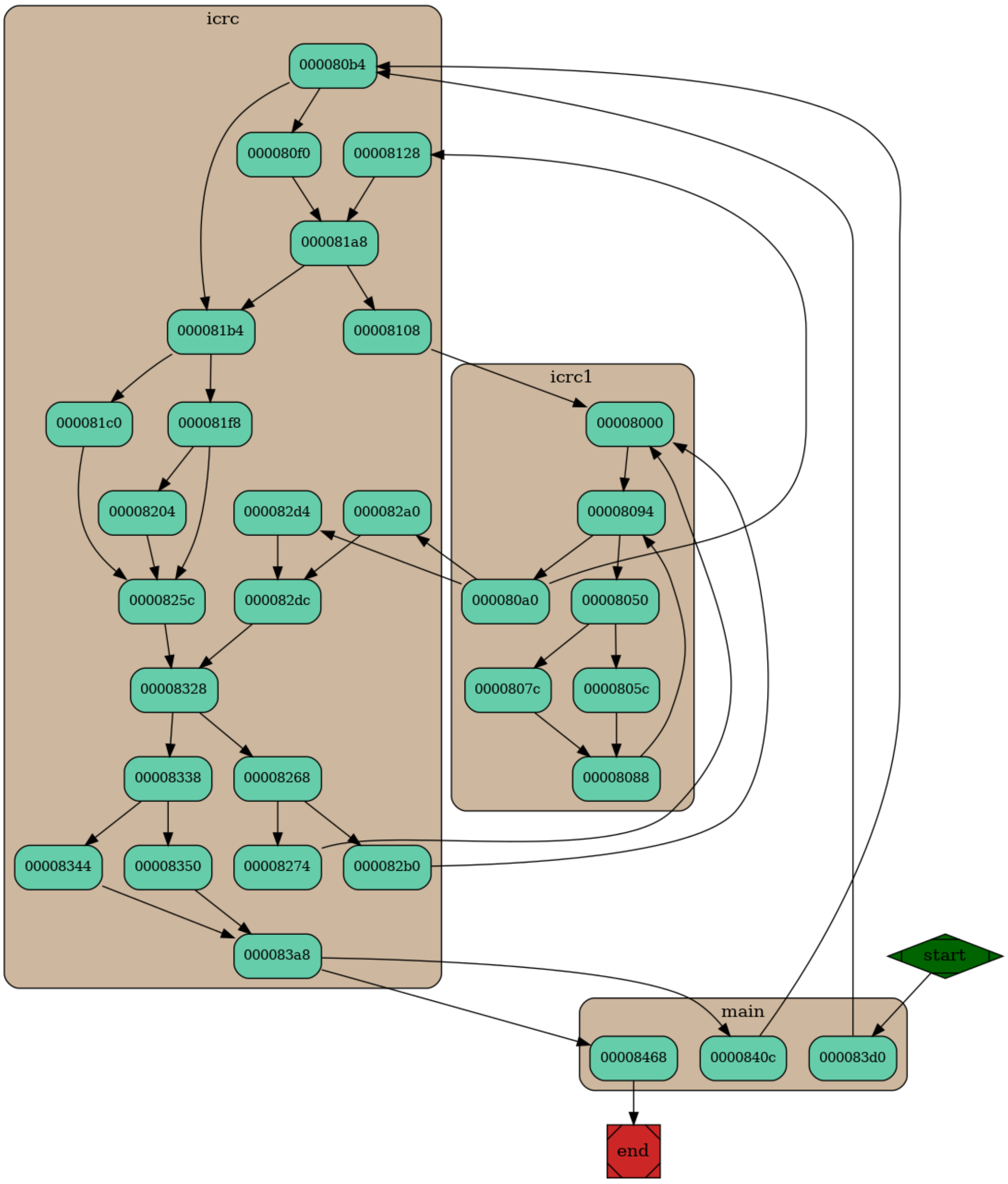
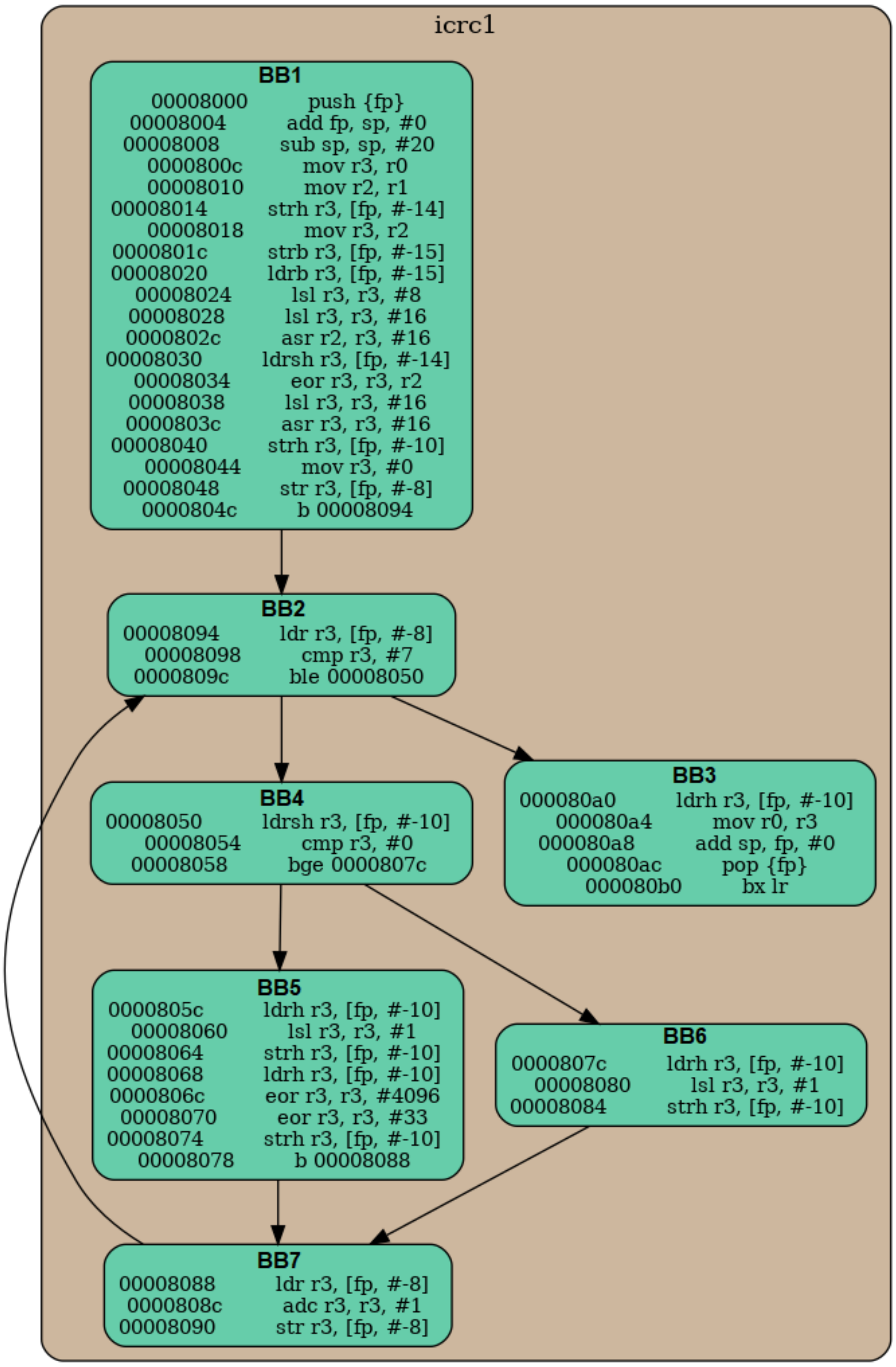


Figure 2.3: DEL 's reconstructed CFG for WCET crc benchmark [6].



```
BB1
00008000    push {fp}
00008004    add fp, sp, #0
00008008    sub sp, sp, #20
0000800c    mov r3, r0
00008010    mov r2, r1
00008014    strh r3, [fp, #-14]
00008018    mov r3, r2
0000801c    strb r3, [fp, #-15]
00008020    ldrb r3, [fp, #-15]
00008024    lsl r3, r3, #8
00008028    lsl r3, r3, #16
0000802c    asr r2, r3, #16
00008030    ldrsh r3, [fp, #-14]
00008034    eor r3, r3, r2
00008038    lsl r3, r3, #16
0000803c    asr r3, r3, #16
00008040    strh r3, [fp, #-10]
00008044    mov r3, #0
00008048    str r3, [fp, #-8]
0000804c    b 00008094
```

```
BB2
00008094    ldr r3, [fp, #-8]
00008098    cmp r3, #7
0000809c    ble 00008050
```

```
BB3
000080a0    ldrh r3, [fp, #-10]
000080a4    mov r0, r3
000080a8    add sp, fp, #0
000080ac    pop {fp}
000080b0    bx lr
```

```
BB4
00008050    ldrsh r3, [fp, #-10]
00008054    cmp r3, #0
00008058    bge 0000807c
```

```
BB5
0000805c    ldrh r3, [fp, #-10]
00008060    lsl r3, r3, #1
00008064    strh r3, [fp, #-10]
00008068    ldrh r3, [fp, #-10]
0000806c    eor r3, r3, #4096
00008070    eor r3, r3, #33
00008074    strh r3, [fp, #-10]
00008078    b 00008088
```

```
BB6
0000807c    ldrh r3, [fp, #-10]
00008080    lsl r3, r3, #1
00008084    strh r3, [fp, #-10]
```

```
BB7
00008088    ldr r3, [fp, #-8]
0000808c    adc r3, r3, #1
00008090    str r3, [fp, #-8]
```

Figure 2.4: DEL 's reconstructed CFG for the icrc1 function.

- **Translation**

  After re-constructing the CFG of a program, a lifting tool typically translates each assembly code instruction into its equivalent set of IR instructions.

  DEL implements its assembly to IR translator as a C++ API. DEL's translator takes an assembly code instruction as an argument and maps its opcode to the relevant API that translates it into a set of LLVM instructions. For this study, DEL's translator API was implemented for a subset of the ARMv7-M ISA [7] present in the assembly code of the Tasking Framework's case study, the Join fork example shown in Listing A.1. ARM assembly instructions that were not included in the assembly code of our case study have not been considered in the implementation of the translator API. Chapter 4 explains in more detail how the API was implemented.

  With translation marking the end of the binary lifting process, next, we explore the current limitations of binary lifting tools.

2.1.2 Today's challenges in binary lifting

Binary lifting has not gained much traction in practice due to its reliance on static disassembly. This approach fails to account for indirect control-flow targets, distinguish between data constants and code pointers, and recognize instruction and data byte boundaries [14].

[14] argued that transformation, analysis, and recompilation of binary code could be complicated without accurate representation at the high level. If the binary code is encoded or ciphered, the problem is exacerbated. [14] highlighted some prominent challenges when performing binary lifting and program transformations using purely static approaches. Here, we provide an overview of these challenges and promote our hybrid approach of lifting by highlighting why static approaches cannot be relied upon in the context of binary lifting.

- **Code vs data, and reference ambiguity:**

  Data and references inserted into programs by compilers are usually not labeled. Program analysis must deduce the relevant labels to segregate code from data and constants from references. In the general case, there is no clear-cut answer to such an issue, and modern analyzers use heuristics to estimate the right labeling scheme [82, 83, 89]. If, for example, a data value has the correct alignment and a valid binary code address, it can be deemed a code reference. However, value collisions are common [82], and many platforms do not require alignment.

  Generally speaking, analyzing how the processor interprets values from memory can aid a dynamic tool with precisely assigning labels.

- **Indirect control flow:**

  Based on the execution context, indirect control flow transfers (ICFTS) can pass control to multiple target locations. Such indirect calls take the form of function pointers in C code, and they are, in fact, more common in C++ code appearing as virtual functions. Additionally, position-independent code (PIC) and switch statements are frequently enacted in indirect branches. All direct branches turn into indirect branches in PIC, which append the offset of the binary/ library's memory mapping to the branch target.

  Based on the standard scenario, statically determining all potential targets of ICFTS is impossible [46]. However, when it comes to determining the possible targets of branching instructions that get their target address from jump tables, static techniques have proven to be proficient [39, 89]. Nevertheless, resolving indirect function calls and returns remains an issue. Although Wang et al. [83] claim that their technique can assist in dealing with ICFTs, their prototype Uroboros indeed does not [14]. Moreover, based on architecture, the rudimentary analytical techniques [39] employed in Rev.Ng [40] state they at best have achieved 90-95 percent jump target recovery.

As opposed to the approaches mentioned above, given enough input configurations (to cover as many execution paths as possible), the DSE of code at the intermediate level can effectively identify control flow targets. The DSE follows the execution path to any jump target, regardless of how the destination address is derived.

- **poorly-structured code:**

  Apart from optimization, manually produced assembly code is also used for debugging and disassembly prevention. Although the generated code is deterministic, excessive optimizations added by a compiler may lead to poorly defined instruction constructs [18]. Overlapping instructions remain a popular anti-disassembly strategy [86], but they can also be found in highly optimized libraries [18]. [14] mentioned that some compilers reduce selection control structures (e.g., switch/case) to jump tables and inline data. However, detecting function boundaries can become challenging with overlapping multi-entry functions, basic blocks, and tail calls.

  Typically, dynamic techniques avoid handling ill-formed code, as they are only concerned with instructions that the processor executes [14].

- **Obfuscation:**

  Binary lifting techniques will inevitably encounter binary files that have been actively modified to impede analysis. Even though various obfuscation strategies have been thoroughly published [34, 35, 79], they nevertheless pose major difficulties in actual use. Virtualizing obfuscators, for example, convert executable code in code segments to bytecode in data segments and insert a virtual machine in the program to elucidate the bytecode [17, 34]. The static code parts of a program covered by such an obfuscator give very little insight into the program's functionality. Moreover, control-flow flattening [35], obscure predicates [36] and aliasing [81] are some additional obfuscation approaches that can indeed pose problems. These modifications can be utilized to synthetically expand the complexity and size of the control-flow graph of a program to the point where performing an accurate IR analysis becomes very challenging. However, employing dynamic approaches midst of the lifting process can reverse all of these obfuscating processes by eliminating dead code and aliases that are not needed [14].

Now that we have discussed the most prevailing challenges faced by today's existing binary lifters, we move forward to review the state-of-the-art binary analysis and lifting tools.

2.2 State of the Art Analysis Tools

There have been many frameworks developed for program analysis. In most cases, these projects go beyond simply analyzing binaries to reverse engineer systems and firmware. There is currently no single tool capable of performing all the tasks required in the analysis process. Rather than choosing one, it is worthwhile to explore all alternatives. Tools such as these are primarily utilized for binary analysis, malware analysis, and reverse engineering. The purpose of this section is to provide a comprehensive overview of the most popular program analysis tools out there, emphasizing their strengths and shortcomings.

2.2.1 McSema

McSema is a static binary lifter that transforms executable binaries to LLVM IR. Analysts can use McSema to detect security vulnerabilities in binary programs, independently verify vendor source code, and write high-code-coverage application tests. Despite its strengths when employed in the static binary analysis, McSema does indeed have its shortcomings. When it comes to CFG reconstruction, McSema relies heavily on IDA Pro, where only directly called functions can be accurately identified. As a result, IDA Pro, in a way, hinders McSema's performance due to its inability to detect function pointers in real-world code correctly. Such a scenario is demonstrated in Listing 4 of the excerpt of decompress.c: libjpeg example from a case study by [13]. Here, the structure object "progress" provides a member field "progress monitor" that stores the address of a callback function at line 8, While at the same time, a second member "pass-limit" holds an integer indicating a loop bound at line 9, which turns out to be in a comparable value range as that of the address of the callback function. Altinay [13] reasons that the fact that IDA utilizes heuristics to determine integers with values in the executable section as code pointers will cause McSema's lifted binaries in this specific case to incorrectly modify the integer, which in turn alters the program's semantics. Likewise, if the code pointers are not identified correctly, callbacks could be poorly managed in this case.

Another challenge would be dealing with obfuscated code. McSema is designed for the translation of compiler-generated binaries and due to its reliance on the thoroughness of IDA pro's recovered CFG, using McSema in the accurate binary analysis of obfuscated code becomes infeasible. After all, IDA pro's recovered CFG will not always accurately capture the program's semantics, especially if code encryption takes place.

```
1   void callback_func(j_common - ptr cinfo) {
2           printf("");
3   }
4   int main(int arge, char** argv) {
5           struct jpeg - decompress - struct info; //jpeg info
6           struct jpeg - progress - mgr progress;
7           // After some initialization code
8           progress.progress monitor = callback_func;
9           progress.pass_limit = 0x8048860;
10          progress.pass - counter = 0L;
11          info.progress = &progress;
12          jpeg - start_decompress(&info);
13          char* data = (char*)malloc(dataSize);
14          readData(info, data);
15  }
```

Listing 4: Excerpt of decompress.c: libjpeg example in C [8].

2.2.2 BinRec

BinRec uses dynamic analysis to lift binary code to LLVM IR, where complicated transformations can be applied, then lowers it back to machine code, resulting in a recovered binary [14].

Binrec's primary purpose is to retrieve code that is difficult to analyze statically. Even though their use of dynamic analysis eliminates this obstacle, it also introduces the issue of covering code that is not used when lifting. While dynamically lifting a program from a single trace, the user is only presented with one of the multiple alternative code pathways. As a result, the recovered binary only works for paths with all of the control flow edges detected during lifting.

2.2.3 BAP

BAP is an open-source platform for performing binary code verification and analysis. One of BAP's flaws is that its lifting mechanism assumes it will be directed to an aligned sequence of instructions. As a result, the user must determine code locations. Although this can be accomplished by using a recursive descent analysis [26], still such analysis technique is once more ineffectual at resolving indirect control flow targets. Consequently, employing BAP in analyses where indirect jumps must be resolved becomes exceedingly challenging. Moreover, BAP uses IR instructions that are not explicit, which makes the prospect of the DSE of its lifted IR challenging and hence restricts the tool's ability to perform control- and data-flow analyses based on the DSE [10].

2.2.4 REV.NG

REV.NG is a binary analysis framework that works with a variety of architectures and is based on QEMU [24]and LLVM. When it comes to CFG recovery, REV.NG largely relies on the Simple Expression Tracker (SET) and Offset Shifted Range Analysis (OSRA) [39].

SET is a technique for extracting jump targets from translated code. It recognizes all store instructions and keeps track of how the value being stored is calculated successively. The analysis continues as long as the operations that make up the expression rely only on one non-constant operand. In actuality, the purpose of SET is to gather the destination addresses of direct and indirect jumps that realize the target address in many instructions. This method can be quite useful for finding the most basic jump targets embedded in the code. It can retrieve the destinations of direct jumps, indirect jumps with a fixed destination manifested in a register, and all call instruction return addresses [39]. However useful it may be, SET still fails to retrieve jump targets resulting from switch statements in which the jump destination address is dependent on a non-constant operand: the result of the switch statement's expression evaluation [39].

OSRA however, is a specific data flow analysis whose purpose is to illustrate how the target address of an indirect jump caused by a switch statement is calculated. It achieves this by formulating each Static Single Assignment (SSA) value of the relevant IR instruction as an expression that eventually highlights all the operations involved in the target address computation. It is primarily implemented to recover jump targets for a specific variety of switch statements. OSRA however, is not without flaws. In general, OSRA is not capable of reading data from memory segments contained in binary code and only supports a limited set of binary operations [39].

In REV.NG, both SET and OSRA collaborate while utilizing an SSA intermediate representation and cycle several times until they yield no further information that could be used in the CFG reconstruction process. Indeed, these analyses could be utilized as a prelude towards obtaining a basic CFG. However, the recovered CFGs' accuracy tends to be a problem [39]. A source of such inaccuracy could be, for example, an aggressively optimized nested switch. Where REV.NG could not determine the size of the jump tables utilized by the inner switch statement in specific functions that used nested switch statements. Another source of inaccuracy is the jump table addresses spilled on the stack. Because the initial address of a jump table may be utilized several times within the function, GCC can spill it on the stack in the function prologue in some cases [40]. Furthermore, due to it not having a dynamic component that involves the actual execution of the program being analyzed, REV.NG has no information about function calls, making tracking stack values across function calls exceedingly challenging [40].

2.2.5 Angr

Angr is a binary analysis framework that combines many current cutting-edge binary analysis algorithms. It provides a reliable foundation for many different analyses, both static and dynamic. When analyzing binaries, this binary analysis tool particularly introduces a dynamic component. It employs a technique known as under-constrained symbolic execution UCSE [66], which rather than executing the full program, executes an arbitrary function within the program that is being analyzed directly.

The fundamental goal of Angrs' usage of UCSE is to prove the correctness of Real Code. Instead of starting with main, UCSE starts with an arbitrary function chosen by the user. When the function exits all possible execution paths within it, the intended check of the function's correctness is complete (a real case example would be checking that the introduction of a patch does not cause a crash). However, directly invoking functions within a program poses a unique complication where a program's crash points detected by UCSE are not reproducible. This problem happens because each function is executed independently while at the same time the analysis cannot reason about how to get to a certain function. Since each function is generated without prior knowledge of its arguments and the global variables with which it is called in actual executions, the analysis is rendered inaccurate [66]. On the other hand, performing the DSE of the whole program involves acquiring input values from outside sources. [66] argued that in most circumstances, valid software should reject erroneous external inputs rather than crashing. Individual functions, however, frequently have preconditions forced on their inputs. A function may, for example, require non-null pointer arguments. Moreover, because UCSE executes functions without prompting the user for their preconditions, the inputs it takes into account may be an over-approximation of the permissible values the function can take [66]. As a result, UCSE symbolic inputs are labeled as under-constrained, indicating that they lack specific constraints. While this approach allows inaccessible code to be thoroughly examined, the lack of preconditions may result in unfounded errors being reported during execution [66].

Strictly speaking, the DSE of the whole program essentially investigates every execution path during a program's execution in a bit-precise manner and considers all possible input values. It explores a much larger number of paths than conventional testing, hence guaranteeing a high program coverage and making it even possible to check whether a particular combination of inputs could result in the program failure.

Chapter 3

Tasking Framework

Nowadays, running sophisticated algorithms and complex processing data pose a formidable challenge for space missions, which is why managing resources is of great importance for the success of such missions. Rather than using trajectory control advance algorithms, which necessitated the use of more power [10], missions like Rosetta or the Mars rover landing were built on a list of directives to regulate landing and maneuvering in order to conserve energy. The estimator and observer control modules were developed in a fixed fashion (order and time) during the creation of the TET-1 satellite mission (Technology demonstration) and the Bi-spectral Infrared Detection (BIRD) missions. The module's full calculation duration was the time it took to wait for the sensors' data plus an extra delay to guarantee a full data delivery prior to the start of the calculation. Because of an overestimation of the timing delay, this model causes a timing violation throughout the control cycle. This problem was not found until after the launch, when a timing failure in another bus application caused the computed tasks to be reordered, resulting in erroneous data and the malfunctioning of the orbit control systems [60].

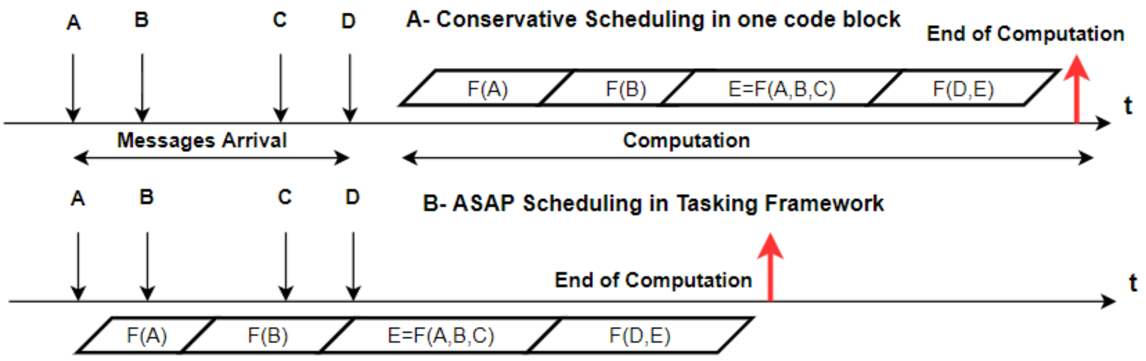


Figure 3.1: Scheduling in the Tasking Framework adapted from [10].

DLR's onboard computer-next-generation project began to establish reliable processors and network nodes with an operating system that would guarantee satellite timing behavior.

This design must also account for multi-core and distributed systems' timing behavior. The Tasking Framework constituted the foundation of this concept. The framework was created primarily to increase the performance of attitude control systems by breaking the computational data from the sensor into small portions, each of which is referred to as a task, and then scheduling them according to their readiness [10]. The Tasking Framework was created using the inversion of control design paradigm, which is commonly utilized in creating lightweight frameworks. The Autonomous Terrain-based Optical Navigation (ATON) project [76] uses the framework, which is a technology to navigate a lunar landing scenario that uses multiple image processing techniques [60]. The framework's most essential feature is its ability to alter the time behavior of the tasks being processed [10]. Figure 3.1 from [10] shows the impact of using the ASAP scheduling policy on overall response time, as opposed to conventional scheduling, which starts calculation at a predetermined time in the computation cycle.

3.1 Task-Channel Model

The task-channel paradigm presented in [43] was used to create the Tasking Framework. The idea is to create a barrier between functionality and data. [10] described a **task** as a ”stateless executable program” with memory and I/O ports in this model, whereas a **channel** is a message queue that links the output port of one task to the input port of another. The channel in the Tasking Framework is a data container that the task object may handle. It works as an interface that serves as a link between tasks and connects software outputs and inputs, as demonstrated in Figure 3.2. The use of a task-channel architecture improves the reusability of code [10]. It is conducive in systems that are distributed in which some components of the software must be moved between processing nodes [10]. The Tasking Framework was created with data-flow-oriented applications in mind. The operation of a system is understood by looking at how data flows through it. [10] argued that data-flow-focused methodologies require that the input data of the system be determined and processed to produce the appropriate outputs. The program is constructed as a sequence of successive operations that occur in a specific order using this method. The Tasking Framework employs this design paradigm to introduce an interface that is structural and not reliant on the availability of data but rather on its flow. All APIs, except for the **Execution class APIs**, demonstrate a high level of generalization, as they are no longer constrained by the presence of input data and the current task [10]. The framework can be compared to operating systems in that it controls the entire process in a deterministic, generic, and abstract manner [45].

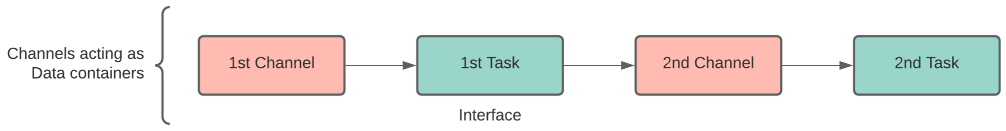


Figure 3.2: Task Channel Model adapted from [10].

3.2 Execution Model

[10] stated that when all task inputs are active, a task instance τ is launched in the Tasking Framework. For example, Task **A** will be executed in Figure 3.3 when input 1 is active immediately after receiving Msg.A from sensor A. Marking one of the task's inputs as final is another

approach to triggering it right away. If this input is enabled, the task will execute regardless of the state of the other inputs. in Figure 3.3 Task **E**, for example, will be triggered when the task event (Timer) gives the input 0, which is marked as final, regardless of the state of the other inputs. **C** will be triggered immediately after that.

The Tasking Framework's schematic diagram is depicted in Figure 3.4. When a message from a sensor is received, the main execution thread uses the channel class's **push()** method to alert the related inputs. In the scenario where all task inputs have been set up, the Tasking Framework will instantly inform a thread to run the waiting instance of this task by invoking **perform()**. The framework's scheduler kicks off the task right away. The job will begin as soon as a free resource, such as a CPU core, becomes available; or else, the task will be queued [45, 60].

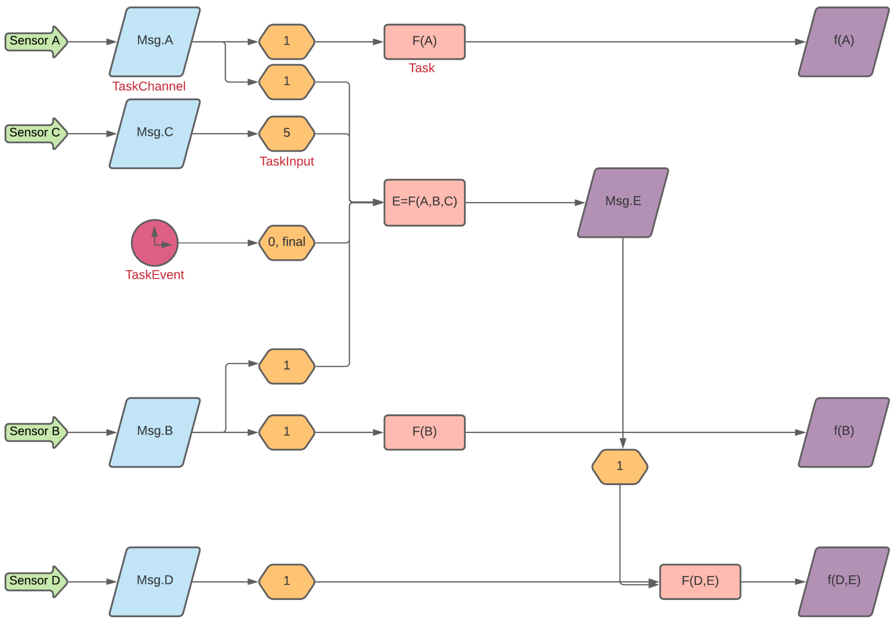


Figure 3.3: BIRD - AOCS and the Tasking Framework Components adapted from [10].

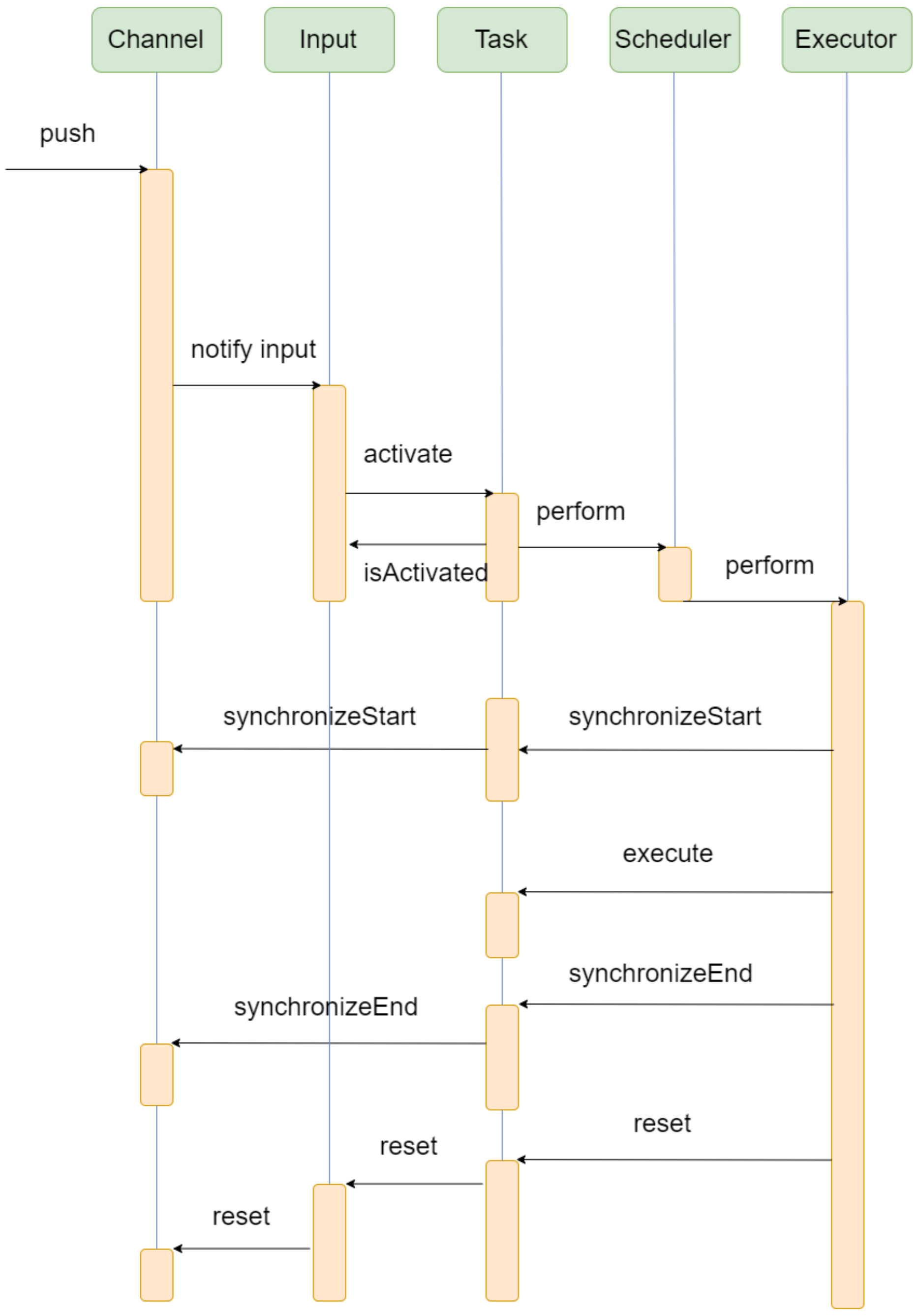


Figure 3.4: The Tasking Framework's sequence diagram adapted from [10].

3.3 Tasking Framework in use

Several DLR initiatives have made use of the Tasking Framework. This section briefly highlights 3 projects mentioned in [10] where the Tasking Framework plays a key role. The Tasking Framework was utilized to apply the functional tasks and link them by means of channels in ATON. In this model, channels are data-containers that store data, while events are used to trigger the different system components routinely. 4 threads were employed by the developers to run the software on the prototype flying computer.

The Attitude and Orbit Control System (AOCS) was developed using the Tasking Framework in the Euglena Combined Regenerative Organic Food Production In Space project (Eu:CROPIS) [59].

Another DLR project presents and evaluates a novel onboard computing architecture consisting of re-configurable interlinked commercial off-the-shelf processors coupled in a single distributed system. The project is called Scalable On-Board Computing for Space Avionics (ScOSA) [77]. The Tasking Framework is a component of the middle-ware and the core API for developing the ScOSA-based application. On-board Data Analysis and Real-time Information System (ODARIS) [68], and Rendezvous Navigation [67] are two examples of applications that will be implemented utilizing the Tasking Framework to run on ScOSA.

3.4 Tasking Framework as a C++ Library

The Tasking Framework is created by the German Aerospace Center's Institute for Software Technology (DLR). The framework is a platform for event-driven execution for onboard software systems that run in real-time. It enables tasks to be implemented as graphs with capricious patterns of activation [10]. It is developed in C++ based on the event-driven programming approach and is capable of multi-threading programming [45]. Even though C++ is not often used to construct aerospace applications, it was employed in the development of this framework because of the following factors as discussed by [10]:

- To begin with, the language is modular since it employs object-oriented programming. Class implementation in C can also be achieved using struct, but due to constructors and deconstructors, C++ surpasses it. These constructors ensure that objects are always properly instantiated, while the destructors ensure that they are permanently deleted when they are no longer needed, guarding against bugs or leaks.
- Thanks to C++'s templates, it is possible to program abstractly and generically. It is possible to transform a template into a macro that creates a unique data type that is fully functional. These templates can help adapt algorithms efficiently for various types with reduced resource demand.
- Another reason to use C++ is type safety. In type safety, the compiler makes sure all variables are valid and that there are no mix-ups of data types involved in the relevant operations. In contrast to C++, the C function memcpy can copy double values into an array of char values, eventually generating meaningless data.

- It is not easy to create distinctive descriptive naming in large projects; hence prefixes have traditionally been appended to the names. As a result, names become long and difficult to read. The simplest way to fix that issue is to use namespaces in C++. Multiple occurrences of the same name in various contexts are permitted by such namespaces, which are determined later during compilation. This C++ feature ensures that the name is used in many informative and distinctive ways.
- As opposed to C, which allocates and frees memory by invoking malloc() and free(), C++ utilizes new and delete, adding constructors and destructors to prevent memory leaks and other errors.
- C++ provides novel features such as inheritance, operator overloading, and virtual functions that are not present in C.
- In C++, references and smart pointers are far more secure than regular C pointers because they prevent pointers from referencing NULL or being uninitialized.

3.5 Tasking Framework and its relevance to static analysis

[10] argued the relevance of the Tasking Framework in static analysis. Their argument is based on a handful of points, which we highlight in the following paragraphs:

Strictly speaking, the application is treated as a directed graph in data-flow programming, where the tasks handle the data and then pipe-lined to the following tasks in the sequence. In this case, instructions' execution does not depend on the completion of previous tasks; instead, once the data becomes available, they can be executed, which is known as event-driven execution.

Abstract classes are provided by the Tasking Framework. The classes can be used to create applications organized as a directed network of tasks and channels. As a result, the API loops that link the channels and tasks are constrained. Put another way; the API is not reliant on any data provided at run-time.

The channels in Tasking Framework serve as data containers. The Tasking Framework's technique for exchanging data between tasks and inputs is preordained and independent of the data type or value.

3.6 Tasking Framework in this thesis

Typically, strict verification and validations are required before the Tasking Framework could be employed in hard real-time safety-critical applications.

A static analysis is the only way to assess the system's real-time capabilities and demonstrate its ability to meet deadlines. At all execution stages, a static analysis computes a general overestimate for all sets of architectural states. It guarantees that a specific condition will not be encountered at a specific execution point. This safety attribute permits a safe WCET upper bound to be established. The framework's provision of WCET is the initial step toward developing safety-critical applications. This thesis aims to optimize the lifting of C++ applications' binaries for effective loop bounding and WCET analysis at the IR level. We evaluate our novel approach of combining both static and dynamic symbolic execution while lifting binaries of the Tasking Framework case study, the Join fork example in Listing A.1. The Join fork example is an illustrative example for an onboard data processing application inspired by the ATON project.

Similar to [10], The Tasking Framework makes a suitable candidate test case for our thesis for two main reasons. Firstly, the Tasking Framework is designed to help create data-handling applications. It proposes a novel satellite onboard data modeling and computes scheduling approach. The framework deviates from conventional scheduling, which requires all compute processes to wait to receive a single message. It does, however, introduce an ASAP schedule, which more effectively utilizes the time available and improves the worst-case response time of the entire onboard system. For this reason, conducting a WCET analysis at the IR level is critical for computing end-to-end real-time assurances on the envisaged satellite onboard system scheduling model.

The second reason for using the Tasking Framework as our case study is that it is a real-world application that investigates whether WCET can be conducted on model-based C++ code. The framework provides most C++ structures employed in embedded safety-critical applications [10]. Abstract classes and virtual methods are covered. As a result, it is a viable candidate for WCET analysis at the IR level.

Chapter 4

DEL Lifter

This chapter introduces DEL, our new lifting tool that combines static and dynamic symbolic execution while lifting into LLVM IR. We start by presenting the motives behind the techniques we adopted, then discuss some definitions, followed by presenting our concept and implementation.

4.1 Motivation

There are three main reasons behind the techniques we adopted while implementing our lifting tool, DEL:

- We chose to integrate symbolic execution into the lifting process as such a method gives a clear insight into the program's workflow. Dynamic and static analyses are coupled together in this procedure. It generalizes a valid and exact program trace to forecast how the program will behave when presented with a particular input [10]. Through the proposed technique, we resolve indirect control flow targets and check the correctness of the lifting process.
- DEL's intermediate representation language DSEIR uses a subset of LLVM instructions due to its popularity and its support of various forms of analyses as natural loop information, memory dependence analysis, and many more. Such analyses could be very useful when applied to data flow space applications.
- To generate an enhanced IR of data flow space applications, DEL implements its memory and register models. It makes sure the effect of condition flags checking and updating functionalities of assembly instructions are captured in the lifted IR module.

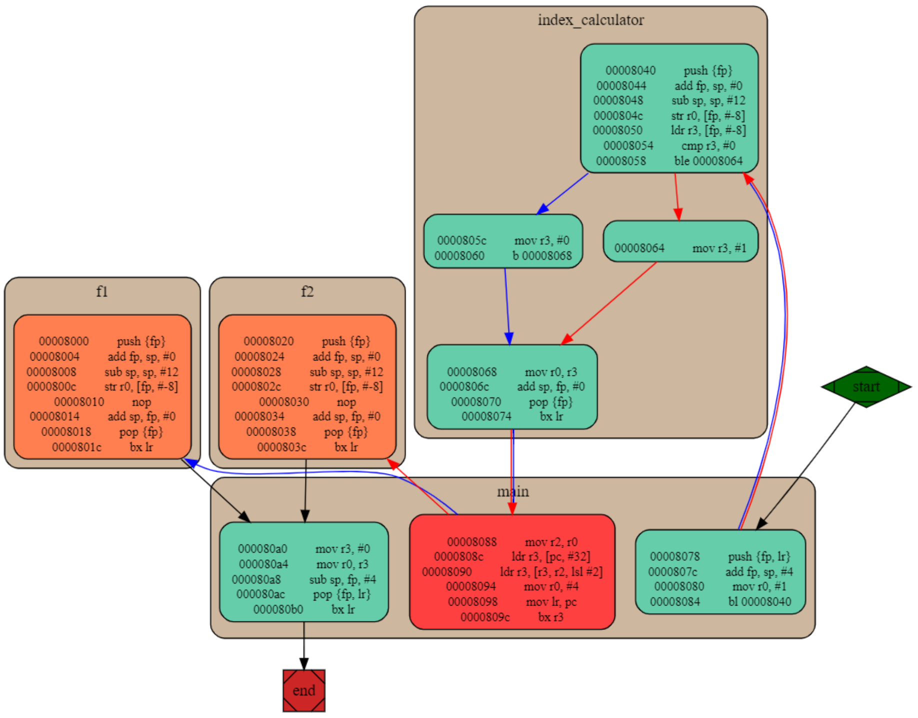
Let us consider the example program in Listing 5. Here, different inputs to the program could result in different potential-jump targets for a single indirect jump instruction. Based on the input argument of the index_calculator function, the program ends up either invoking function f1 or function f2 in line 18 of Listing 5. Listing 6 shows a snippet of the assembly code of the program. Figure 4.1 shows DEL's re-constructed control flow graph (CFG) for the program. The figure illustrates the two possible paths that could be taken from the start entry point of the program to the basic block with the indirect jump in line 31 of Listing 6. The two paths define two potential-jump target addresses for the indirect jump, and those are the addresses of functions f1 and f2 $(00008000, 00008020)$. The assembly instruction in line 28 of Listing 6, **ldr r3, [r3, r2, lsl 2]** defines the calculation of each possible jump target address. The computation takes the form of Equation 4.1 where each jump target address (J) is computed by adding the jump table's base address (A) with a variable offset (X) multiplied by the memory byte size in bits (k).

$$J = A + kX \tag{4.1}$$

In this scenario, the base address of the jump table is stored in $r3$, and the offset is stored in $r2$. The **lsl 2** resembles a multiply operation by the constant 4. This factor represents the size of a memory byte in bits for the used architecture at the time of disassembling the program.

As the program invokes the index calculator function, the input argument is stored in $r0$, which is then conveyed to $r3$ through the store and load operations in lines 5 and 6 of Listing 6, respectively. The compare instruction in line 7 of Listing 6, **cmp r3, #0**, checks if the value held in $r3$ is equal to zero or not. This value reflects the argument passed to the index_calculator function. Depending on the result of the comparison, the program can branch to basic block L6 setting $r3$ to 1 in line 12 of Listing 6 or Alternatively, execute the instruction in line 9 of Listing 6 and set $r3$ to 0. This results in two potentially different offset calculations by the instruction in line 28, and hence a different jump target address for the indirect jump in line 31.

This example program was intentional to highlight how different inputs to a program can result in different potential-jump target addresses for a single indirect branching instruction. For this purpose, our approach firstly performs a static symbolic execution (SSE) to formulate each potential-jump target address of each indirect jump in the program as a Z3 expression. Secondly, we perform a dynamic symbolic execution (DSE) using the Z3 solver from Microsoft Research [37] to resolve the Z3 expressions of the indirect jump target addresses to their concrete values.



```
00008040    push {fp}
00008044    add fp, sp, #0
00008048    sub sp, sp, #12
0000804c    str r0, [fp, #-8]
00008050    ldr r3, [fp, #-8]
00008054    cmp r3, #0
00008058    ble 00008064
```

```
0000805c    mov r3, #0
00008060    b 00008068
```

```
00008064    mov r3, #1
```

```
00008068    mov r0, r3
0000806c    add sp, fp, #0
00008070    pop {fp}
00008074    bx lr
```

```
00008000    push {fp}
00008004    add fp, sp, #0
00008008    sub sp, sp, #12
0000800c    str r0, [fp, #-8]
00008010    nop
00008014    add sp, fp, #0
00008018    pop {fp}
0000801c    bx lr
```

```
00008020    push {fp}
00008024    add fp, sp, #0
00008028    sub sp, sp, #12
0000802c    str r0, [fp, #-8]
00008030    nop
00008034    add sp, fp, #0
00008038    pop {fp}
0000803c    bx lr
```

```
000080a0    mov r3, #0
000080a4    mov r0, r3
000080a8    sub sp, fp, #4
000080ac    pop {fp, lr}
000080b0    bx lr
```

```
00008088    mov r2, r0
0000808c    ldr r3, [pc, #32]
00008090    ldr r3, [r3, r2, lsl #2]
00008094    mov r0, #4
00008098    mov lr, pc
0000809c    bx r3
```

```
00008078    push {fp, lr}
0000807c    add fp, sp, #4
00008080    mov r0, #1
00008084    bl 00008040
```

Figure 4.1: Two potential paths from the main entry point till the basic block of the indirect jump (00008088).

```c
// Type your code here, or load an example.
typedef int (*function_pointer) (int);
int f1(int a) {
    return (a * 2);
}
int f2(int a) {
    return (a * 7);
}
int index_calculator(int n) {
    n = n * 1;
    if (n > 0) {
        return 0;
    }
    return 1;
}
function_pointer jumpTable[] = { f1,f2 };
int main() {
    jumpTable[index_calculator(1)](4);
    return 0;
}
```

Listing 5: Source code of the example program.

```asm
index_calculator(int):
        str     fp, [sp, -4]!
        add     fp, sp, #0
        sub     sp, sp, #12
        str     r0, [fp, -8]
        ldr     r3, [fp, -8]
        cmp     r3, #0
        ble     .L6
        mov     r3, #0
        b       .L7
.L6:
        mov     r3, #1
.L7:
        mov     r0, r3
        add     sp, fp, #0
        ldr     fp, [sp], #4
        bx      lr
jumpTable:
        .word   f1(int)
        .word   f2(int)
main:
        push    {fp, lr}
        add     fp, sp, #4
        mov     r0, #1
        bl      index_calculator(int)
        mov     r2, r0
        ldr     r3, .L10
        ldr     r3, [r3, r2, lsl 2]
        mov     r0, #4
        mov     lr, pc
        bx      r3
        mov     r3, #0
        mov     r0, r3
        sub     sp, fp, #4
        pop     {fp, lr}
        bx      lr
.L10:
        .word   jumpTable
```

Listing 6: Assembly code of the example program.

4.2 Preliminaries

This section covers some of the terminology and definitions that the reader will encounter in this chapter.

- **Assembly code:**

  Microprocessors and other programmable devices use assembly code as a low-level programming language. Assembly code is a symbol for the machine code needed to program a specific CPU architecture.

- **Instruction:**

  A computer instruction code is a set of bits that tells the computer how to complete a specific task. The operation code is a group of bits in an instruction that defines the operation to be done, such as addition, subtraction, shift, complement, and so on.

- **Basic block:**

  A basic block is a sequence of instructions without branches going in except at the entry and without branches going out except at the exit.

- **Control flow graph (CFG):**

  The CFG of a program is a graphical representation of all the possible paths a program can take during execution.

- **Path**:

  A path in the CFG is represented by a series of basic blocks B_1,B_2,...B_k such that k $>$ 0 and for all 1 $\leq$ i $<$ k there is a transition from B_i to B_{i+1} [30].

- **Jump table**:

  A jump table is an array of pointers to functions. Functions are called through indexing into the array. The first address in the jump table is called the base address. The functions' addresses are stored in the table as an offset added to the base address.

- **Symbolic execution:**

  It is a method of conceptually executing a program. The execution encompasses more than one input of the program that follows a common execution path. During execution, these inputs are interpreted symbolically, and expressions based on these symbols are returned [15]. There are two distinct types of symbolic execution:

  – **Static symbolic execution (SSE):**

    This technique evaluates a sequential program P's viability by examining its control flow by assigning symbols representing the program's inputs. It is intended to execute the instructions ordinarily, only now the values are formulated as symbolic expressions of the input symbols [10]. This corresponds to an expression Φ(P) that defines the set of inputs i ∈ I used to assess the feasibility of the path [10]. As a result of conditional branching, the execution is divided to find a set of inputs i ∈ I that fulfill each path separately [10]. The execution of each instruction along each path is validated against the branching condition. Upon failure, Φ(P) contains an empty value, meaning that no path could be followed [48].

– **Dynamic symbolic execution (DSE):**
  This method inspects program P by executing it with the input of **i** to produce a viable path for the execution process [10]. Whenever appropriate, it substitutes symbol expressions with the true values from P(i) execution. DSE exhibits real-time execution capabilities combined with symbolic expressions [10]. As a consequence of the program's symbolic execution, while making use of actual concrete values, the symbolic expressions can be greatly simplified [48].

For understanding symbolic execution, we will consider the example shown in Listing 7 adapted from [10] and its corresponding assembly in Listing 8. Here, the program uses the result computed value from the performCalculations() function as an input **i** and stores it in r0. A multiplication step, followed by a conditional if evaluation, follows. Symbolic execution reads a symbolic value (β) and assigns it to r0. Following that, the multiplication operation is carried out, which will set $\beta << 1$ to r0. Then, at the **cmp** instruction, β is compared with 9. Now, β can be assigned any random value, and symbolic execution continues in either direction. Every path is designated a set of constraints and a program state. Here, the path constraint is $\beta * 4 > 9$ for Branch2 and $\beta * 4 <= 9$ for the Branch1. It is possible that the two paths are symbolically executed separately. Whenever the paths finish executing, symbolic execution calculates an exact value for β by resolving each path's cumulative constraints. In order to use DSE on this program, actual values will be used to substitute the symbol expressions β.

```cpp
int perfromCalculations() {
    //return a computation value
}
int fail() { return 0; }
int success() { return 1; }
int main() {
    int x, z;
    x = perfromCalculations();
    z = x * 4;
    if (z <= 9) {
        return fail();
    }
    else {
        return success();
    }
}
```

Listing 7: Symbolic execution example in C++ adapted from [10].

```asm
        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     Branch2
        b       Branch1
 Branch1:
        bl      fail()
        str     r0, [r11, -4]
        b       exit
 Branch2:
        bl      sucess()
        str     r0, [r11, -4]
        b       exit
```

Listing 8: Symbolic execution example in ARMv7-M assembly.

- **Satisfiability modulo theories (SMT):**

  SMT deals with the determination of an expression's satisfiability with regard to a combination of first-order background(decidable) theories. Real-number theory, the integer theory, and other data structure theories such as array and bit-vector theory are examples of SMT theories. Programming problems can be formalized and constrained with SMT. SMT solvers are primarily used for creating test cases and determining model bounds [37].

  In many ways, SMT is a variant of the Boolean Satisfiability problem (SAT). By the Boolean Satisfiability problem, it is assessed if it is feasible to provide values to a set of variables of an expression in a way that will result in it evaluating to true. For instance, the expression in Equation 4.2 from [10] is satisfied if p is set to true and q is assigned to false, in which case the trinomial (expression) will result in true.

$$(p \vee q) \wedge (\neg p \vee \neg q) \wedge (p \vee \neg q) \tag{4.2}$$

Boolean Logic is used to solve Boolean satisfiability problems. SMT solvers, meanwhile, use first-order theories. In first-order theories, statements are broken down into relations (e.g., predicate:assert (a<b)), component parts (e.g., functions and variables), quantifiers (e.g., ∀) and connectives (e.g., ||) [10]. According to first-order linear inequality theory, the expression in Equation 4.3 adapted from [10] is satisfiable if variables x, y, z, and w are set to 30, 27, 32, and 21, respectively.

$$(2 * x > y + z) \wedge (2 * y > z + w) \wedge (2 * z > 3w) \wedge (3 * w > x + z) \tag{4.3}$$

The purpose of SMT is to evaluate the satisfiability of the expression β for a theory **T**. The expression is characterized by signatures containing a set of function symbols and a set of conditional symbols. Such a problem can be polynomial or undecidable depending on β and **T** [10]. Examples for **T** from [25] are:

- Real Arithmetic Theory with $\Sigma = \{+, x, \leq\}$ includes all isomorphic structures to real numbers with $+$, x and $\leq$ functionalities [10].
- Array Theory with $\Sigma = \{select, store\}$ includes all the isomorphic structures to the memory read (select) and memory write (store) functionalities [10].

- **Array theory:**

  [64] first introduced the arrays theory which has the signature $\Sigma = \{select, store, =\}$. When the **select(a,i)** function is called, it returns the element i of the array a, while the **store(a, i, e)** function returns the array a with the element e in place of the index i. Array elements are only subject to the = predicate if they follow the principles of array theory [27]:

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

- **Bit-vector theory:**

  A bit-vector is an array that stores data in a close-packed manner in one vector unit. It is characterized by its width, which represents the number of bits of the vector. The bit-vector theory problem seeks to determine whether it is feasible to ascribe values to the bit-vector in an expression such that the expression evaluates to true. This technique is useful for simulating bit-level operations on the hardware level. The bit-vector theory handles bit-wise operations as ∨, ∧, ¬, <<, >>, etc [10]. An example of a summation operation utilizing bit-vectors is shown in Figure 4.2 in which the summation of 160 and 230 gives a result of 6 because of an overflow. [29] argued that as the theory handles an array of bits, the expression in Equation 4.4 from [10] that applies to integers does not apply to bit vectors since there's a possibility that an overflow can happen. In this thesis, bit-vector theory is used to determine the SSA expression's satisfiability similar to the work done by [10].

$$x - z > 0 \implies x > z \tag{4.4}$$

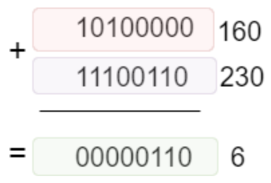


Figure 4.2: Bit-vector addition with overflow example.

- **Z3 SMT Solver:**

  Microsoft Research developed the Z3 SMT solver. It is intended to be used for software analysis problems and verification. Z3 offers an SAT solver, a satellite solver that supports array and arithmetic theories, and a core theory solver that handles functions, [37]. Throughout this thesis, we utilized the C++ API provided by Z3.

- **Static single assignment (SSA):**

  As defined by compiler theory, a static single assignment (SSA) is a distinctive property of an IR, meaning that a variable can only be assigned once, and its definition must precede its use. With SSA, compiler optimizations are significantly streamlined and enhanced [19]. As an example, in Listing 9 adapted from [10], the value of **a** in the group (1) is determined by the instruction in line 2 and the first instruction in line 1 is unnecessary. In order to identify cases like this, a reach definition analysis must be conducted [10]. In SSA instructions group (2), on the other hand, it is readily apparent that **b1** is meaningless.

```
b := 7              b1 := 7
b := 12             b2 := 12
a := b              a1 := b2
 (1)                  (2)
```

Listing 9: SSA instructions adapted from [10].

Control flow merges provide an additional ϕ function when an SSA instruction is coming from more than one path, which implies that there are instructions that may acquire different values depending on which path they fall on. Listing 10 and Figure 4.3 from [10] illustrate how merging of the control-flow works in SSA. The value of **b** in Figure 4.3 has two interpretations, either b1 or b2 depending on the execution path. According to the control flow, **b3** can be set to **b1** or **b2** by the ϕ function.

```
if ( condition )
  b := 5
else
  b := 36
a := b
```

Listing 10: CFG merging adapted from [10].

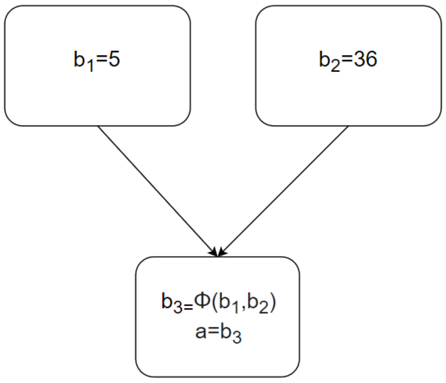


Figure 4.3: Φ Function adapted from [10].

- **Memory model:**
  DEL implements a memory model based on C++ map object theory. Data inside the memory model is formulated as Z3 bit-vector expressions. Thus the memory model object is defined as a pair, a memory hex string address as a key, and a data bit-vector as a value. The state of the memory model is updated with the execution of each instruction in the set of IR objects equivalent to the load/store (Main memory) and the push/pop assembly instructions (Stack). Before DEL is run, the memory model is populated with the initial values of all the program's data variables in the provided input file. This chapter refers to DEL's memory model by the symbol μ.
- **Register model:**
  To facilitate the DSE of IR instructions. DEL implements a register model. The register model is constructed as a C++ map object with the registers' names as the map's keys. The map's values are expressed as Z3 bit-vectors of the data stored inside the registers. The size of the bit-vectors matches the target architecture. During the dynamic run of DEL, for each register in the register model, the bit-vector value of the data stored inside the register is updated with the execution of each SSA IR instruction that sets that register. This chapter refers to DEL's register model by the symbol ρ.
- **Condition flags:**
  Many architectures, including ARM, provide conditional execution by storing state information about previous operations in a set of flags. An s suffix can be appended to many ARM assembly instructions to update the condition flags based on the result of the instruction's operation. The additional information is held in four condition flag bits in the Application Processor Status Register (APSR) or the Current Processor Status Register (CPSR). In the flag bits, fundamental

information such as whether or not the result of an operation was negative is specified. Those bits can be used in different combinations to recognize higher-level relationships, such as "less than" and similar concepts.
DEL handles the condition flags in a similar manner to the work done by [71]. To examine the side effects of instructions on the condition flags, DEL decides to expose such effects. For each assembly instruction updating the flags(i.e., the opcode ending with the optional s suffix), the corresponding effect is represented by a sequence of IR instructions. For example, DEL's API translates each **adds** assembly instruction object into a sequence of IR instructions which model not only the effects of the add operation on its operands but also the effects of the result of the operation on the condition flags. Similarly, DEL integrates IR instructions that model the checking of the state of the condition flags done by some instructions. Certain assembly instructions have an optional condition suffix added to their opcode. Taking **addeq** for example, with the "**eq**" being a condition that has to be met (i.e., the Z flag bit has to be set to 1) for the instruction to be executed and its effects reflected on the state of the corresponding register in DEL's register model.

4.3 Concept

DEL is a lifter that implements a combined translation with static and dynamic symbolic execution. Strictly speaking, DEL comprises a static and dynamic component. The static component consists of a CFG re-constructor, an assembly code to LLVM IR translator, and an SSE engine. DEL's dynamic component has a Z3 solver coupled with a memory model and a register model, acting as a DSE engine.

DEL has two different run modes; the tool can either run in a static mode or a dynamic mode. The static run mode only makes use of the static component. The dynamic run mode makes use of both the static and dynamic components. In a way, DEL's dynamic run starts with a static run, then the dynamic component steps in to perform the DSE. Figure 4.4 illustrates DEL's static and dynamic run modes. Algorithms 3 and 4 highlight the pseudocode descriptions for the static and dynamic runs, respectively. Before starting a static or a dynamic run, the memory model initial state μ_i is initialized through a separate input data file extracted from the disassembled binary. The input data file captures the state of the memory before stepping into the start entry point of the program.

When DEL runs in the static mode, it reconstructs a preliminary CFG, statically translates the input binaries into a primary IR module, and finally performs the SSE, specifically targeting the indirect branching instructions in the primary IR module. At this point, the IR module does not cover indirect control flow targets with exact resolved values. The SSE aims to define the possible range of addresses an indirect jump target could resolve to for all statically detected indirect jumps. The SSE determines the possible program paths leading to the basic block whose tail instruction is the indirect jump in question. The goal is to generate a Z3 expression for each possible path identified for each indirect jump in the program. The Z3 expressions define the possible jump target addresses' calculations in the form of Equation 4.1.

When DEL is run in the dynamic mode, it executes a static run. The generated Z3 expressions of all indirect control flow targets are passed to the Z3 solver of the dynamic component as additional satisfiability constraints when solving for indirect control flow targets. For a given input configuration, DEL's dynamic component performs the DSE of the primary IR module to resolve indirect control flow targets to their concrete values. The dynamic run is input dependent, meaning different inputs to the program result in different paths being executed during the run. The dynamic component requires different inputs to ensure all the possible paths leading to an indirect jump's basic block are covered. Generally speaking, the dynamic run operates iteratively. Each iteration involves using an input configuration that results in a particular execution path. DEL then performs the DSE of the path governed by the chosen input to resolve an exact target address value for each indirect jump statically detected on that particular execution path. The dynamic run ends once all possible jump targets of all statically detected indirect jumps have been resolved or when the tool has exhausted all given input cases.

In a way, DEL's static and dynamic components complement one another to generate a final IR Module that tries to capture the complete control flow of the program being analyzed. Following are the sections illustrating the implementation of each component.

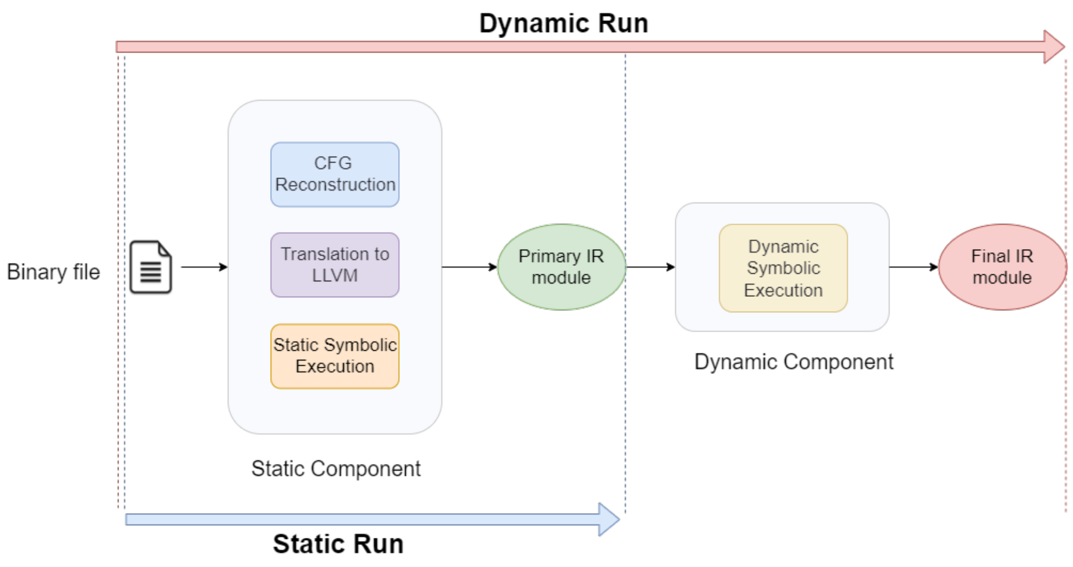


Figure 4.4: DEL 's static and dynamic run modes.

Algorithm 3 Static run pseudocode.

```
Require: assemblyInstructionsContainer
Output: preliminaryIRModule, IndirectJumpsZ3expressionsMap
Function staticRun(assemblyInstructionsContainer)
    Call controlFlowGraphReconstruction(assemblyInstructionsContainer)
    for i = 0 to assemblyInstructionsContainer.size - 1 do
        Call translationToLLVM(assemblyInstructionsContainer[i])
        i = i + 1
    Call staticSymbolicExecution(assemblyInstructionsContainer)
```

Algorithm 4 Dynamic run pseudocode.

```
Require: assemblyInstructionsContainer, InputConfiguration
Output: finalIRModule
Function dynamicRun(assemblyInstructionsContainer, InputConfiguration)
    if staticRun function was not called before then
        Call staticRun(assemblyInstructionsContainer)
    for i = 0 to preliminaryIRmodule.size - 1 do
        dynamicSymbolicExecution(preliminaryIRmodule[i])
        i = i + 1
    Repeat
        Call dynamicRun(newInputConfiguration)
    Until numberOfIndirectJumpsResolved = numberOfIndirectJumpsDetected
            Or no new inputConfiguration given
```

4.4 Implementation

This section illustrates how DEL's static and dynamic components are implemented. DEL's static component has three main features: CFG re-construction, translating assembly instructions into LLVM IR instructions, and SSE. DEL's dynamic component's main feature is performing the DSE. The following sections explain how each feature of the static and dynamic components is implemented, starting with the static component.

4.4.1 Static component

DEL's static component has three main roles during the static run. It iterates through the input assembly code while gathering information to reconstruct a preliminary control flow graph. It then iterates once more through each instruction in the input assembly code and translates it into an equivalent set of IR objects. Finally, it performs an SSE, targeting the indirect control flow branching instructions. The next sections explain the implementation of the three main features of the static component.

4.4.1.1 CFG re-construction

DEL's static component adopts the basic block creation algorithm for control flow graph re-construction, shown in Algorithm 5. Firstly, it iterates through the input assembly code and identifies instructions that are leaders. A leader is the first instruction of a basic block. The first instruction in the program is identified as a leader. Moreover, any instruction that succeeds a jump is also identified as a leader. Instructions that are targets of branching instructions are also classified as leaders. Once leaders are identified, DEL proceeds with the identification of tail instructions. Those are defined as any instruction that marks the termination of a basic block. A good example of tail instructions would be any jump instruction. Once DEL identifies leader and tail instructions, it segregates the input assembly code into basic blocks. Each basic block is represented as a block of instructions with incoming and outgoing edges. The incoming edges are the group of basic blocks that end with a jump instruction whose target address is the address of the first instruction of the basic block in question. On the contrary, a basic block's outgoing edges are those blocks that start with an instruction whose address is the target address to which the tail instruction of the basic block jumps. Once basic blocks have been specified, DEL reconstructs a preliminary control flow graph illustrating the predecessor and successor relationships between the different basic blocks.

DEL's implementation makes use of C++'s object-oriented programming concepts. It creates Assembly_Code_Instruction objects. Each Assembly_Code_Instruction object has, as attributes, an address, an opcode, registers, and immediates. Each Assembly_Code_Instruction object belongs to an Assembly_Code_Basic_Block object, which belongs to an Assembly_Code_Function object. As DEL statically constructs the preliminary control flow graph, assembly objects are updated with information highlighting relationships between the different objects. For example, how basic blocks are related to one another. Each Assembly_Code_Basic_Block object has successors and predecessors attributes which are also Assembly_Code_Basic_Block objects. The control flow graph also highlights the caller-callee relationships between different Assembly_Code_Function objects. Each Assembly_Code_Function object has callers and callees attributes of class type Assembly_Code_Function. Figure 4.5 illustrates a customized UML class diagram that highlights the relationships between the classes of the different assembly code objects implemented by DEL (for simplicity, class methods have been omitted from the diagram).

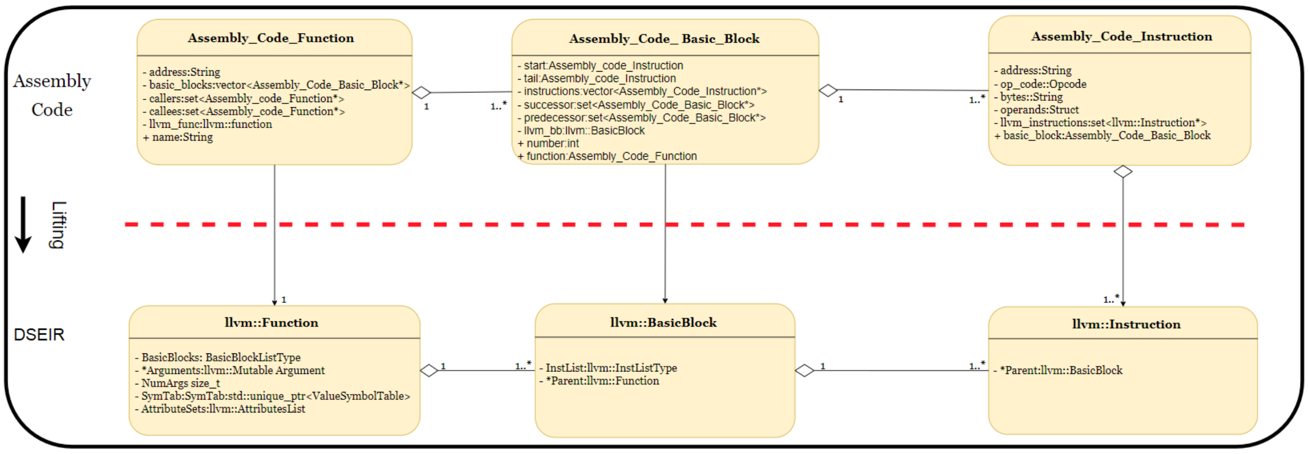

Assembly Code
Lifting
DSEIR
Assembly_Code_Function
- address:String
- basic_blocks:vector<Assembly_Code_Basic_Block*>
- callers:set<Assembly_code_Function*>
- callees:set<Assembly_code_Function*>
- llvm_func:llvm::function
+ name:String
Assembly_Code_ Basic_Block
- start:Assembly_code_Instruction
- tail:Assembly_code_Instruction
- instructions:vector<Assembly_Code_Instruction*>
- successor:set<Assembly_Code_Basic_Block*>
- predecessor:set<Assembly_Code_Basic_Block*>
- llvm_bb:llvm::BasicBlock
+ number:int
+ function:Assembly_Code_Function
Assembly_Code_Instruction
- address:String
- op_code::Opcode
- bytes::String
- operands:Struct
- llvm_instructions:set<llvm::Instruction*>
+ basic_block:Assembly_Code_Basic_Block
llvm::Function
- BasicBlocks: BasicBlockListType
- *Arguments:llvm::Mutable Argument
- NumArgs size_t
- SymTab:SymTab:std::unique_ptr<ValueSymbolTable>
- AttributeSets:llvm::AttributesList
llvm::BasicBlock
- InstList:llvm::InstListType
- *Parent:llvm::Function
llvm::Instruction
- *Parent:llvm::BasicBlock
1
1..*
1
1..*
1
1..*
1
1..*
1
1..*

Figure 4.5: DEL 's UML class diagram.

At this point, the constructed control flow graph does not account for indirect control flow targets. To resolve indirect control flow targets, DEL has a dynamic component, explained in detail in Section 4.4.2.

After the re-construction of the CFG, the static component proceeds with the next step of statically translating assembly instructions to their equivalent set of IR instructions.

Algorithm 5 Basic Block Partition Algorithm

```
1  $leaders = \{1\}$
2  for $i = 1$ to $|Number\ of\ instructions|$ do
3    if $instr(i)$ is a jump instruction then
4      $leaders = leaders \cup targets\ of\ instr(i) \cup instr(i+1)$
5  $worklist = leaders$
6  while $worklist\ not\ empty$ do
7    $x = first\ instruction\ in\ worklist$
8    $worklist$ = worklist - x
9    $block(x) = x$
10   for $i = x + 1;\ i <= |Number\ of\ instructions|\ \&\&\ i\ not\ in\ leaders;\ i++$ do
11     $block(x) = block(x) \cup \{i\}$
```

4.4.1.2 Static translation

Our dynamic lifting tool DEL lifts assembly instructions from the ARMv7-M ISA [7] into an LLVM IR. The IR is tailored for the DSE and hence the name DSEIR. DSEIR only uses a subset of LLVM instructions that are both explicit and self-contained, as was explained in Chapter 2.

DSEIR's design is quite straightforward; it has only 14 instructions. That means that every assembly instruction in the ARMv7-M ISA is translated into an average of 3-5 DSEIR instructions. Table 4.1 shows as an example, the equivalent DSEIR instructions for the **add**, **sub** and **lsl** ARMv7-M instructions. Each lifted DSEIR instruction is considered a static single assignment (SSA) where each IR statement updates only a single variable in the execution context. In other words, DSEIR is characterized as an explicit Binary-Based IR, as discussed in Chapter 2. Furthermore, the flag checking and updating functionalities of a lifted assembly instruction are also broken down into their own set of DSEIR instructions during lifting. In the DSE, Each DSEIR instruction could either update the memory model or the register model. Being SSA, while having its memory and register model alongside its condition flags checking and setting features, DSEIR becomes optimized for performing the DSE at the IR level.

Different analyses can be applied to the output DSEIR module, such as natural loop information analysis or a memory dependence analysis.

Table 4.1: DSEIR example table.

ARMv7-M instruction	DSEIR instructions
add r3, #4	%55 = load i32, i32* %R3, align 4 %56 = add i32 %55, 4 store i32 %56, i32* %R3, align 4
sub r1, #1	%55 = load i32, i32* %R1, align 4 %56 = sub i32 %55, 1 store i32 %56, i32* %R1, align 4
lsl r1,r2 #3	%85 = load i32, i32* %R2, align 4 %86 = shl i32 %85, 3 store i32 %86, i32* %R1, align 4

For each assembly instruction in the ARMv7-M ISA, DEL implements a C++ API that translates it into its equivalent set of DSEIR instructions.

Following the re-construction of the CFG, DEL iterates through each assembly instruction object and translates it into an equivalent set of DSEIR instruction objects.

DEL transfers information of the program's CFG stored as attributes of the assembly objects to the newly lifted IR objects. The control flow information is also represented at the IR level using C++ attributes. For instance, IR instruction objects belong to an IR basic block object. At the same time, each IR basic block object belongs to an IR function object. Essentially, DEL accurately represents assembly objects with their equivalent IR counterpart objects.

4.4.1.3 Static symbolic execution

DEL performs an SSE to identify all the potential target addresses that each indirect jump could resolve to. It determines all the possible program paths leading to the basic block whose tail instruction is the indirect branch in question. Each possible path would yield a Z3 expression for a potential target address of the indirect jump. The generated Z3 expressions are later used by DEL's dynamic component when resolving indirect jumps through the DSE.

The SSE can be divided into two main steps: path detection and formulating indirect jumps' target addresses into Z3 expressions. In SSE, DEL starts by detecting all possible paths that might lead to a basic block with an indirect branch. For each path detected, DEL then formulates the set of IR instructions on the path into a single Z3 formula that expresses the potential target address of the indirect jump in the form of a base and an offset, as was illustrated by Equation 4.1.

Next, we discuss each step of the SSE in more detail.

- Path detection

  Firstly, DEL's static component identifies all possible paths that might lead to a basic block with an indirect jump. It combines two algorithms, a depth-first search algorithm (DFS) [75] to detect all possible paths from a source s to a destination d in a directed acyclic graph and Johnson algorithm [51] for finding all simple cycles in the program's CFG. Both algorithms work together to identify all possible paths starting from the start entry point of the program up to the point of an indirect branch. As the DFS algorithm traverses the CFG, it makes sure not to visit the same node twice [75]; hence it is unable to detect and integrate cycles (loops) in a path between two nodes. Consequently, we additionally use the Johnson algorithm to detect the loops and subsequently add them to their corresponding paths.

- Formulating indirect jump target addresses into Z3 expressions

  Here, in this step, the main goal is to primarily identify for each indirect jump's target address register all the relevant assembly instructions that directly influence the value stored in that register. Algorithm 6 illustrates our approach to identifying all the assembly instructions that take part in calculating the value stored in the indirect branch target address register. For each identified potential execution path from the start entry point of the program to the indirect branch in question, the algorithm starts from the indirect jump instruction and goes back up the path, searching for the first preceding instruction **I_pre** that sets the register containing the target address and appends it to the relevant instructions set. The second step is to identify the set of registers that hold the operands used by the instruction **I_pre** to set the target address register. The algorithm repeats this process of identifying the first preceding instructions that set each register **r** in the registers set while appending the instructions to the relevant instructions set. For each newly identified **I_pre** instruction, the registers set is updated with the registers that hold the operands of **I_pre**. The algorithm ends once we reach an **I_pre** instruction that is a memory load instruction or a **mov{s}** instruction that sets **r** to an immediate value. All instructions in the relevant instructions set must belong to a single potential execution path leading to the indirect jump in question. For clarity, Figure 4.6 highlights in yellow what our algorithm considers as relevant instructions for the indirect jump in basic block 4 of a simple example program.

  After identifying the relevant assembly instructions, each DSEIR instruction in the set of IR instructions of each relevant assembly instruction is parsed into a Z3 expression. Finally all Z3 expressions for all the IR instructions on a single potential execution path are factorised into one single Z3 expression taking the form of Equation 4.1.

Algorithm **6** Find relevant instructions pseudocode

```
Require: assemblyInstructionsContainer
Output: relevantInstructionsSet
Function getRelevantInstructions(assemblyInstructionsContainer, registerToFollow)
for i = 0 to assemblyInstructionsContainer.size - 1 do
    if assemblyInstructionsContainer[i].getRegisters()[0] == registerToFollow then
        I_pre = assemblyInstructionsContainer[i]
        assemblyInstructionsContainer.erase(assemblyInstructionsContainer[i])
        relevantInstructionsSet.insert(I_pre)
        registersSet = {}
        i = i + 1
        if I_pre is memory load or mov{s} with immediate then
            break From Current Function Call Frame
        else
            for j = 1 to I_pre.getRegisters().size - 1 do
                registersSet.insert(I_pre.getRegisters()[j])
                j = j + 1
            for r in registersSet do
                Call getRelevantInstructions(assemblyInstructionsContainer, r)
```

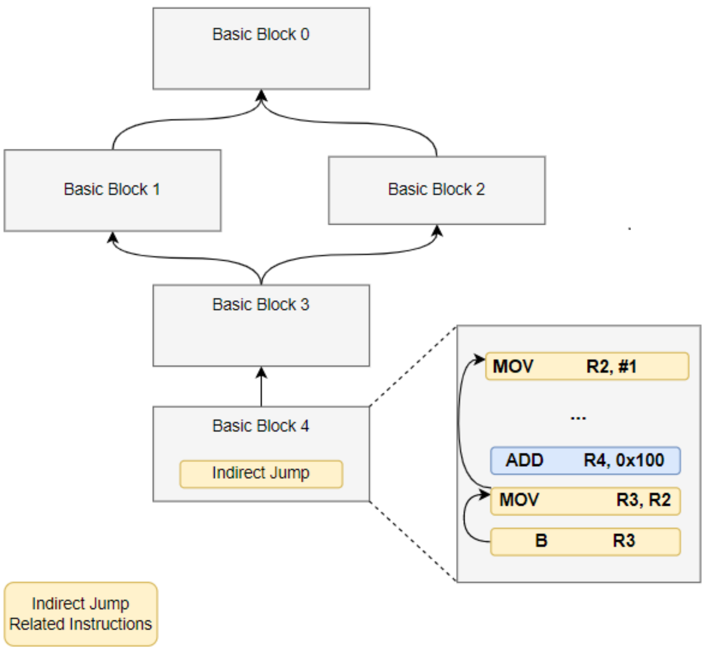


Figure 4.6: An indirect jump's relevant instructions highlighted in yellow.

After performing the SSE on the example program from Listing 5, DEL generated two Z3 expressions for the two potential-jump targets of the indirect branch instruction in line 31 of Listing 6. Equations 4.5 and 4.6 define the two possible jump target addresses of the indirect branch as Z3 expressions as generated by the SSE.

$$J_{Z3}\_potential\_target\_1 = (select\ MEM\ (bvadd\ (select\ MEM\ (bvadd\ pc\ \#x00000020))\\ (bvshl\ \#x00000000\ \#x00000002))) \tag{4.5}$$

$$J_{Z3}\_potential\_target\_2 = (select\ MEM\ (bvadd\ (select\ MEM\ (bvadd\ pc\ \#x00000020))\ (bvshl\ \#x00000001\ \#x00000002))) \quad (4.6)$$

Once the static run ends, DEL passes the Z3 formulas of all potential paths of all indirect branching instructions to DEL's dynamic component. The formulas are treated as additional satisfiability constraints by the Z3 solver when performing the DSE of the preliminary IR module.

4.4.2 Dynamic component

The dynamic component takes as input the IR module generated from the static component and the Z3 formulas for each potential target address of each indirect jump in the program. The dynamic component then performs the DSE using the Z3 solver. Each IR instruction in the input module is first parsed into a Z3 expression to be dynamically symbolically executed. The following sections illustrate in detail how the dynamic component is implemented.

- **Translation to SMT expressions**

  In the DSE, the first step is to compile the DSEIR module generated from DEL's static run into SMT expressions. In the later stages, we analyze the program's execution by using these expressions. IR instruction objects are parsed into Z3 expressions, which the Z3 solver then evaluates during DEL's dynamic run. The operands of each instruction are then fed into the Z3 solver in a way that reflects the mathematics underlying the IR instruction's effect on the solution state. Each SSA instruction can be aptly converted into one SMT expression by applying array and bit-vector theories, easing the translation process [10]. For example, the SSA IR [$\%r1 = add\ i32\ \%r0,\ 1$] is translated as shown in Equation 4.7 adapted from [10]. The same applies to memory instructions. For example, the SSA instruction shown in Equation 4.8 adapted from [10] is calculated as μ[0x00008000] where μ is the memory model and 0x00008000 is the load address. The translator repeats the preceding steps for each IR operation.

$$[\%r1 = add\ i32\ \%r0,\ 1] \Rightarrow BitVec(r1, size) = BitVec(r0, size) + BitVec(1, size) \quad (4.7)$$

$$r2 = [data\_0x0008000] \Rightarrow \mu[0x00008000] \quad (4.8)$$

- **Symbolic execution engine**

  Z3 is used to construct a dynamic execution engine. Its purpose is to execute SMT expressions directly on the memory and register models in DEL's dynamic run. Similar to the work done by [10], the engine has n states each of them reflects any alteration in the registers ($\Delta\rho$), the memory ($\Delta\mu$), or the stack ($\Delta\sigma$) state following a single expression's execution (a single DSEIR instruction). The number of states n should be identical to the number of executions of each instruction in the IR module during the dynamic run. The execution path followed during DEL's dynamic run depends on the input configuration given to DEL before the dynamic run starts. While translating [$\%r1 = add\ i32\ \%r0,\ 1$], the translator is first examining if r1 and r0 have existing variables in the register model. If yes, the value of r0 is retrieved from the engine, then an immediate value of 1 is added to it, and finally, the result is stored in r1. If r0 has a former value of 10, then the translation is performed as outlined in Equation 4.9 adapted from [10].

$$[\%r1 = add\ i32\ \%r0,\ 1] \Rightarrow BitVec(r1, size) = BitVec(10, size) + BitVec(1, size) \quad (4.9)$$

- **Execution**

  Similar to the approach adopted by [10], the initial state S_i $<\rho_i,\ \mu_i,\ \sigma_i>$ is fed into the DSE solver. As each DSEIR instruction is symbolically executed, the engine state changes from S_i to S_{i+1}. DEL iterates through all instructions in the control flow path until the final state S_f is attained.

  By combining the execution engine and the memory and register models, SMT expressions can be executed dynamically [10]. The satisfiability of each expression is verified before the execution engine alters the engine state from S_i to S_{i+1}. Consider an example where the previous value of r0 was 10. In Equation 4.10 adapted from [10], the SMT expression evaluates to true and sets the value of r1 to 11. SMT expressions involving memory follow the same principle. As a result of executing each instruction, the engine will transition from state s_i to state $s_{[i+1]}$. Execution of an expression results in an engine state $s_{[i+1]} = s_i + \Delta_k$ where $k = [\rho, \mu, \sigma]$ [10].

$$BitVec(r1, size) = BitVec(r0, size) + BitVec(1, size) \tag{4.10}$$

  As they are executed, expressions are divided into three main classes: memory expressions, registers expressions, and director expressions [10]. A solver follows the execution path determined by director expressions (branching instructions). For instance, the DSEIR instructions in Listing 11 are parsed into the SMT expression in Listing 12 which evaluates the condition r1 = 0 in order to determine the following basic block to be visited.

```
%1 = icmp eq i32 %r1, 0
br i1 %1, label %BB1, label %BB2
```

Listing 11: DSEIR branching instructions.

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

Listing 12: CFG merging in the SSA context.

For SMT expressions, Algorithm 7 adapted from [10] describes how the DSE works. The algorithm takes as input the CFG from DEL's static component. As discussed in Section 4.4.1.1, the CFG highlights the predecessor and successor relationships between different basic blocks in assembly code. The algorithm iterates through the instructions of each basic block of the CFG. For every instruction, I in basic block B, the satisfiability of its expression is checked. The state s_i is modified depending on its effect on the engine model. In order to execute the instructions, the engine state must be modified and the transition condition assessed. If the instruction is a conditional branch, it can lead to either basic block B_x or B_y. The current state of the condition flags is checked to determine which basic block should be executed next. The execution normally runs from one basic block to the next till the exit function of the program.

Algorithm 7 Z3 Execution Engine

```
Input : $g_{cfg}$ : control flow graph
Initialize $S_i < \rho_i, \mu_i, \sigma_i >$
for $B_i$ in $g_{cfg}$ do
    for $I_i$ in $B_i$ do
        if $I_i$ is RegisterSet then
            $s_i = s_i + \Delta \rho_i$
        if $I_i$ is MemoryWrite then
            $s_i = s_i + \Delta \mu_i$
        if $I_i$ is Push || Pop then
            $s_i = s_i + \Delta \sigma_i$
        if $I_i$ is conditional branch: $B_x$, $B_y$ then
            $B_{next} = B_x \mid\mid B_y$
            if $B_{next}$ is $NULL$ then
                exit
```

- **Input based execution paths**
  DEL's dynamic run is input-based. Strictly speaking, different inputs govern different program execution paths. More than one execution path from the start of the program could lead to the basic block of an indirect jump. DEL's SSE aims to acquire all possible paths leading to an indirect jump's basic block and formulate each target of an indirect branching instruction as a Z3 expression. Since each possible path might yield a different jump target for an indirect jump, different program input configurations could result in a different control flow target for a given indirect branching instruction. The source code in Listing 5 shows an example where different inputs to the program could result in different potential-jump targets for a single indirect jump, as was explained in Section 4.1.
  DEL's dynamic component operates by iterating through different program inputs. It performs the DSE of each possible execution path governed by a chosen input. Its main goal is to resolve all possible indirect control flow targets of indirect branching instructions to their exact address values. The dynamic run ends once ideally all possible jump targets of all detected indirect jumps have been resolved or when all possible input configurations passed to the tool have been exhausted.
  It is important to highlight that DEL's dynamic capability is limited by the range of the inputs tested out during the dynamic run. However, choosing the input configurations that guarantee the execution of all possible program paths during the dynamic run is outside the scope of this thesis.

- **Loop bound analysis**
  A useful feature of DEL's dynamic component is detecting how many times a basic block has been executed during the dynamic run. During the DSE process, DEL keeps track of each assembly instruction that has been executed. Since each instruction object belongs to a basic block object as was explained in Section 4.4.1.1, the number of times an instruction has been executed reflects the number of times its basic block has been visited during the dynamic run. Such a feature, when coupled with Johnson's cyclic graph detection algorithm [50], could be particularly useful in conducting a loops bound analysis. This application could help improve the work done by [10].

Chapter 5

Evaluation

So far, we have looked at the challenges that modern-day lifters face when considering IR analysis. We have also introduced a novel hybrid symbolic execution technique in the lifting process for resolving indirect jumps using the Z3 solver. Our approach aims at generating a complete and enhanced IR of data flow space applications. In this chapter, we firstly evaluate DEL's ability to resolve indirect jumps present in the example program in Listing 13 adapted from [65, 69] as compared to Angr [32], a recent binary analysis framework based on UCSE. Secondly, we evaluate the effectiveness of our approach in resolving indirect jumps of a large scale C++ application developed by the Tasking Framework, the Join fork example in Listing A.1.

5.1 DEL Vs Angr

Here, we showcase DEL's ability to resolve indirect jumps present in the example program of Listing 13 in comparison to Angr, which uses a control-flow recovery algorithm that tries to resolve indirect control flow targets by employing a data-flow analysis.

Our work aims to lift to an IR module that captures the control flow of an input binary. In order to perform WCET analysis at the IR level, the control flow model needs to be valid, which means that all potential control flow that exists in the binary must also be present in the IR module. The IR module quality relies on the control flow's preciseness. Ideally, there should be as few infeasible transitions in the control flow as possible. However, accurate resolution of indirect control flow targets necessitates the calculation of all feasible outcomes, which in principle is impractical [65].

In Listing 13, since the indirect call at line 10 relies on the value that the count variable holds, the resolution of the possible targets of the indirect jump necessitates an examination of the potential values the count variable can take. Since the value stored in the argument vector (ARGV) is dependent on the user input, the previous loop results in a massive number of paths during analysis. The path explosion problem arises, making precise analysis impossible as was argued by [65] for the used example.

```
1  int foo_1(void) { return 5; }
2  int foo_2(void) { return 6; }
3  int main(int arge, char** argv) {
4          int (*procs[]) (void) = { foo_1,foo_2 };
5          int count = 0;
6          for (int i = 0; i < 100; i++) {
7                  if (argv[1][i] == '0') break;
8                  if (argv[1][i] == 'Z') count ^= 1;
9          }
10         return procs[count]();
11 }
```

Listing 13: Example program adapted from [65, 69].

We proposed SSE to formulate all potential-jump targets of the indirect jump in question to solve this issue. The generated formulas take the form of a fixed base address (the first address in the jump table) added to a variable offset (limited to the range of addresses of functions in the input program) as was explained in Section 4.1. Our solution narrows down the search scope for the possible jump target values that the indirect jump can take. We then perform the DSE to resolve as many potential-jump targets as possible by trying different inputs to our example that satisfy the formulas generated from the SSE.

As previously discussed, the quality of the IR module generated depends on the precision of the constructed control flow graph. The precision of a CFG is notoriously difficult to assess as one would need a perfect comparison model [65]. Rather, in this section, we evaluate our solution's quality through lifting the example given in the Listing 13. For the given example, [65] argued that most of the control flow could be reconstructed directly without performing a data-flow analysis. Because branches and loops are constructed via direct branch instructions, it is possible to resolve them without further input. On the other hand, the indirect call at line 10 cannot be resolved easily since its target address relies on the values that r3 can hold during execution, as shown in the red block of Figure 5.2. The algorithm utilized by Angr for control flow recovery was unable to resolve the jump targets of the indirect jump in question, resulting in an erroneous outcome [65]. In contrast, our solution successfully resolved both potential-jump targets for the indirect jump. Firstly, DEL's static component generated the Z3 formula shown in Equation 5.1 for the potential jump targets of the indirect jump. Figure 5.1 shows an incomplete CFG reconstructed from DEL's static component. It shows that the SSE was able to identify all six potential paths from the main entry point of the program to the basic block with the indirect jump. However, just one formula was generated for all six paths. Although the potential target formula seems the same for all detected paths, the underlying path to be executed for the formula is different. Each path can enclose unique instructions that store data in specific addresses in the memory model. Since it is impossible to know the exact addresses to which str instructions store into memory without executing the program, the effects of such instructions are not observed in the Z3 formulas generated. Consequently, the DSE execution of each path might yield a different potential-jump target address for what seems to be a single identical formula for different potential paths. By iteratively varying the input configuration, DEL's dynamic component was able to execute each of the possible six paths identified, resolving Equation 5.1 into two possible addresses from the jump table { 00008000, 0000801c }, which are the addresses for functions foo_1 and foo_2, respectively.

$$\begin{aligned} J_{Z3} = (&Select\ MEM\ (bvadd\ (bvadd\ (bvadd\ \#xfffffffc\ (bvadd\ \#x00000004\ sp)) \\ &(bvnot\ (bvor\ (bvnot\ maskBit)\ (bvnot\ (select\ MEM\ (bvadd\ (bvadd\ \#x00000004\ sp) \\ &\#xfffffff8)))))) \ \#xfffffff0)) \end{aligned} \tag{5.1}$$

Figure 5.2 shows the final reconstructed CFG after DEL has performed the DSE; the red block highlights the basic block with the indirect jump. The orange blocks highlight the two potential-jump targets, foo_1 and foo_2, as identified by our algorithms. The red arrows resemble the resolved indirect control flow targets, where based on the inputs entered by the user, either function foo_1 or function foo_2 are called.

Listing B.1 illustrates the final lifted IR module of the example program. For the given input configuration, DEL's dynamic run followed an execution path that yielded function foo_1 as the resolved indirect control flow target for the indirect branch in the example program.

For further evaluating the performance of our approach, We compared our solution's execution time to that of Angr's CFG re-construction approach. Both tools were operated on a workstation with a Linux operating system, i7-9750H processor, and 16GB RAM. The results are shown in Table 5.1. Here, it shows that DEL is significantly slow compared to Angr; however, this is insignificant to our objective as the IR analysis of DEL's lifted module is normally carried out offline amidst the design validation and verification phase. As DEL operates, it also consumes more memory storage space compared to Angr due to the size and complexity of its C++ implemented objects.

Table 5.1: Performance results: Angr vs DEL.

Tool	Binary Size (Kbyte)	%CPU	Average Memory (MiB)	Execution Time (sec)
Angr	34	24%	55	35
DEL	34	24%	335	125

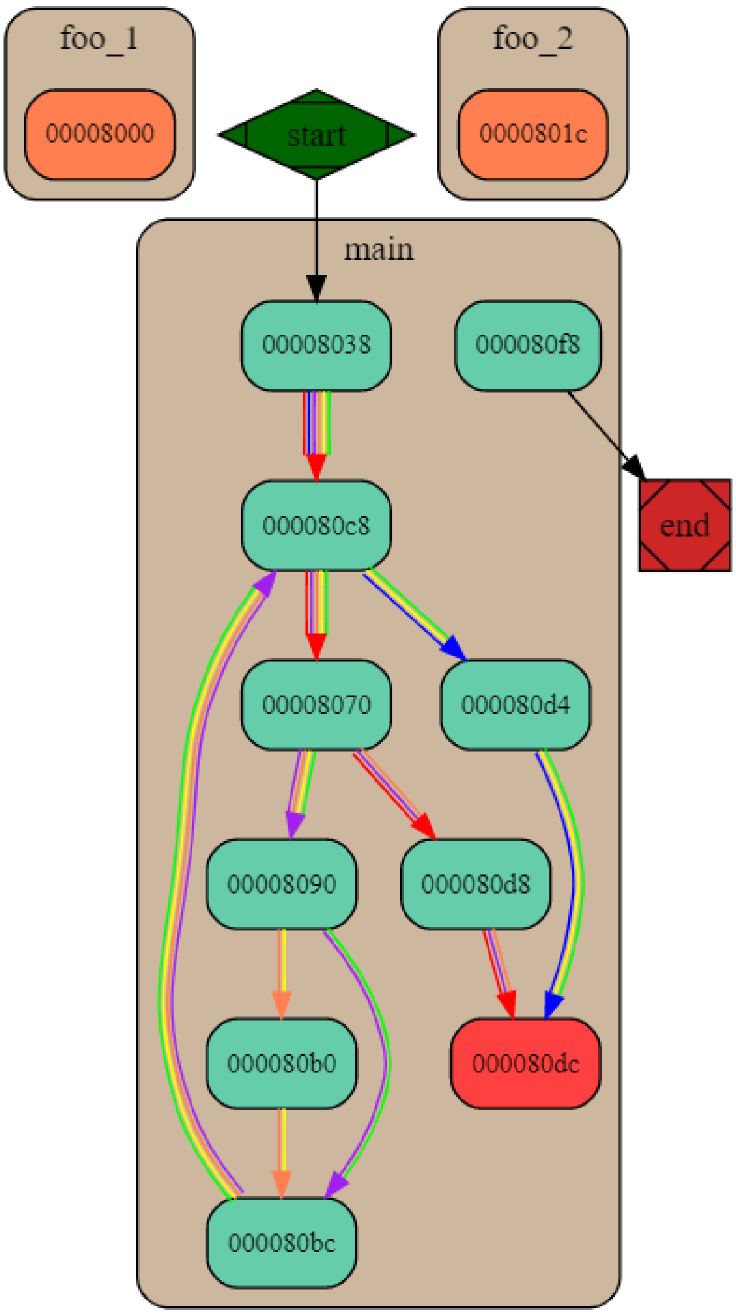
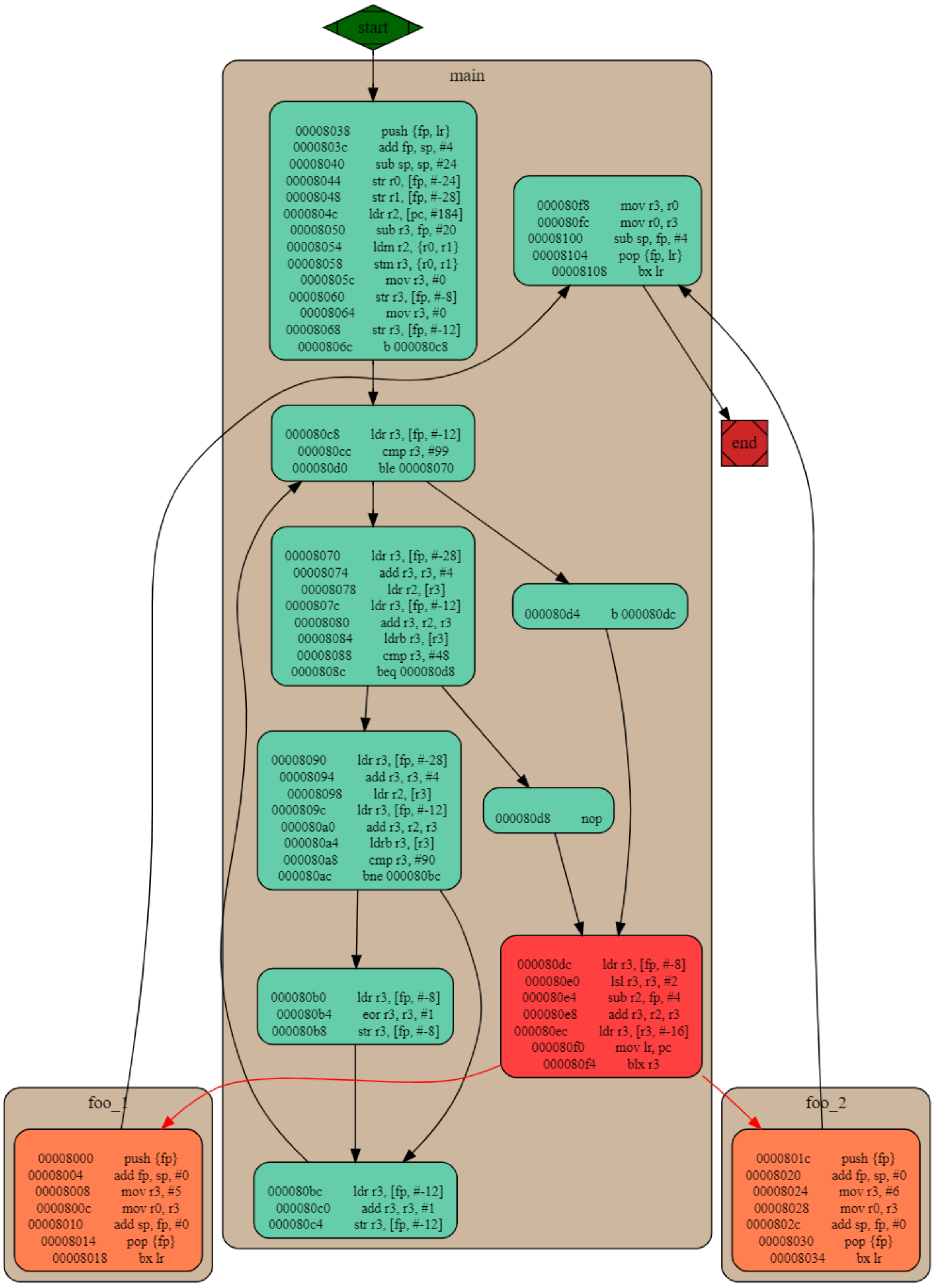


Figure 5.1: Six potential paths from the start entry point till the basic block of the indirect jump (000080dc).



```
00008038    push {fp, lr}
0000803c    add fp, sp, #4
00008040    sub sp, sp, #24
00008044    str r0, [fp, #-24]
00008048    str r1, [fp, #-28]
0000804c    ldr r2, [pc, #184]
00008050    sub r3, fp, #20
00008054    ldm r2, {r0, r1}
00008058    stm r3, {r0, r1}
0000805c    mov r3, #0
00008060    str r3, [fp, #-8]
00008064    mov r3, #0
00008068    str r3, [fp, #-12]
0000806c    b 000080c8
```

```
000080f8    mov r3, r0
000080fc    mov r0, r3
00008100    sub sp, fp, #4
00008104    pop {fp, lr}
00008108    bx lr
```

```
000080c8    ldr r3, [fp, #-12]
000080cc    cmp r3, #99
000080d0    ble 00008070
```

```
00008070    ldr r3, [fp, #-28]
00008074    add r3, r3, #4
00008078    ldr r2, [r3]
0000807c    ldr r3, [fp, #-12]
00008080    add r3, r2, r3
00008084    ldrb r3, [r3]
00008088    cmp r3, #48
0000808c    beq 000080d8
```

```
000080d4    b 000080dc
```

```
00008090    ldr r3, [fp, #-28]
00008094    add r3, r3, #4
00008098    ldr r2, [r3]
0000809c    ldr r3, [fp, #-12]
000080a0    add r3, r2, r3
000080a4    ldrb r3, [r3]
000080a8    cmp r3, #90
000080ac    bne 000080bc
```

```
000080d8    nop
```

```
000080dc    ldr r3, [fp, #-8]
000080e0    lsl r3, r3, #2
000080e4    sub r2, fp, #4
000080e8    add r3, r2, r3
000080ec    ldr r3, [r3, #-16]
000080f0    mov lr, pc
000080f4    blx r3
```

```
000080b0    ldr r3, [fp, #-8]
000080b4    eor r3, r3, #1
000080b8    str r3, [fp, #-8]
```

```
000080bc    ldr r3, [fp, #-12]
000080c0    add r3, r3, #1
000080c4    str r3, [fp, #-12]
```

```
00008000    push {fp}
00008004    add fp, sp, #0
00008008    mov r3, #5
0000800c    mov r0, r3
00008010    add sp, fp, #0
00008014    pop {fp}
00008018    bx lr
```

```
0000801c    push {fp}
00008020    add fp, sp, #0
00008024    mov r3, #6
00008028    mov r0, r3
0000802c    add sp, fp, #0
00008030    pop {fp}
00008034    bx lr
```

Figure 5.2: DEL's full re_constructed control flow graph of the example program.

5.2 Evaluating DEL's dynamic run on the Tasking Framework

Here, we evaluate DEL's ability to resolve indirect jumps in a large-scale program, the Tasking Framework.

Jumps in the Tasking Framework fall into two classes. Firstly, direct jumps in the framework's architecture that are based solely on the system's design. Thus, their targets are determined during the compile-time and stay the same as the application runs. Secondly, Indirect jumps that can be found as virtual functions of developed tasks. Their targets' calculations are determined by the tasks being executed in run time. Our primary evaluation measure is calculating the percentage of indirect jumps resolved out of the indirect jumps visited during the DSE for each function in the case study.

Similar to [10], this thesis focuses solely on binary input task-triggered events. In a strict sense, the input events can be thought of as task on/off switches [10]. Through DEL, it is possible to force events (inputs) during the DSE of the enhanced IR module.

5.2.1 Experimental setup

Figure 5.3 illustrates the setup we proposed for measuring the number of indirect jumps resolved when lifting the Join fork example case study using DEL. Our experimental setup was built in as part of DEL's implementation. It primarily uses DEL's static component to detect all indirect jumps (bx and blx instructions) for each function in the case study, assuming the disassembler correctly disassembles the input binary. It then passes the control flow information of all the statically identified possible paths leading to each indirect jump detected and the preliminary DSEIR module to DEL's dynamic component. As explained in Chapter 4, DEL's dynamic component works iteratively. Each iteration tries out a different input configuration to visit a different execution path between iterations. The setup makes use of DEL's dynamic component to resolve as much as possible of the detected indirect jumps through the DSE of the DSEIR module. During this process, for each function, the setup reports the percentage of indirect jumps visited out of the indirect jumps statically detected and the number of indirect jumps resolved out of the indirect jumps visited. DEL moves on to the next function regardless of the unresolved indirect jumps. The current DSE iteration ends once DEL reaches the exit block of the program. The next DSE iteration starts with a different input aiming to visit a different execution path. The experiment ends once all given inputs to DEL have been exhausted or if DEL could resolve all statically detected indirect jumps. In practice, a developer can be satisfied with a finite number of inputs that resolve a subset of the indirect jumps detected. The setup then outputs a log file reporting the number of indirect jumps visited during the dynamic run for each function in the binary input file and the number of resolved indirect jumps.

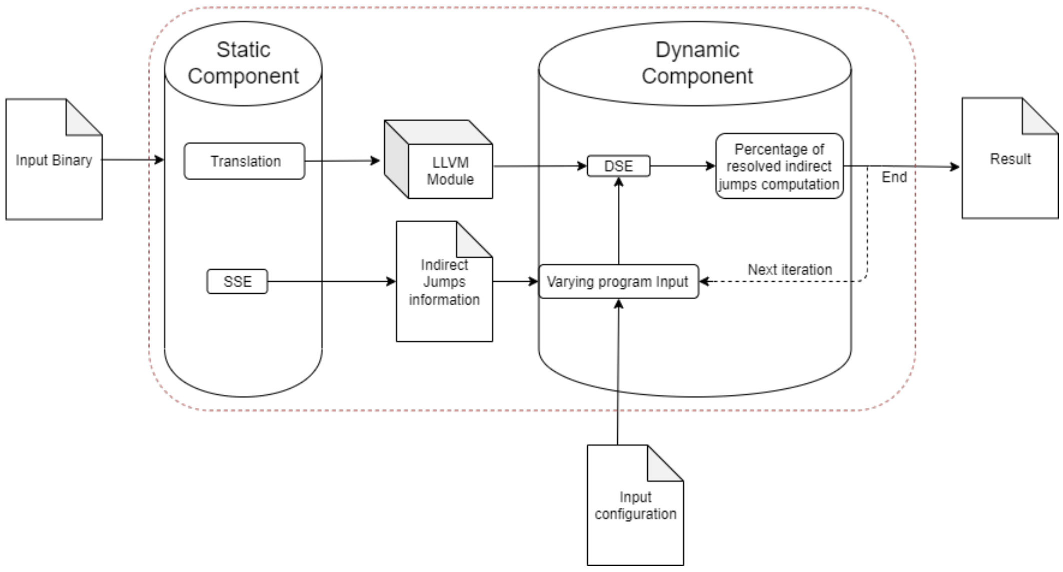


Figure 5.3: Experimental setup.

5.2.2 Indirect jumps' results

In this section, we detail the results of the indirect jumps resolved by DEL as tested on lifting the Tasking Framework's Join fork example use-case for a given input configuration.

Strictly speaking, there are three main sources of indirect jumps: virtual function calls, switch statements, and function pointers. In our case study, the indirect jumps detected were caused by virtual function calls. Neither switch statements nor function pointers were used in the Tasking Framework's implementation.

Functions containing indirect jumps in the Tasking Framework are mainly distributed across six modules: InputArray, Scheduler, Task, Event, Clock, and Group. We chose an example function for each module in our case study that encloses one or more indirect jumps caused by virtual function calls. We begin by briefly explaining the role of each function in the Tasking Framework accompanied by its source code and CFG. We then tabulate DEL's results for a given input configuration, highlighting the number of indirect jumps detected and the number of visited and resolved indirect jumps for each example function. Finally, we tabulate the results for all functions visited during DEL's dynamic run for the given input configuration.

5.2.2.1 Tasking Framework's functions

- **InputArray:**

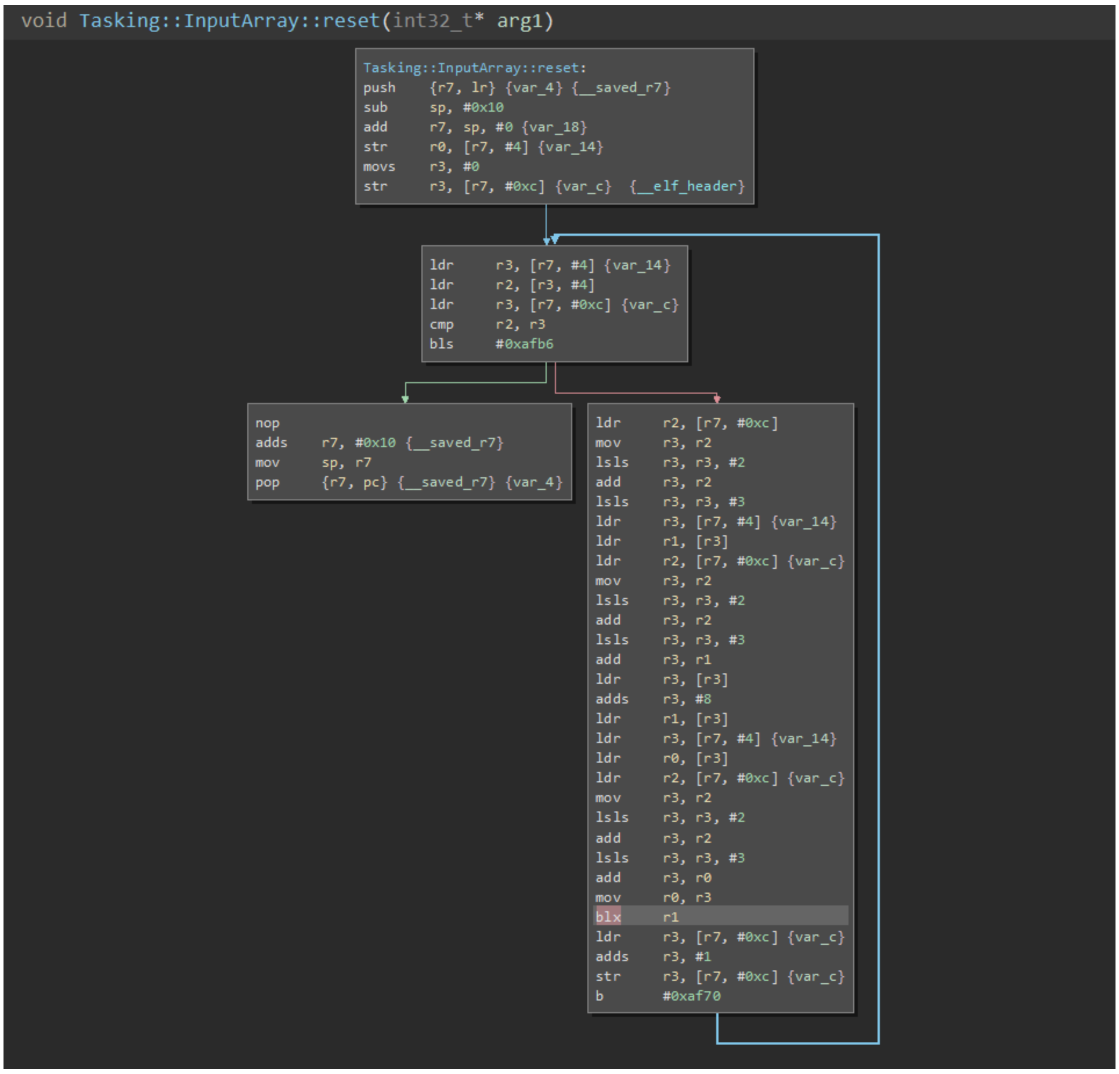
  The Tasking::InputArray::reset function performs the reset operation on all task inputs. All task inputs are stored in an input array. Listing 14 highlights the source code of the Tasking::InputArray::reset function. Here, the function invokes the reset method of each input array element in line 5. The reset method is implemented by the Input class as a virtual method as highlighted in line 158 in Listing C.1. It resets the activation state of each input task to 0 activations when the scheduler starts.

  Figure 5.4 highlights the CFG of the Tasking::InputArray::reset function where the virtual function call can be seen as the indirect branch blx r1.

Table 5.2 shows that for the given input configuration, DEL was able to successfully detect and resolve the single indirect jump detected for the Tasking::InputArray::reset function.

```cpp
void Tasking::InputArray::reset(void)
{
    for (unsigned int i = 0; i < impl.length; ++i)
    {
        impl.inputs[i].reset();
    }
}
```

Listing 14: Tasking::InputArray::reset function C++ source code.



```
void Tasking::InputArray::reset(int32_t* arg1)
```

```
Tasking::InputArray::reset:
push    {r7, lr} {var_4} {__saved_r7}
sub     sp, #0x10
add     r7, sp, #0 {var_18}
str     r0, [r7, #4] {var_14}
movs    r3, #0
str     r3, [r7, #0xc] {var_c}  {__elf_header}
```

```
ldr     r3, [r7, #4] {var_14}
ldr     r2, [r3, #4]
ldr     r3, [r7, #0xc] {var_c}
cmp     r2, r3
bls     #0xafb6
```

```
nop
adds    r7, #0x10 {__saved_r7}
mov     sp, r7
pop     {r7, pc} {__saved_r7} {var_4}
```

```
ldr     r2, [r7, #0xc]
mov     r3, r2
lsls    r3, r3, #2
add     r3, r2
lsls    r3, r3, #3
ldr     r3, [r7, #4] {var_14}
ldr     r1, [r3]
ldr     r2, [r7, #0xc] {var_c}
mov     r3, r2
lsls    r3, r3, #2
add     r3, r2
lsls    r3, r3, #3
add     r3, r1
ldr     r3, [r3]
adds    r3, #8
ldr     r1, [r3]
ldr     r3, [r7, #4] {var_14}
ldr     r0, [r3]
ldr     r2, [r7, #0xc] {var_c}
mov     r3, r2
lsls    r3, r3, #2
add     r3, r2
lsls    r3, r3, #3
add     r3, r0
mov     r0, r3
blx     r1
ldr     r3, [r7, #0xc] {var_c}
adds    r3, #1
str     r3, [r7, #0xc] {var_c}
b       #0xaf70
```

Figure 5.4: Tasking::InputArray::reset function CFG.

Table 5.2: Tasking::InputArray::reset function results.

Function	No. of detected indirect jumps	No. of visited indirect jumps	No. of resolved indirect jumps
Tasking::InputArray::reset	1	1	1

- **Scheduler:**

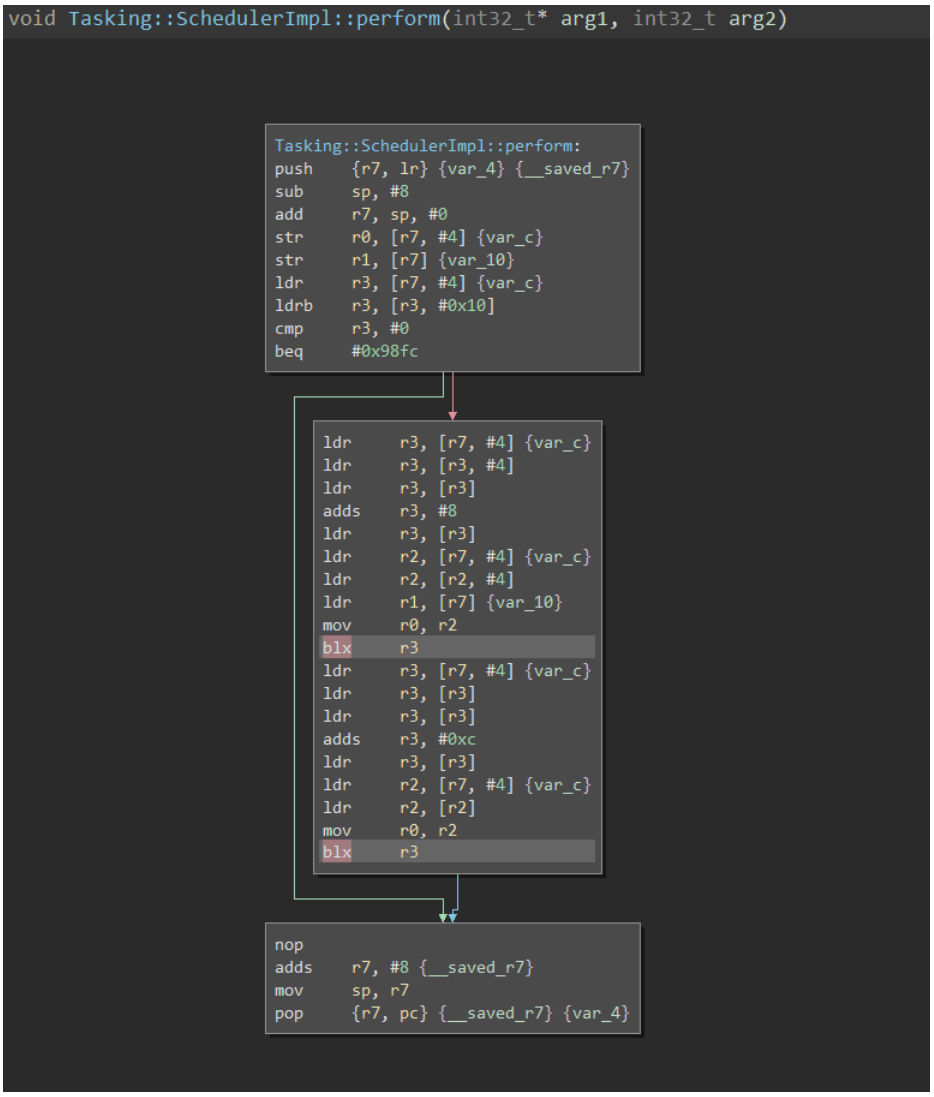
  The Tasking::SchedulerImpl::perform function Initiates the execution of a referenced task passed to it. By default, calling this function switches the state of the referenced task to pending. The exact starting time for executing the referenced task depends on the selected schedule policy and the number of available executors. A perform function call has no effect if the scheduler is not started or terminated. Listing 15 highlights the source code of The Tasking::SchedulerImpl::perform function. Both lines 7 and 8 have virtual function calls where the perform function calls the queue and the signal functions. Both the queue and signal functions are implemented as virtual methods by their relevant classes as seen in lines 70 and 116 in Listing C.2 and Listing C.3 respectively. Firstly, if the scheduler is running, the perform function invokes the queue function. Typically, the queue function is called when a task switches from wait to pending. It queues a task according to the policy into the run queue. An implementation of a scheduling policy must implement this function. Each task provides the management data structure to provide the memory space for the scheduling. Secondly, the perform function invokes the signal function. This function is called whenever a new task should perform, the run queue is empty, or the clock fires an event. It wakes up one of the executors of the scheduler instance.

  Figure 5.5 highlights the CFG of the Tasking::SchedulerImpl::perform function. Since both the queue and the signal functions are implemented as virtual methods in the Tasking Framework, both calls are considered by the compiler as indirect jumps. They are disassembled as the two blx instructions shown in the figure.

  Table 5.3 shows that for the given input configuration, DEL was able to successfully detect and resolve both indirect jumps detected for the Tasking::SchedulerImpl::perform function.

```
1  void Tasking::SchedulerImpl::perform(Tasking::TaskImpl& task)
2  {
3      // Do only something when the scheduler is running.
4      if (running)
5      {
6          // Queue task for execution and signal scheduler execution model
7          policy.queue(task);
8          static_cast<UnprotectedSchedulerAccess&>(parent).signal();
9      }
10 }
```

Listing 15: Tasking::SchedulerImpl::perform function C++ source code.



```
void Tasking::SchedulerImpl::perform(int32_t* arg1, int32_t arg2)

Tasking::SchedulerImpl::perform:
push    {r7, lr} {var_4} {__saved_r7}
sub     sp, #8
add     r7, sp, #0
str     r0, [r7, #4] {var_c}
str     r1, [r7] {var_10}
ldr     r3, [r7, #4] {var_c}
ldrb    r3, [r3, #0x10]
cmp     r3, #0
beq     #0x98fc

ldr     r3, [r7, #4] {var_c}
ldr     r3, [r3, #4]
ldr     r3, [r3]
adds    r3, #8
ldr     r3, [r3]
ldr     r2, [r7, #4] {var_c}
ldr     r2, [r2, #4]
ldr     r1, [r7] {var_10}
mov     r0, r2
blx     r3
ldr     r3, [r7, #4] {var_c}
ldr     r3, [r3]
ldr     r3, [r3]
adds    r3, #0xc
ldr     r3, [r3]
ldr     r2, [r7, #4] {var_c}
ldr     r2, [r2]
mov     r0, r2
blx     r3

nop
adds    r7, #8 {__saved_r7}
mov     sp, r7
pop     {r7, pc} {__saved_r7} {var_4}
```

Figure 5.5: Tasking::SchedulerImpl::perform function CFG.

Table 5.3: Tasking::SchedulerImpl::perform function results.

Function	No. of detected indirect jumps	No. of visited indirect jumps	No. of resolved indirect jumps
Tasking::SchedulerImpl::perform	2	2	2

- **Task:**

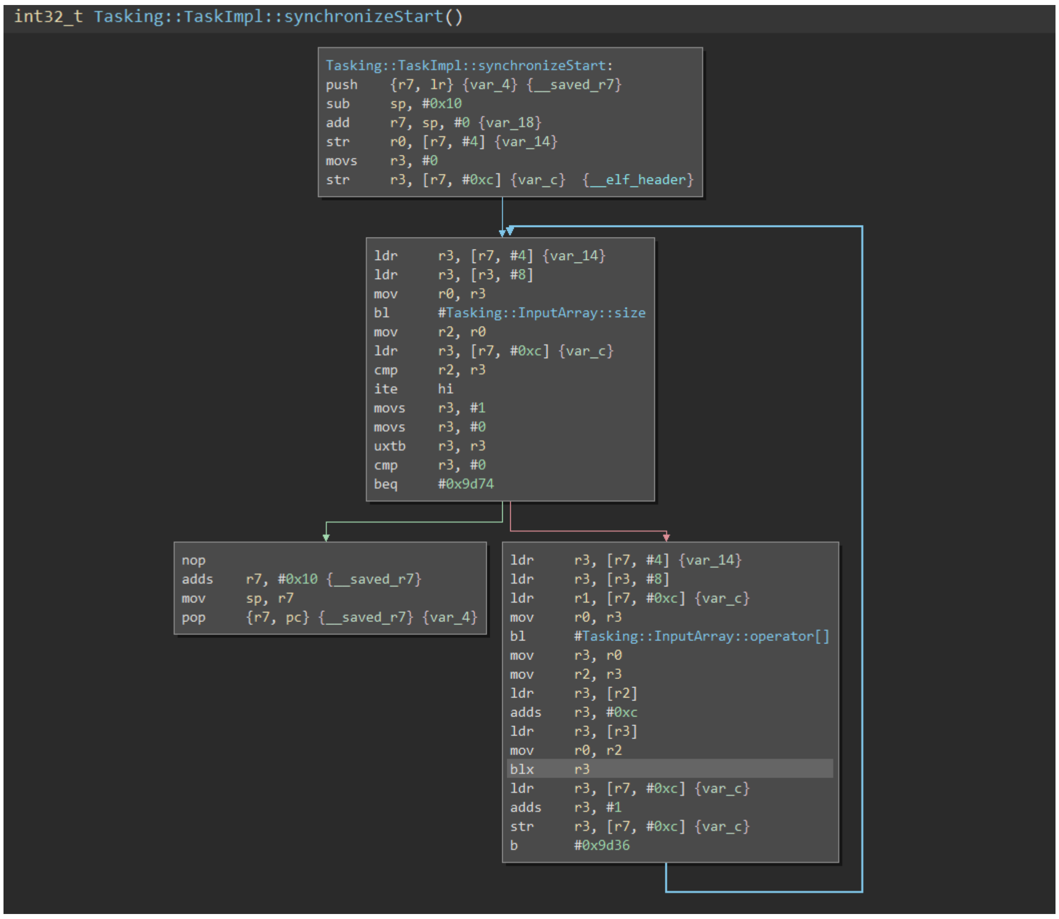
  The Tasking::TaskImpl::synchronizeStart function is called directly by the scheduler before executing a task. It loops over all inputs to call the synchronizeStart of all connected input channels. In line 5 of Listing 16, The Tasking::TaskImpl::synchronizeStart function invokes the Input::synchroniseStart function. The Input class implements the synchronizeStart function as a virtual method as shown in line 215 of Listing C.1. This function defines the associated task start to execute. It is protected against concurrent access to two tasks associated with the scheduler.

Figure 5.6 shows the CFG of the Tasking::TaskImpl::synchronizeStart function. Here, the virtual function call is highlighted as the indirect branching instruction blx r3.

Table 5.4 shows that for the given input configuration, DEL was able to successfully detect and resolve the indirect jump detected for the Tasking::TaskImpl::synchronizeStart function.

```cpp
void Tasking::TaskImpl::synchronizeStart(void)
{
    for (unsigned int i = 0; (i < inputs.size()); i++)
    {
        static_cast<ProtectedInputAccess&>(inputs[i]).synchronizeStart();
    }
}
```

Listing 16: Tasking::TaskImpl::synchronizeStart function C++ source code.



```
int32_t Tasking::TaskImpl::synchronizeStart()

Tasking::TaskImpl::synchronizeStart:
push    {r7, lr} {var_4} {__saved_r7}
sub     sp, #0x10
add     r7, sp, #0 {var_18}
str     r0, [r7, #4] {var_14}
movs    r3, #0
str     r3, [r7, #0xc] {var_c}  {__elf_header}

ldr     r3, [r7, #4] {var_14}
ldr     r3, [r3, #8]
mov     r0, r3
bl      #Tasking::InputArray::size
mov     r2, r0
ldr     r3, [r7, #0xc] {var_c}
cmp     r2, r3
ite     hi
movs    r3, #1
movs    r3, #0
uxtb    r3, r3
cmp     r3, #0
beq     #0x9d74

nop
adds    r7, #0x10 {__saved_r7}
mov     sp, r7
pop     {r7, pc} {__saved_r7} {var_4}

ldr     r3, [r7, #4] {var_14}
ldr     r3, [r3, #8]
ldr     r1, [r7, #0xc] {var_c}
mov     r0, r3
bl      #Tasking::InputArray::operator[]
mov     r3, r0
mov     r2, r3
ldr     r3, [r2]
adds    r3, #0xc
ldr     r3, [r3]
mov     r0, r2
blx     r3
ldr     r3, [r7, #0xc] {var_c}
adds    r3, #1
str     r3, [r7, #0xc] {var_c}
b       #0x9d36
```

Figure 5.6: Tasking::TaskImpl::synchronizeStart function CFG.

Table 5.4: Tasking::TaskImpl::synchronizeStart function results.

Function	No. of detected indirect jumps	No. of visited indirect jumps	No. of resolved indirect jumps
Tasking::TaskImpl::synchronizeStart	1	1	1

- **Event:**

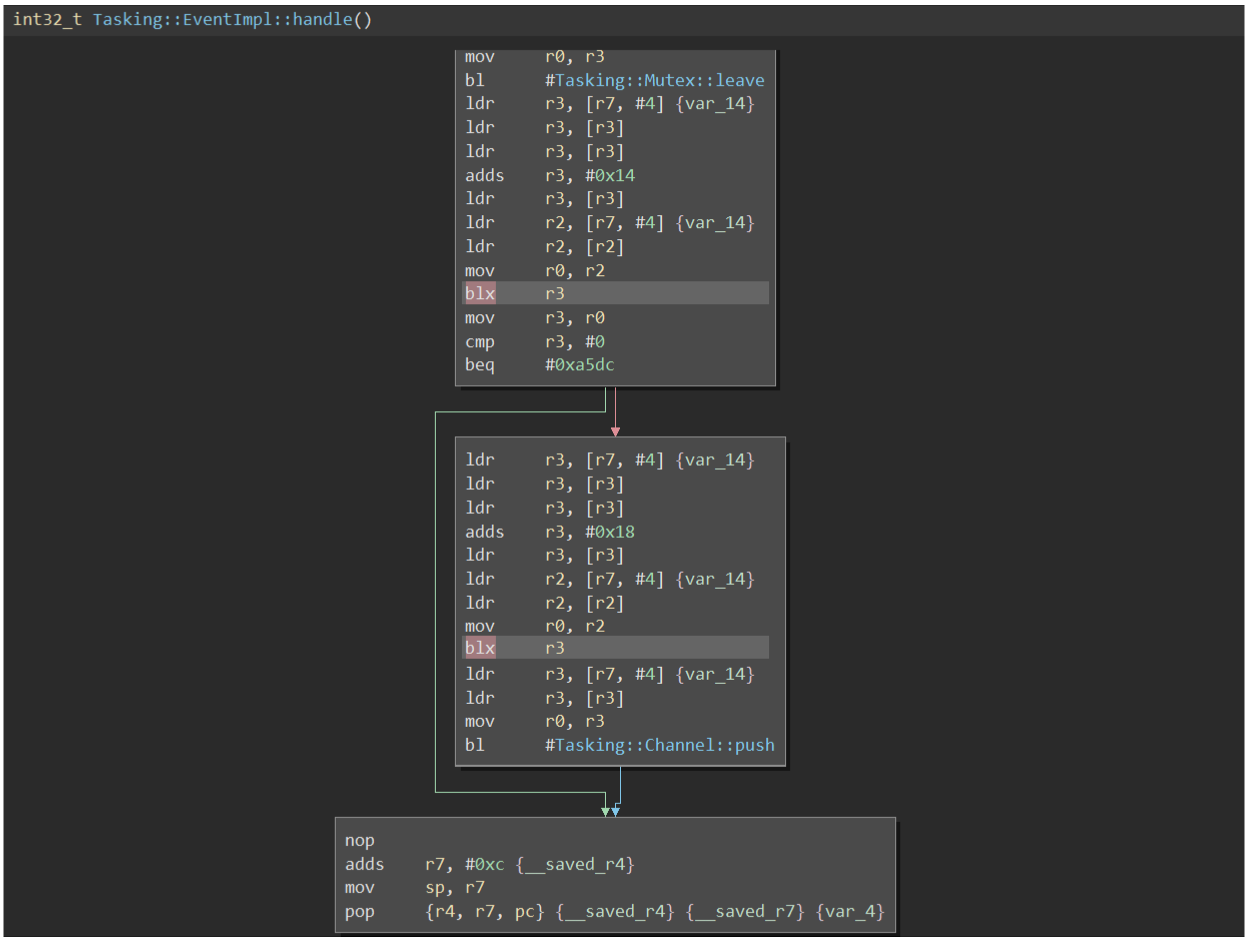
  The Tasking::EventImpl::handle function is responsible for the task-specific processing of a timed event by the Tasking Framework. Its source code is illustrated in Listing 17. The Tasking::EventImpl::handle function makes two virtual function calls in lines 21 and 23. Both the shallFire and onFire functions are implemented as virtual functions in the Event class implementation as shown in lines 186 and 193 in Listing C.4. The shallFire function is called when an event is planned to be handled by the Tasking Framework's scheduler. The onFire function is called to check if the scheduler is currently handling a task event.

  Figure 5.7 highlights the disassembly graph of the Tasking::EventImpl::handle function where both virtual function calls have been disassembled as indirect branching instructions.

  Table 5.5 shows that for the given input configuration, DEL was able to resolve the first indirect branching instruction that is related to the shallFire function call; however, the second indirect branch that is related to the onFire function call was not resolved. Since the if condition in line 21 was never met for the given input configuration, the second indirect branch was not visited during the dynamic run; hence, DEL did not resolve its target address.

```cpp
1   void Tasking::EventImpl::handle(void)
2   {
3       // If the event is periodic, the next wake-up time should hand over to the clock
4       mutex.enter();
5       if (periodical)
6       {
7           if (nullptr == periodicSchedule)
8           {
9               // No periodic schedule to play, jump to next period
10              // If trigger is called now clock are out of order.
11              clock.startAt(*this, (nextActivation_ms + period_ms));
12          }
13          else
14          {
15              // Play periodic schedule
16              periodicSchedule->pushTriggers();
17              clock.startAt(*this, periodicSchedule->stepToNextTriggerOffset());
18          }
19      }
20      mutex.leave();
21      if (parent.shallFire())
22      {
23          parent.onFire();
24          static_cast<UnprotectedChannelAccess&>(parent).push();
25      }
26  }
```

Listing 17: Tasking::EventImpl::handle function C++ source code.



```
int32_t Tasking::EventImpl::handle()

mov     r0, r3
bl      #Tasking::Mutex::leave
ldr     r3, [r7, #4] {var_14}
ldr     r3, [r3]
ldr     r3, [r3]
adds    r3, #0x14
ldr     r3, [r3]
ldr     r2, [r7, #4] {var_14}
ldr     r2, [r2]
mov     r0, r2
blx     r3
mov     r3, r0
cmp     r3, #0
beq     #0xa5dc

ldr     r3, [r7, #4] {var_14}
ldr     r3, [r3]
ldr     r3, [r3]
adds    r3, #0x18
ldr     r3, [r3]
ldr     r2, [r7, #4] {var_14}
ldr     r2, [r2]
mov     r0, r2
blx     r3
ldr     r3, [r7, #4] {var_14}
ldr     r3, [r3]
mov     r0, r3
bl      #Tasking::Channel::push

nop
adds    r7, #0xc {__saved_r4}
mov     sp, r7
pop     {r4, r7, pc} {__saved_r4} {__saved_r7} {var_4}
```

Figure 5.7: Tasking::TaskImpl::synchronizeStart function CFG.

Table 5.5: Tasking::EventImpl::handle function results.

Function	No. of detected indirect jumps	No. of visited indirect jumps	No. of resolved indirect jumps
Tasking::EventImpl::handle	2	1	1

- **Clock:**

  The Tasking::clock::isPending function checks whether the activation time of the clock queue head element is equal or smaller than the current time. In Listing 18, we can see in line 7 that the Tasking::clock::isPending function invokes the getTime function. The getTime function gets the absolute time used to control events. An application programmer can use this time for time stamps or for calculating the offset time of a periodic event. The getTime function is implemented as a virtual method for the Clock module, as seen in line 57 of Listing C.5.

  Figure 5.8 shows in the CFG of the Tasking::clock::isPending function that the virtual function call was disassembled as the single indirect branching instruction, blx r3.

Table 5.6 shows that for the given input configuration, DEL was able to successfully detect and resolve the single indirect jump present in the Tasking::clock::isPending function.

```cpp
bool Tasking::Clock::isPending(void) const
{
    timeQueueMutex.enter();
    bool pends = (queueHead != NULL);
    if (pends)
    {
        pends = (queueHead->nextActivation_ms <= getTime());
    }
    timeQueueMutex.leave();
}
```

Listing 18: Tasking::clock::isPending function C++ source code.

- **Group:**

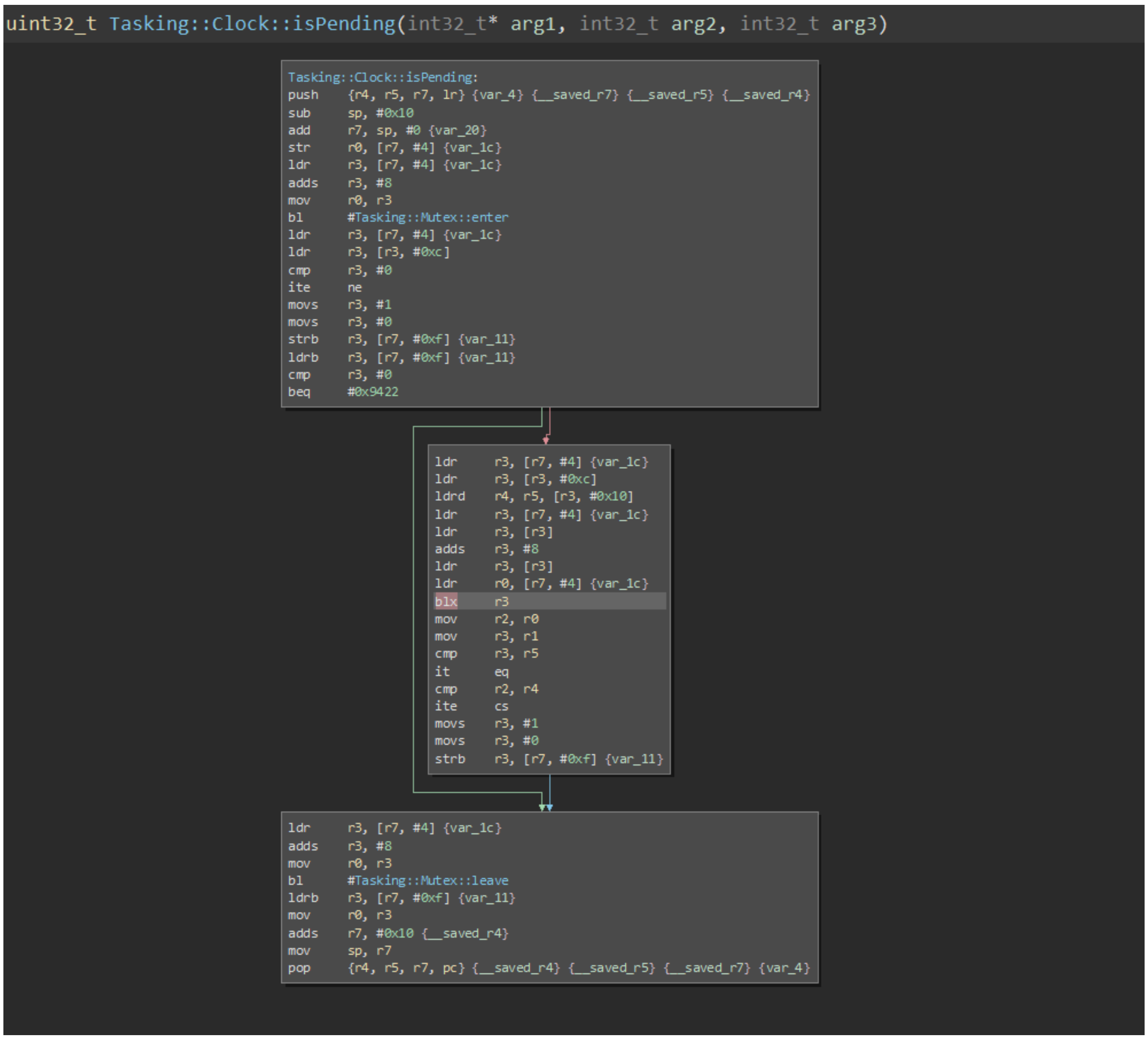
  The Tasking::GroupImpl::reset function call resets all associated tasks. Activated but not yet started threads will not be started after that call. In Listing 19, this function makes a virtual function call in line 6. Here, it calls the reset function for each task, resetting the activation state of all task inputs. The class Task implements the reset function as a virtual method as seen in line 158 in Listing C.6. This function is called whenever a task was executed by the associated scheduler or when the task belongs to a group where all tasks are executed.

  The CFG of the Tasking::GroupImpl::reset function in Figure 5.9 shows that the virtual function call was disassembled as the indirect branching instruction, blx r3.

  Table 5.7 shows that for the given input configuration, DEL was able to successfully detect and resolve the indirect jump of the Tasking::GroupImpl::reset function.

```cpp
1  void Tasking::GroupImpl::reset(void)
2  {
3      // Reset all tasks of the group;
4      for (unsigned int i = 0; (i < maxTasks) && (taskList[i] != NULL); i++)
5      {
6          taskList[i]->parent.reset();
7      }
8  }
```

Listing 19: Tasking::GroupImpl::reset function C++ source code.

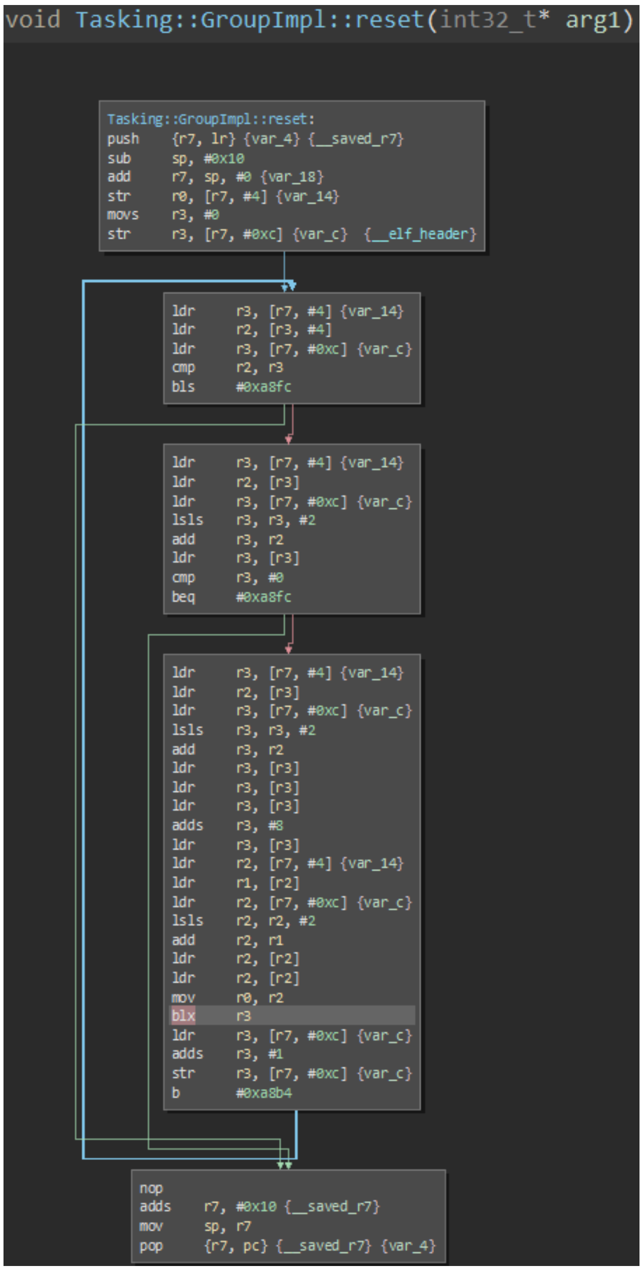


```
uint32_t Tasking::Clock::isPending(int32_t* arg1, int32_t arg2, int32_t arg3)

Tasking::Clock::isPending:
push    {r4, r5, r7, lr} {var_4} {__saved_r7} {__saved_r5} {__saved_r4}
sub     sp, #0x10
add     r7, sp, #0 {var_20}
str     r0, [r7, #4] {var_1c}
ldr     r3, [r7, #4] {var_1c}
adds    r3, #8
mov     r0, r3
bl      #Tasking::Mutex::enter
ldr     r3, [r7, #4] {var_1c}
ldr     r3, [r3, #0xc]
cmp     r3, #0
ite     ne
movs    r3, #1
movs    r3, #0
strb    r3, [r7, #0xf] {var_11}
ldrb    r3, [r7, #0xf] {var_11}
cmp     r3, #0
beq     #0x9422

ldr     r3, [r7, #4] {var_1c}
ldr     r3, [r3, #0xc]
ldrd    r4, r5, [r3, #0x10]
ldr     r3, [r7, #4] {var_1c}
ldr     r3, [r3]
adds    r3, #8
ldr     r3, [r3]
ldr     r0, [r7, #4] {var_1c}
blx     r3
mov     r2, r0
mov     r3, r1
cmp     r3, r5
it      eq
cmp     r2, r4
ite     cs
movs    r3, #1
movs    r3, #0
strb    r3, [r7, #0xf] {var_11}

ldr     r3, [r7, #4] {var_1c}
adds    r3, #8
mov     r0, r3
bl      #Tasking::Mutex::leave
ldrb    r3, [r7, #0xf] {var_11}
mov     r0, r3
adds    r7, #0x10 {__saved_r4}
mov     sp, r7
pop     {r4, r5, r7, pc} {__saved_r4} {__saved_r5} {__saved_r7} {var_4}
```

Figure 5.8: Tasking::clock::isPending function CFG.

Table 5.6: Tasking::clock::isPending function results.

Function	No. of detected indirect jumps	No. of visited indirect jumps	No. of resolved indirect jumps
Tasking::clock::isPending	1	1	1



```
void Tasking::GroupImpl::reset(int32_t* arg1)

Tasking::GroupImpl::reset:
push    {r7, lr} {var_4} {__saved_r7}
sub     sp, #0x10
add     r7, sp, #0 {var_18}
str     r0, [r7, #4] {var_14}
movs    r3, #0
str     r3, [r7, #0xc] {var_c}  {__elf_header}

ldr     r3, [r7, #4] {var_14}
ldr     r2, [r3, #4]
ldr     r3, [r7, #0xc] {var_c}
cmp     r2, r3
bls     #0xa8fc

ldr     r3, [r7, #4] {var_14}
ldr     r2, [r3]
ldr     r3, [r7, #0xc] {var_c}
lsls    r3, r3, #2
add     r3, r2
ldr     r3, [r3]
cmp     r3, #0
beq     #0xa8fc

ldr     r3, [r7, #4] {var_14}
ldr     r2, [r3]
ldr     r3, [r7, #0xc] {var_c}
lsls    r3, r3, #2
add     r3, r2
ldr     r3, [r3]
ldr     r3, [r3]
ldr     r3, [r3]
adds    r3, #8
ldr     r3, [r3]
ldr     r2, [r7, #4] {var_14}
ldr     r1, [r2]
ldr     r2, [r7, #0xc] {var_c}
lsls    r2, r2, #2
add     r2, r1
ldr     r2, [r2]
ldr     r2, [r2]
mov     r0, r2
blx     r3
ldr     r3, [r7, #0xc] {var_c}
adds    r3, #1
str     r3, [r7, #0xc] {var_c}
b       #0xa8b4

nop
adds    r7, #0x10 {__saved_r7}
mov     sp, r7
pop     {r7, pc} {__saved_r7} {var_4}
```

Figure 5.9: Tasking::GroupImpl::reset function CFG.

Table 5.7: Tasking::GroupImpl::reset function results.

Function	No. of detected indirect jumps	No. of visited indirect jumps	No. of resolved indirect jumps
Tasking::GroupImpl::reset	1	1	1

5.2.2.2 Overall Join fork case study results

Table 5.8 highlights DEL's results for the Join fork case study for the given input configuration. DEL resolved the target addresses of 26 out of 28 indirect jumps that were statically detected in the case study. The unresolved indirect control flow targets resulted primarily from the corresponding indirect jump instructions not being visited during DEL's dynamic run for the given input configuration. In other words, those instructions were not on the DSE path and, hence, their target addresses were not resolved. On the other hand, all the indirect jumps that were visited during the dynamic run had their target addresses resolved through the DSE.

Strictly speaking, using multiple input configurations that guarantee the execution of all the possible paths in the program should be sufficient to resolve all detected indirect jumps; however, the design of such input configurations is outside the scope of this thesis.

Table 5.8: Join fork case study overall results.

Function	No. of detected indirect jumps	No. of visited indirect jumps	No. of resolved indirect jumps
Tasking::InputArray::reset	1	1	1
Tasking::Input::synchronizeEnd	1	1	1
Tasking::Input::synchronizeStart	1	1	1
Tasking::Input::reset	1	1	1
Tasking::Scheduler::initialize	1	1	1
Tasking::Scheduler::start	1	1	1
Tasking::Scheduler::getTime	1	1	1
Tasking::Scheduler::terminate	2	2	2
Tasking::SchedulerImpl::perform	2	2	2
Tasking::SchedulerImpl::execute	1	1	1
Tasking::TaskImpl::synchronizeEnd	1	1	1
Tasking::TaskImpl::synchronizeStart	1	1	1
Tasking::TaskImpl::finalizeExecution	1	1	1
Tasking::Event::trigger	1	1	1
Tasking::Event::now	1	1	1
Tasking::EventImpl::configurePeriodicTiming	3	2	2
Tasking::EventImp::handle	2	1	1
Tasking::Clock::readFirstPending	1	1	1
Tasking::Clock::startAt	1	1	1
Tasking::Clock::startIn	1	1	1
Tasking::Clock::isPending	1	1	1
Tasking::GroupImpl::reset	1	1	1
Tasking::GroupImpl::reset	1	1	1

5.2.3 Performance

Table 5.9 shows the performance results during DEL's dynamic run of the Join fork example. Here, we can conclude that owing to the large size of the Join fork example binary, DEL needed more time and memory storage to lift it when compared to the example program used in Section 5.1. However, as discussed earlier, that is not a critical limitation as the tool will normally be operated offline amidst the design validation and verification phase.

Table 5.9: Performance results.

Use-Case	Binary Size (Kbyte)	%CPU	Average Memory (MiB)	Execution Time (sec)
Join fork example of the Tasking Framework	630	37%	819	2213

5.2.4 Bounding loops

Through its DSE engine, DEL can detect and bound loops accurately. DEL keeps track of the number of times instructions are visited during the dynamic run. This feature could be used to detect and bound loops in an input binary accurately. It could be particularly useful when performing a WCET analysis at the IR level of a program similar to the work done by [10].

5.2.5 Limitations

- DEL's translation APIs only cover ARMv7-M ISA instructions that were present in the Tasking Framework's Join fork example case study. Such instructions resemble only a subset of the ARMv7-M ISA (67%).
- Our suggested method cannot handle parallel executing threads that have resources shared between them. Because the symbolic execution strategy applied in our method only explores one control-flow path at a time in the program being analyzed, it is difficult to anticipate the behavior of parallel executing binaries similar to the issue mentioned by [10].

Chapter 6

Discussion

6.1 Conclusion

In this thesis, we presented a new lifter that lifts given binaries to LLIR and applies static and dynamic execution, attempting to recover the control flow of the provided software fully. The lifter, DEL, first performs a static symbolic execution to formulate each indirect jump's control flow target as a Z3 expression. Secondly, it performs a dynamic symbolic execution using the Z3 SMT solver to resolve all the Z3 expressions generated to their concrete values. DEL implements its memory and register models and a condition flags handler to facilitate the dynamic symbolic execution. DEL showed high precision when resolving indirect control flow targets for a case study developed based on the Tasking Framework. According to our experimental results, given the required input configurations, our proposed method is pragmatic and capable of constructing an upgraded intermediate representation of C++ based applications.

This work considers ARMv7-M ISA. The time frame of the Master's thesis was not enough to fully cover the entire ISA. The presented work covers about 60%. Full coverage and different ISAs are left for future work. ARMv7-M ISA has been chosen among many other ISAs because it is commonly used in embedded systems.

DEL lifts the given binaries to static single assignment LLVM instructions. We aim to use the lifted LLIR to apply different code analyses for safety and security purposes. LLVM can help us reach our goal because of its broad support.

The work showed the power of symbolic execution but at the cost of run-time and memory requirements of the developed lifter. The relatively straightforward translation from static single assignment expressions to Z3 expressions is an essential motivation to use symbolic executions. However, many points need to be resolved to improve the capabilities of symbolic execution, such as memory aliasing and multi-threading.

6.2 Future Work

Testing the tool's performance while running real-life programs other than the Tasking Framework is crucial for comprehensively evaluating a binary lifting tool as DEL. SPEC CPU2006 benchmark suite [9], which is typical in the binary lifting literature [14, 16, 23, 40] can be an appropriate benchmark to evaluate DEL's ability to resolve indirect control flow targets. This benchmark suite includes CPU-bound benchmarks, giving a cynical view of run-time overheads. As already mentioned, DEL's translation APIs only cover a subset of the ARMv7-M present in the Tasking Framework's Join fork example case study. Consequently, we leave for future work the implementation of additional translation APIs for covering the remaining assembly instructions for ARMv7-M ISA that were not present in our case study. Only then a comprehensive evaluation of DEL's abilities against the SPEC CPU2006, and other similar performance benchmarks would be possible.

Another paramount future step is introducing parallel-execution SMT solver threads that carry out symbolic execution for architectures that utilize parallel threads.

Appendix A

The Tasking Framework's Join fork example

Listing A.1: The Tasking Framework's Join fork example.

```
/*
 * joinForkExample.cpp
 *
 * Copyright 2012-2020 German Aerospace Center (DLR) SC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *    http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

/*
 * This example
 */
#include <schedulerProvider.h>
#include <schedulePolicyFifo.h>
#include <taskChannel.h>
#include <taskEvent.h>
#include <task.h>
class ImgChannel : public Tasking::Channel
{
public:
    const int& getValue(void) const;
    void pushValue(int);
protected:
    int imgValue = 10;
};
const int&
ImgChannel::getValue(void) const
{
    return imgValue;
}
void ImgChannel::pushValue(int value)
{
    imgValue = value;
    Channel::push();
}
class CamTask : public Tasking::TaskProvider<1u, Tasking::SchedulePolicyFifo>
{
public:
    CamTask(Tasking::Scheduler& scheduler, ImgChannel& outChannel);
    virtual void execute(void);
private:
    ImgChannel& out;
};
CamTask::CamTask(Tasking::Scheduler& scheduler, ImgChannel& outChannel):
    TaskProvider(scheduler),
    out(outChannel)
{
    inputs[0u].configure(1u);
}
void CamTask::execute(void)
{
    int imgValue = 10;
    out.pushValue(imgValue);
}
class CraterTask : public Tasking::TaskProvider<1u, Tasking::SchedulePolicyFifo>
{
public:
    CraterTask(Tasking::Scheduler& scheduler, ImgChannel& craterChannel);
    virtual void execute(void);
private:
    ImgChannel& out;
};
CraterTask::CraterTask(Tasking::Scheduler& scheduler, ImgChannel& craterChannel):
    TaskProvider(scheduler),
    out(craterChannel)
{
    inputs[0u].configure(2u);
}
void CraterTask::execute(void)
{
    int imgValue = getChannel<ImgChannel>(0u)->getValue() + 10;
    out.pushValue(imgValue);
}
class FeatureTask : public Tasking::TaskProvider<1u, Tasking::SchedulePolicyFifo>
{
public:
    FeatureTask(Tasking::Scheduler& scheduler, ImgChannel& featureChannel);
    virtual void execute(void);
private:
    ImgChannel& out;
};
FeatureTask::FeatureTask(Tasking::Scheduler& scheduler, ImgChannel& featureChannel):
    TaskProvider(scheduler),
    out(featureChannel)
{
    inputs[0u].configure(2u);
}
void FeatureTask::execute(void)
{
    int imgValue = getChannel<ImgChannel>(0u)->getValue() + 5;
    out.pushValue(imgValue);
}
class NavigationFilter : public Tasking::TaskProvider<3u,
Tasking::SchedulePolicyFifo>
{
public:
    NavigationFilter(Tasking::Scheduler& scheduler, ImgChannel& featureChannel);
    virtual void execute(void);
private:
    ImgChannel& out;
};
NavigationFilter::NavigationFilter(Tasking::Scheduler& scheduler,
ImgChannel& outChannel):
    TaskProvider(scheduler),
    out(outChannel)
{
    inputs[0u].configure(0u);
    inputs[1u].configure(0u);
    inputs[2u].configure(1u,true);
}
void NavigationFilter::execute(void)
{
    int imgValue = getChannel<ImgChannel>(0u)->getValue() +
    getChannel<ImgChannel>(1u)->getValue();
    out.pushValue(imgValue);
}
class TerminalTask : public Tasking::TaskProvider<1u, Tasking::SchedulePolicyFifo>
{
public:
    TerminalTask(Tasking::Scheduler& scheduler);
    virtual void execute(void);
    int val = 0;
};
TerminalTask::TerminalTask(Tasking::Scheduler& scheduler):
    TaskProvider(scheduler)
{
    inputs[0u].configure(0u);
}
void TerminalTask::execute(void)
{
    val += getChannel<ImgChannel>(0u)->getValue() + 1;
}
// <<<<<<== instances ==>>>>>
Tasking::SchedulerProvider<1u, Tasking::SchedulePolicyFifo> scheduler;
ImgChannel imgChannel10;
ImgChannel imgChannel45;
ImgChannel craterPos;
ImgChannel featurePos;
ImgChannel outPos;
Tasking::Event inputTrigger(scheduler);
Tasking::Event processTrigger(scheduler);
CamTask camTask1(scheduler, imgChannel10);
CamTask camTask2(scheduler, imgChannel45);
CraterTask craterTask0(scheduler, craterPos);
FeatureTask featureTask0(scheduler, featurePos);
NavigationFilter navTask0(scheduler, outPos);
TerminalTask terminalTask1(scheduler);
TerminalTask terminalTask2(scheduler);
// <<<<<< == program code == >>>>>
int main(void)
{
    // Connect tasks to input channels
    camTask1.configureInput(0u, inputTrigger);
    camTask2.configureInput(0u, inputTrigger);
    craterTask0.configureInput(0u,imgChannel10);
    featureTask0.configureInput(0u,imgChannel45);
    navTask0.configureInput(0u, craterPos);
    navTask0.configureInput(1u, featurePos);
    navTask0.configureInput(2u, processTrigger);
    terminalTask1.configureInput(0u, outPos);
    terminalTask2.configureInput(0u, outPos);
    // Set periods
    inputTrigger.setPeriodicTiming(500, 1000u);
    processTrigger.setPeriodicTiming(850, 100u);
    // Start Tasking scheduler
    scheduler.start();
    while(terminalTask1.val<73){
       }
    // Stop Tasking scheduler
    scheduler.terminate(true);
    return 0;
}
```

Appendix B

Example DSEIR module

Listing B.1: Final lifted DSEIR module of the example program.

```
define void @"foo_1;"(i32 %0, i32 %1) {
"0":
  %memory_address_to_store_in = alloca i32, align 4
  %2 = load i32, i32* %SP, align 4
  %3 = sub i32 %2, 4
  store i32 %3, i32* %memory_address_to_store_in, align 4
  %4 = load i32, i32* %memory_address_to_store_in, align 4
  %5 = load i32, i32* %FP, align 4
  %memory_cell = alloca i32, align 4
  store i32 0, i32* %memory_cell, align 4
  store i32 %5, i32* %memory_cell, align 4
  %6 = load i32, i32* %SP, align 4
  %7 = sub i32 %6, 4
  store i32 %7, i32* %SP, align 4
  %8 = load i32, i32* %SP, align 4
  store i32 0, i32* %IMM, align 4
  %9 = add i32 %8, 0
  store i32 %9, i32* %FP, align 4
  store i32 5, i32* %IMM, align 4
  %10 = load i32, i32* %IMM, align 4
  store i32 %10, i32* %R3, align 4
  %11 = load i32, i32* %R3, align 4
  store i32 %11, i32* %R0, align 4
  %12 = load i32, i32* %FP, align 4
  store i32 0, i32* %IMM, align 4
  %13 = add i32 %12, 0
  store i32 %13, i32* %SP, align 4
  %memory_address_to_load_from = alloca i32, align 4
  %14 = load i32, i32* %SP, align 4
  store i32 %14, i32* %memory_address_to_load_from, align 4
  %15 = load i32, i32* %memory_address_to_load_from, align 4
  store i32 0, i32* %memory_cell, align 4
  %DATA = load i32, i32* %memory_cell, align 4
  store i32 %DATA, i32* %FP, align 4
  %16 = load i32, i32* %SP, align 4
  %17 = add i32 %16, 4
  store i32 %17, i32* %SP, align 4
  %18 = icmp eq i32 0, 0
  br i1 %18, label %"9"
}
define void @"foo_2;"(i32 %0, i32 %1) {
"0":
  %memory_address_to_store_in = alloca i32, align 4
  %2 = load i32, i32* %SP, align 4
  %3 = sub i32 %2, 4
  store i32 %3, i32* %memory_address_to_store_in, align 4
  %4 = load i32, i32* %memory_address_to_store_in, align 4
  %5 = load i32, i32* %FP, align 4
  store i32 0, i32* %memory_cell, align 4
  store i32 %5, i32* %memory_cell, align 4
  %6 = load i32, i32* %SP, align 4
  %7 = sub i32 %6, 4
  store i32 %7, i32* %SP, align 4
  %8 = load i32, i32* %SP, align 4
  store i32 0, i32* %IMM, align 4
  %9 = add i32 %8, 0
  store i32 %9, i32* %FP, align 4
  store i32 6, i32* %IMM, align 4
  %10 = load i32, i32* %IMM, align 4
  store i32 %10, i32* %R3, align 4
  %11 = load i32, i32* %R3, align 4
  store i32 %11, i32* %R0, align 4
  %12 = load i32, i32* %FP, align 4
  store i32 0, i32* %IMM, align 4
  %13 = add i32 %12, 0
  store i32 %13, i32* %SP, align 4
  %memory_address_to_load_from = alloca i32, align 4
  %14 = load i32, i32* %SP, align 4
  store i32 %14, i32* %memory_address_to_load_from, align 4
  %15 = load i32, i32* %memory_address_to_load_from, align 4
  store i32 0, i32* %memory_cell, align 4
  %DATA = load i32, i32* %memory_cell, align 4
  store i32 %DATA, i32* %FP, align 4
  %16 = load i32, i32* %SP, align 4
  %17 = add i32 %16, 4
  store i32 %17, i32* %SP, align 4
  %18 = icmp eq i32 0, 0
  br i1 %18, label %"9"
}
define void @"main;"(i32 %0, i32 %1) {
"0":
```

```
%memory_address_to_store_in = alloca i32, align 4
%2 = load i32, i32* %SP, align 4
%3 = sub i32 %2, 4
store i32 %3, i32* %memory_address_to_store_in, align 4
%4 = load i32, i32* %memory_address_to_store_in, align 4
%5 = load i32, i32* %FP, align 4
store i32 0, i32* %memory_cell, align 4
store i32 %5, i32* %memory_cell, align 4
%6 = load i32, i32* %SP, align 4
%7 = sub i32 %6, 4
store i32 %7, i32* %SP, align 4
%memory_address_to_store_in1 = alloca i32, align 4
%8 = load i32, i32* %SP, align 4
%9 = sub i32 %8, 4
store i32 %9, i32* %memory_address_to_store_in1, align 4
%10 = load i32, i32* %memory_address_to_store_in1, align 4
%11 = load i32, i32* %LR, align 4
store i32 0, i32* %memory_cell, align 4
store i32 %11, i32* %memory_cell, align 4
%12 = load i32, i32* %SP, align 4
%13 = sub i32 %12, 4
store i32 %13, i32* %SP, align 4
%14 = load i32, i32* %SP, align 4
store i32 4, i32* %IMM, align 4
%15 = add i32 %14, 4
store i32 %15, i32* %FP, align 4
%16 = load i32, i32* %SP, align 4
store i32 24, i32* %IMM, align 4
%17 = sub i32 %16, 24
store i32 %17, i32* %SP, align 4
%memory_address_to_store_in2 = alloca i32, align 4
%pre_index = alloca i32, align 4
%18 = load i32, i32* %FP, align 4
store i32 %18, i32* %pre_index, align 4
%19 = load i32, i32* %pre_index, align 4
%20 = add i32 %19, -24
store i32 %20, i32* %memory_address_to_store_in2, align 4
%21 = load i32, i32* %R0, align 4
%22 = load i32, i32* %memory_address_to_store_in2, align 4
store i32 0, i32* %memory_cell, align 4
store i32 %21, i32* %memory_cell, align 4
%memory_address_to_store_in3 = alloca i32, align 4
%pre_index4 = alloca i32, align 4
%23 = load i32, i32* %FP, align 4
store i32 %23, i32* %pre_index4, align 4
%24 = load i32, i32* %pre_index4, align 4
%25 = add i32 %24, -28
store i32 %25, i32* %memory_address_to_store_in3, align 4
%26 = load i32, i32* %R1, align 4
```

```
  %27 = load i32, i32* %memory_address_to_store_in3, align 4
  store i32 0, i32* %memory_cell, align 4
  store i32 %26, i32* %memory_cell, align 4
  %28 = load i32, i32* %FP, align 4
  store i32 20, i32* %IMM, align 4
  %29 = sub i32 %28, 20
  store i32 %29, i32* %R3, align 4
  store i32 0, i32* %IMM, align 4
  %30 = load i32, i32* %IMM, align 4
  store i32 %30, i32* %R3, align 4
  %memory_address_to_store_in5 = alloca i32, align 4
  %pre_index6 = alloca i32, align 4
  %31 = load i32, i32* %FP, align 4
  store i32 %31, i32* %pre_index6, align 4
  %32 = load i32, i32* %pre_index6, align 4
  %33 = add i32 %32, -8
  store i32 %33, i32* %memory_address_to_store_in5, align 4
  %34 = load i32, i32* %R3, align 4
  %35 = load i32, i32* %memory_address_to_store_in5, align 4
  store i32 0, i32* %memory_cell, align 4
  store i32 %34, i32* %memory_cell, align 4
  store i32 0, i32* %IMM, align 4
  %36 = load i32, i32* %IMM, align 4
  store i32 %36, i32* %R3, align 4
  %memory_address_to_store_in7 = alloca i32, align 4
  %pre_index8 = alloca i32, align 4
  %37 = load i32, i32* %FP, align 4
  store i32 %37, i32* %pre_index8, align 4
  %38 = load i32, i32* %pre_index8, align 4
  %39 = add i32 %38, -12
  store i32 %39, i32* %memory_address_to_store_in7, align 4
  %40 = load i32, i32* %R3, align 4
  %41 = load i32, i32* %memory_address_to_store_in7, align 4
  store i32 0, i32* %memory_cell, align 4
  store i32 %40, i32* %memory_cell, align 4
  %42 = icmp eq i32 0, 0
  br i1 %42, label %"5"

"1":                ; preds = %"5"
  %43 = load i32, i32* %R3, align 4
  store i32 4, i32* %IMM, align 4
  %44 = add i32 %43, 4
  store i32 %44, i32* %R3, align 4
  %45 = load i32, i32* %R2, align 4
  %46 = load i32, i32* %R3, align 4
  %47 = add i32 %45, %46
  store i32 %47, i32* %R3, align 4
  %memory_address_to_load_from = alloca i32, align 4
  %48 = load i32, i32* %R3, align 4
  store i32 %48, i32* %memory_address_to_load_from , align 4
  %49 = load i32 , i32* %memory_address_to_load_from , align 4
  %temp1 = alloca i32 , align 4
  store i32 0, i32* %memory_cell , align 4
  %Full_DATA = load i32 , i32* %memory_cell , align 4
  store i32 %Full_DATA, i32* %temp1, align 4
  %v_0 = alloca i32 , align 4
  %Full_DATA9 = load i32 , i32* %temp1, align 4
  %v_010 = shl i32 %Full_DATA9, %I_246
  store i32 %v_010 , i32* %v_0 , align 4
  %50 = load i32 , i32* %v_0 , align 4
  %v_011 = lshr i32 %50, %I_246
  store i32 %v_011 , i32* %v_0 , align 4
  %51 = load i32 , i32* %v_0 , align 4
  store i32 %51, i32* %R3, align 4
  %52 = load i32 , i32* %R3, align 4
  store i32 48, i32* %IMM, align 4
  %53 = sub i32 %52, 48
  store i32 %53, i32* %TEMP, align 4
  %54 = icmp eq i32 0, 0
  br i1 %54, label %"7", label %"2"

"2":                ; preds = %"1"
  %55 = load i32 , i32* %R3, align 4
  store i32 4, i32* %IMM, align 4
  %56 = add i32 %55, 4
  store i32 %56, i32* %R3, align 4
  %57 = load i32 , i32* %R2, align 4
  %58 = load i32 , i32* %R3, align 4
  %59 = add i32 %57, %58
  store i32 %59, i32* %R3, align 4
  %memory_address_to_load_from12 = alloca i32 , align 4
  %60 = load i32 , i32* %R3, align 4
  store i32 %60, i32* %memory_address_to_load_from12 , align 4
  %61 = load i32 , i32* %memory_address_to_load_from12 , align 4
  %temp113 = alloca i32 , align 4
  store i32 0, i32* %memory_cell , align 4
  %Full_DATA14 = load i32 , i32* %memory_cell , align 4
  store i32 %Full_DATA14, i32* %temp113, align 4
  %v_015 = alloca i32 , align 4
  %Full_DATA16 = load i32 , i32* %temp113, align 4
  %v_017 = shl i32 %Full_DATA16, %I_246
  store i32 %v_017 , i32* %v_015 , align 4
  %62 = load i32 , i32* %v_015 , align 4
  %v_018 = lshr i32 %62, %I_246
  store i32 %v_018 , i32* %v_015 , align 4
  %63 = load i32 , i32* %v_015 , align 4
  store i32 %63, i32* %R3, align 4
  %64 = load i32 , i32* %R3, align 4
  store i32 90, i32* %IMM, align 4
  %65 = sub i32 %64, 90
  store i32 %65, i32* %TEMP, align 4
  %66 = icmp eq i32 0, 0
  br i1 %66, label %"4", label %"3"

"3":                  ; preds = %"2"
  %67 = load i32, i32* %R3, align 4
  store i32 1, i32* %IMM, align 4
  %68 = xor i32 %67, 1
  store i32 %68, i32* %R3, align 4
  %memory_address_to_store_in19 = alloca i32, align 4
  %pre_index20 = alloca i32, align 4
  %69 = load i32, i32* %FP, align 4
  store i32 %69, i32* %pre_index20, align 4
  %70 = load i32, i32* %pre_index20, align 4
  %71 = add i32 %70, -8
  store i32 %71, i32* %memory_address_to_store_in19, align 4
  %72 = load i32, i32* %R3, align 4
  %73 = load i32, i32* %memory_address_to_store_in19, align 4
  store i32 0, i32* %memory_cell, align 4
  store i32 %72, i32* %memory_cell, align 4

"4":                  ; preds = %"2"
  %74 = load i32, i32* %R3, align 4
  store i32 1, i32* %IMM, align 4
  %75 = add i32 %74, 1
  store i32 %75, i32* %R3, align 4
  %memory_address_to_store_in21 = alloca i32, align 4
  %pre_index22 = alloca i32, align 4
  %76 = load i32, i32* %FP, align 4
  store i32 %76, i32* %pre_index22, align 4
  %77 = load i32, i32* %pre_index22, align 4
  %78 = add i32 %77, -12
  store i32 %78, i32* %memory_address_to_store_in21, align 4
  %79 = load i32, i32* %R3, align 4
  %80 = load i32, i32* %memory_address_to_store_in21, align 4
  store i32 0, i32* %memory_cell, align 4
  store i32 %79, i32* %memory_cell, align 4

"5":                  ; preds = %"0"
  %81 = load i32, i32* %R3, align 4
  store i32 99, i32* %IMM, align 4
  %82 = sub i32 %81, 99
  store i32 %82, i32* %TEMP, align 4
  %83 = icmp eq i32 0, 0
  br i1 %83, label %"1", label %"6"

"6":                  ; preds = %"5"
```

```
  %84 = icmp eq i32 0, 0
  br i1 %84, label %"8"

"7":                ; preds = %"1"

"8":                ; preds = %"6"
  store i32 2, i32* %IMM, align 4
  %85 = load i32, i32* %R3, align 4
  store i32 2, i32* %IMM, align 4
  %86 = shl i32 %85, 2
  store i32 %86, i32* %R3, align 4
  %87 = load i32, i32* %FP, align 4
  store i32 4, i32* %IMM, align 4
  %88 = sub i32 %87, 4
  store i32 %88, i32* %R2, align 4
  %89 = load i32, i32* %R2, align 4
  %90 = load i32, i32* %R3, align 4
  %91 = add i32 %89, %90
  store i32 %91, i32* %R3, align 4
  %92 = load i32, i32* %PC, align 4
  store i32 %92, i32* %LR, align 4
  %93 = icmp eq i32 0, 0
  br i1 %93, label %"0", label %"9"

"9":                ; preds = %"8", %"0", %"0"
  %94 = load i32, i32* %R0, align 4
  store i32 %94, i32* %R3, align 4
  %95 = load i32, i32* %R3, align 4
  store i32 %95, i32* %R0, align 4
  %96 = load i32, i32* %FP, align 4
  store i32 4, i32* %IMM, align 4
  %97 = sub i32 %96, 4
  store i32 %97, i32* %SP, align 4
  %memory_address_to_load_from23 = alloca i32, align 4
  %98 = load i32, i32* %SP, align 4
  store i32 %98, i32* %memory_address_to_load_from23, align 4
  %99 = load i32, i32* %memory_address_to_load_from23, align 4
  store i32 0, i32* %memory_cell, align 4
  %DATA = load i32, i32* %memory_cell, align 4
  store i32 %DATA, i32* %LR, align 4
  %100 = load i32, i32* %SP, align 4
  %101 = add i32 %100, 4
  store i32 %101, i32* %SP, align 4
  %memory_address_to_load_from24 = alloca i32, align 4
  %102 = load i32, i32* %SP, align 4
  store i32 %102, i32* %memory_address_to_load_from24, align 4
  %103 = load i32, i32* %memory_address_to_load_from24, align 4
  store i32 0, i32* %memory_cell, align 4
  %DATA25 = load i32, i32* %memory_cell, align 4
  store i32 %DATA25, i32* %FP, align 4
  %104 = load i32, i32* %SP, align 4
  %105 = add i32 %104, 4
  store i32 %105, i32* %SP, align 4
  br i32 1, label %"lr"
}
```

Appendix C

Relevant Tasking Framework header files

Listing C.1: Task input header file.

```
/*
 * taskInput.h
 *
 * Copyright 2012-2019 German Aerospace Center (DLR) SC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *   http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
#ifndef TASKINPUT_H_
#define TASKINPUT_H_
#include "impl/taskInput_impl.h"
namespace Tasking
{
class Task;
class Channel;
/**
 * Manage the activation state of incoming channels to a task.
 If all task inputs of a task are activated
 * or at least one is activated and marked as final, the task will execute.
 * A task input is activated, if the number of activations reaches
 the activation threshold defined
 * by the constructor of the task input. Defining a task input with
 activation threshold of zero
 * means, that the input is only optional for a task and will not
 block task activation by other inputs.
 */
class Input
{
public:
    /**
     * Null initialization of a task input.
     */
    Input(void);
    /**
     * Destructor
     */
    virtual ~Input(void)
    {
    }
    /**
     * Connect the input to a channel and configure the behavior for
     the activation of the input.
     * Without this call, the input is invalid and an application
     can not start. As side effect
     * the input is configured as synchronous input.
     To get an unsynchronized input a call to
     * method setSynchron with parameter false is necessary.
     *
     * @param channel Reference to the channel where this input is associated to.
     *
     * @param activations Threshold value of new data
     notifications at channel to activate the task.
     * Default value is one incoming message to trigger a task.
     A value of 0 mark the task input
     * optional for the accepting tasks.
     *
     * @param final Flag to indicate that reaching the
     activation threshold activate the task
     * immediately without respect to other activation
     states of other inputs from the task.
     * Default value is false.
     */
    void configure(Channel& channel, unsigned int activations = 1,
    bool final = false);
    /**
     * Configure the settings of the input without setting a
     channel to the input. The input remains
     * invalid until a channel is associated to the input.
     As side effect the input is configured as
     * synchronous input. To get an unsynchronized input a
     call to method setSynchron with parameter
     * false is necessary.
     *
     * @param activations Threshold value of incoming messages
     on a channel to activate the task.
     * Default value is one incoming message to trigger a task.
     A value of 0 mark the task input
     * optional for the accepting tasks.
     *
     * @param final Flag to indicate that reaching the activation
     threshold triggers the task
     * immediately without respect to other activation states of
     other inputs from the task.
     * Default value is false.
     *
     * @see associate
     */
    void configure(unsigned int activations, bool final = false);
    /**
     * Configure input synchronization as on. If synchronization
     is on and the input is activated, the reset operation
     * consumes only the number of expected activations.
     No notifications are lost when the input is activated and the
     * reset operation is not executed for this activation cycle.
     If enough notifications have been received when the
     * reset operation is started, the input get's immediately
     activated directly after the reset operation.
     * E.g. if activations is set to two and five notifications happens without
     * the reset operation, the input is activated directly
     again by the reset operation. After the next
     * reset operation the input will wait for a further
     notification to get activated.
     *
     * By default the synchronization is switched on.
     *
     * @param syncState Setting for the synchronization state.
     If set to false notifications will be lost after the
     * activation of the input and before its reset operation is finalized
     An associated channel can hold in this
     * case unread data items and the associated task
     has to handle these circumstance.
     */
    void setSynchron(bool syncState = true);
    /**
     * Connect a channel to the input. If the input is configured,
     it becomes valid after the call.
     *
     * @param channel Reference to the message where this
     input is associated to.
```

```
     *
     * @result true if the association succeed. false if the
     input is already associated to the channel.
     *
     * @see configure
     */
    bool associate(Channel& channel);
    /**
     * Remove the association between the input and the channel.
     The input is no longer notified by the channel
     * and can not be activated until a new association to a channel is set.
     The input becomes invalid
     * after the call.
     */
    void deassociate(void);
    /**
     * Connect the input with a task. By usage of a TaskProvider
     the method is called by the constructor.
     * The method is also called when instantiating a task or
     connect an input array to a task. By default
     * from application code no call is necessary.
     *
     * @see Task::construct
     * @see InputArray::connectTask
     */
    void connectTask(TaskImpl& task);
    /**
     * Reset the activation state to 0 activations.
     */
    virtual void reset(void);
    /**
     * Request if the task input is notified the expected
     number of times since the last reset.
     *
     * @result True, if the task input is activated. False
     if not the required number of notification
     * happens. For optional and final inputs the result is
     false if not at least one notification happens.
     */
    bool isActivated(void) const;
    /**
     * Check if the input is marked as final.
     *
     * @result True, if the input is marked as final.
     False if not.
     */
    bool isFinal(void) const;
    /**
     * Check if the input is configured as optional
     *
     * @result True if the input is configured with zero
     arrival as activation threshold, else false.
     */
    bool isOptional(void) const;
    /**
     * True if the input is correctly configured.
     * @see configure
     */
    bool isValid(void) const;
    /**
     * Request the number of activations since last
     reset of the task input.
     * Special case: optional final input returns
     only true if an activation came
     *
     * @result Number of activations since last call to reset.
     */
    unsigned int getActivations(void) const;
    /**
     * Type safe request of a channel from a task input
     * @tparam ChannelType Type of the channel to request
     * @result Pointer of corresponding task channel
     type associated with the this input
     */
    template<typename ChannelType>
    ChannelType*
    getChannel(void) const
    {
        return static_cast<ChannelType*>(impl.getChannel());
    }
protected:
    /**
     * The associated task start to execute. This method
     is protected by the scheduler against concurrent
     * access of two tasks associated with the scheduler.
     */
    virtual void synchronizeStart(void);
    /**
     * The associated task has finalize its run.
     This method is protected by the scheduler against concurrent
     * access of two tasks associated with the scheduler.
     */
    virtual void synchronizeEnd(void);
private:
    /// Implementation part of the input
    InputImpl impl;
};
} // namespace Tasking
```

```
#endif /* TASKINPUT_H_ */
```

Listing C.2: Scheduler policy header file.

```
/*
 * schedulePolicy.h
 *
 * Copyright 2012-2019 German Aerospace Center (DLR) SC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *   http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
#ifndef TASKING_INCLUDE_SCHEDULEPOLICY_H_
#define TASKING_INCLUDE_SCHEDULEPOLICY_H_
namespace Tasking
{
class TaskImpl;
/**
 * Interface class of a scheduling policy. For the implementation of a new
 scheduling policy the two structures
 * and two methods have to be implemented by a specialization of this class.
 */
class SchedulePolicy
{
public:
    /**
     * Structure to initialize policies with settings for a task,
     e.g. the task priority for a priority based
     * scheduling policy. A specialization of this class has to
     provide the corresponding structure when
     * task settings are needed for the policy. It is used to
     initialize the management data of a task.
     * @see ManagementData
     */
    struct Settings
    {
    };
    /**
     * Structure for data used by the implementation. This data is held by each task.
     Typical data are for
     * example pointers between tasks to implement a run queue.
```

```
         It is initialized with task settings.
         * A specialization of a scheduling policy has to provide this data structure.
         * @see Settings
         */
        struct ManagementData
        {
        };
        /// Needed for virtual methods
        virtual ~SchedulePolicy(void)
        {
        }
        /**
         * Queue a task according to the policy into the run queue.
         An implementation of a scheduling policy must implement
         * this method. Each task provides the management data structure to
         provide the memory space for the scheduling
         * policy. The method is called when a task switches the state from
         wait to pending.
         * @param task Reference to the task to queue in the run queue by
         the scheduling policy
         * @return True when queue was empty at call time.
         * @see ManagementData
         */
        virtual bool queue(Tasking::TaskImpl& task) = 0;
        /**
         * Request and remove the next task in the scheduling order.
         An implementation of a scheduling policy has to provide
         * this method. The delivered task will switch from state
         pending to run.
         * @return Pointer to the next task in the order of
         the scheduling policy.
         If no pending task is available, a NULL
         * pointer is returned.
         */
        virtual Tasking::TaskImpl* nextTask(void) = 0;
};
} // namespace Tasking
#endif /* TASKING_INCLUDE_SCHEDULEPOLICY_H_ */
```

Listing C.3: Scheduler header file.

```
/*
 * scheduler.h
 *
 * Copyright 2012-2019 German Aerospace Center (DLR) SC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *    http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
#ifndef TASKING_INCLUDE_SCHEDULER_H_
#define TASKING_INCLUDE_SCHEDULER_H_
#include "impl/scheduler_impl.h"
namespace Tasking
{
// Forward name declarations
class TaskImpl;
/**
 * Common interface to the scheduler used by the Tasking Framework elements.
   It is recommended to use the template
 * class SchedulerProvider to instantiate a scheduler.
 * @see SchedulerProvider
 */
class Scheduler
{
public:
    /**
     * Initialize the scheduler.
     *
     * @param schedulePolicy Reference to the used scheduling policy
     for the scheduler.
     * @param clock Reference to the clock used by the scheduler implementation
     */
    Scheduler(SchedulePolicy& schedulePolicy, Clock& clock);
    /// Virtual destructor of interface
    virtual ~Scheduler(void);
    /**
     * Set a zero time with an offset time to the current time when
     the function is called.
     By default a zero time
     * is set at construction time of the scheduler without offset,
     but for synchronization issues the clock can
     * adjusted to an outer signal from time to time.
     *
     * If the system is currently running,
     adjusting the clock will have an effect on the start time of all events,
     * because all time points to start an event in the clock queue
     are organized by absolute time points.
     *
     * The bare metal implementation has to implement this functionality.
     *
     * @param offset Offset time to the current time. Using the
     current time of the clock will have nearly no effect
     * to the timing.
     */
    virtual void setZeroTime(Time offset) = 0;
    /**
     * Start the scheduling of tasks.
     *
     * @param doReset If set to true, a reset on all associated tasks is performed.
     If set to false, each activated task
     * will be queued for execution.
     *
     * @see terminate
     */
    void start(bool doReset = true);
    /**
     * Stopping the scheduling of tasks. The scheduler didn't
     accept tasks to perform until start is called.
     *
     * @param doNotRemovePendingTasks If the flag is set to false,
     after stop acceptance of task activations is stopped,
     * pending tasks in the run queue are removed. Currently running
     tasks will not terminated by this call.
     *
     * @see start
     */
    void terminate(bool doNotRemovePendingTasks = false);
    /**
     * Call initialize method of all associated tasks of the scheduler.
     A task is associated to a task when it
     * is constructed with a reference to the scheduler instance.
     */
    void initialize(void);
    /**
     * Get the absolute time used to control events. The zero time
     depends on the bare metal implementation. Application
     * programmer can use this time for time stamps or to calculate
     the offset time of a periodic event.
     *
     * @result Time which is in the time frame used for triggering events in ms.
     Most of the time, zero time is start
     * of the system.
     *
     * @see Event::setPeriodicTiming
     * @see setZeroTime
     */
    Time getTime(void) const;
protected:
    /**
```

```
     * Pure abstract method which must be implemented by
     the bare metal implementation of the scheduler.
     * The method implementation shall wake up one of
     the executors of the scheduler instance.
     The method is called
     * whenever a new task should perform and the run queue
     is empty or an event is fired by the clock.
     */
    virtual void signal(void) = 0;
    /**
     * A call to the method waits until the run queue of the scheduler runs empty.
     If pending tasks activate other tasks
     * also this task will be executed before waitUntilEmpty returns.
     The bare metal model has to implement these
     * functionality to enable a safe termination of the Tasking Framework.
     */
    virtual void waitUntilEmpty(void) = 0;
    /**
     * @return Reference to the implementation part of the scheduler.
     */
    SchedulerImpl& getImpl(void);
private:
    SchedulerImpl impl;
};
} // namespace Tasking
// ————————— inlines —————————
inline Tasking::Time
Tasking::Scheduler::getTime() const
{
    return impl.clock.getTime();
}
inline Tasking::SchedulerImpl&
Tasking::Scheduler::getImpl(void)
{
    return impl;
}
#endif /* TASKING_INCLUDE_SCHEDULER_H_ */
```

Listing C.4: Task event header file.

```
/*
 * taskEvent.h
 *
 * Copyright 2012-2019 German Aerospace Center (DLR) SC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *   http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
#ifndef TASKEVENT_H_
#define TASKEVENT_H_
#include "impl/taskEvent_impl.h"
namespace Tasking
{
// Forward definition of periodic schedule
class PeriodicSchedule;
/**
 * The task event is a timed event. The behavior of the event can be
 periodically or relative to the
 * call of the method reset.
 *
 * The implementation specializes the class Channel with
 timing functionalities.
 An application programmer can
 * specialize the task event by overriding the two methods
 shallFire and onFire
 with own functionalities.
 *
 * @see TaskChannel
 */
class Event : public Channel
{
public:
    /**
     * @param scheduler Reference to the scheduler responsible to execute the event.
     * @param eventId Identifier for this channel.
     *
     * NOTE:
     *    It is the responsibility of the user to ensure uniqueness of the channel
     and events identifications.
     */
    explicit Event(Scheduler& scheduler, ChannelId eventId = 0);
    /**
     * @param scheduler Reference to the scheduler responsible
     to execute the event.
     * @param eventName Null-terminated string specifying a name
     for this event.
     The name will be
     *                  truncated after 4 characters.
     */
    explicit Event(Scheduler& scheduler, const char* eventName);
    /*
     * Destructor of the task event
     */
    ~Event(void);
    /**
     * Set the timing of event to a fix periodic behavior.
     Call this method only:
     from a constructor, when
     * the scheduler is initializing, or when the timer is stopped.
     *
     * @param period Period time in case of a periodical clock.
     A period of zero will
     lead to a single shot with
     * an absolute time
     *
     * @param offset Offset of the start time of the system.
     If the offset is in the past,
     the method computes
     * the next time point in the future by adding a multiple of
     the period to the offset.
     For a single shot with
     * period zero this event is fired immediately.
     */
    void setPeriodicTiming(const Time period, const Time offset);
    /**
     * Set the timing of event to play schedule of periodic triggers.
     Call this method only:
     from a constructor,
     * when the scheduler is initializing, or when the timer is stopped.
     * In this configuration this event itself will not notify an
     associated task input,
     only the periodic triggers in
     * the periodic schedule notifies associated task inputs. To
     change this behavior,
     the method shallFire can be
     * overridden.
     *
     * @param period Period time in case of a periodical clock.
     If the trigger time of
     the first periodic trigger in
     * the periodic schedule is not within the given period,
     the event is not started to
     play the periodic schedule.
     *
     * @param offset Offset of the start time of the system. If the
     offset is in the past,
     the method computes
     * the next time point in the future by adding a multiple of the
     period to the offset.
     *
     * @param schedule Reference to the schedule of periodic triggers
     to play by the event.
     If triggers are in the
     * schedule with an bigger offset than the period of the event,
     these triggers will not fired.
     *
     * @see shallFire
     */
     void setPeriodicSchedule(const Time period, const Time offset,
     PeriodicSchedule& schedule);
    /**
     * Set the timing of the event relative to the reset operation.
     A call to reset
     will trigger the task event
     * for the next activation. To start the relative timing a call to
     the reset operation
     is necessary. Keep in
     * mind that a reset restarts the timer, when the event is
     connected to several tasks
     or a final input is
     * connected to the task.
     *
     * @param delay Delay time in milliseconds which is used as
     trigger time relative
     to the reset operation.
     */
    void setRelativeTiming(const Time delay);
    /**
     * Trigger the event out of order. When the event is configured
     to periodic or
     relative timing the call of
     * the method has no effect, until the periodic or
     relative timing is stopped.
     An event can be only triggered
     * once. If it is queued by the clock, the event is
     removed from the clock
     before it is queued again. This
     * means reset operations on connected tasks will
     stop the event timer, e.g.
     when the event is connected to
     * several tasks or anconnected task with an input
     configured as final.
     *
     * @param time Offset time in ms when the event
     is triggered out of order.
     This can use to trigger an
     * task after a specified time to another task.
     *
     * @see setPeriodicTiming
     * @see setRelativTiming
     */
    void trigger(Time time = 0);
    /// @return True, when the clock is still queued for
    triggering at the clock.
    bool isTriggered(void) const;
    /**
     * Remove the task event from the list of time events
     in the clock. The event
     will not fire until a new
     * timing is programmed to the task event.
     */
    void stop(void);
    /**
     * Reset the task event. In case of a relative timing
     this method starts the
     timer and calls the
     * reset method of the overridden channel.
     */
    void reset(void) override;
    /**
     * The method is called when the event is handled.
     * @result By default true, so long no periodic schedule
     is played by the event.
     If the method or an override
     * return false, the associated input is not notified.
     */
    virtual bool shallFire(void);
    /**
     * The method is called every time the task event is
     handled by the schedule.
     The method can be overridden by
     * by the application software. By default it does nothing.
     */
    virtual void onFire(void);
    /**
     * @result Current time of the associated scheduler.
     */
    Tasking::Time now(void) const;
private:
    /// Structure for implementation
    EventImpl impl;
};
} // namespace Tasking
#endif /* TASKEVENT_H_ */
```

Listing C.5: Task clock header file.

```
/*
```

```
 * clock_impl.h
 *
 * Copyright 2012-2019 German Aerospace Center (DLR) SC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *   http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
#ifndef TASKING_INCLUDE_CLOCK_H_
#define TASKING_INCLUDE_CLOCK_H_
#include "../taskEvent.h"
#include "../taskUtils.h"
namespace Tasking
{
class Scheduler;
/**
 * Base class to manage the start of events at a time point.
 It must be overloaded with a system specific clock
 * mechanism which trigger the scheduler for the execution
  of events at a specific time.
 */
class Clock
{
public:
    /**
     * Initialization of the clock and connect it to scheduler
     *
     * @param scheduler Reference to the schedule which should
     wake up in case of a clock event.
     */
    Clock(Scheduler& scheduler);
    /// Destructor
    virtual ~Clock(void);
    /**
     * Get the absolute time used to control events. The zero time
     depends on the bare metal implementation.
     * The method must be implemented by the bare metal implementation
     of the clock. Application programmer
     * can use this time for time stamps or to calculate the offset
     time of a periodic event.
     *
     * @result Time which is in the time frame used for triggering
     events in ms. Most of the time zero time is start
     * of the system.
     *
     * @see Event::setPeriodicTiming
     */
    virtual Time getTime(void) const = 0;
    /// @return True when no event is in the clock queue
    bool isEmtpy(void) const;
    /// @return True when activation time of the clock queue head
    element is equal or smaller than the current time.
    bool isPending(void) const;
    /**
     * Start an event at an absolute time.
     *
     * @param p_event Reference to the event to start at an absolute time
     * @param time Absolute time in ms when the event should started.
     Time zero depends on the bare metal
     * implementation. By default it should be the instantiation time of this class.
     */
    void startAt(EventImpl& p_event, const Time time);
    /**
     * Start an event at a relative time span from now.
     *
     * @param p_event Reference to the event to start at the relative time
     * @param time Relative time span from now in ms in which the
     event should started.
     */
    void startIn(EventImpl& p_event, const Time time);
    /**
     *  Enqueue an element to the clock queue. The method search the
     right position in the queue by the time,
     *  earliest time first. The last enqueued event is triggered first.
     *
     *  @param event Reference to the element to enqueue
     *
     *  @return True when the head element is replaced by the enqueued
     element, else false.
     */
    bool enqueue(EventImpl& event);
    /**
     * Replace directly the head of the clock queue without searching
     the correct spot. This is done with events
     * which has a delay time with zero or smaller.
     *
     * @param event Reference to the element which becomes the
     new head of the queue.
     */
    void enqueueHead(EventImpl& event);
    /**
     *  Dequeue an element from the queue.
     *  This method is used if an event is deleted to satisfy that
     the event will not triggered in the future. Such
     *  a trigger can lead into a memory corruption.
     *
     *  @param event Event to dequeue from the list.
     */
    void dequeue(EventImpl& event);
    /**
     * Remove all events from the clock queue.
     */
    void dequeueAll(void);
    /**
     * Stop a running timer and start the timer to wake up the system
     after a time span is over.
     * The method must override by the bare metal implementation
     *
     * @param timeSpan Length of the time interval. When the
     time interval pass, the system should wake up and trigger
     * the scheduler to handle pending time events.
     */
    virtual void startTimer(Time timeSpan) = 0;
    /**
     * Read and remove the first pending element from the clock queue.
     *
     * @return Pointer to the from the clock queue removed head element.
     */
    EventImpl* readFirstPending(void);
    /**
     *  @return The time between head of the clock queue and the
     next different time point in the clock queue.
     * If there is no further time point in the clock queue or the
     clock queue is empty, the method return 0.
     */
    Time getNextGapTime(void) const;
    /**
     * @return Wake up time point of the clock queue head. If the
     clock queue is empty, the method return 0.
     */
    Time getHeadTime(void) const;
    /// Reference to the scheduler, which execute events from this
    clock implementation.
    Scheduler& scheduler;
    /// Mutex to protect the clock queue against concurrent access.
    mutable Mutex timeQueueMutex;
    /// Flag to indicate if still in mutex.
    bool inTimeQueueMutex;
    /// Mutex to protect change of pair timeQueueMutex and inTimeQueueMutex.
    mutable Mutex timeQueueMutexMutex;
    /**
     * Pointer to the clock queue head. This event has the earliest
     absolute wake up time or the same time like an
     * event with the same time queued first.
     */
    EventImpl* queueHead;
    /**
     * Pointer to the clock queue tail. This event has the highest
     absolute wake up time or an equal time to the
     * event enqueued after.
     */
    EventImpl* queueTail;
};
} // namespace Tasking
#endif /* TASKING_INCLUDE_CLOCK_H_ */
```

Listing C.6: Task header file.

```
/*
 * task.h
 *
 * Copyright 2012-2019 German Aerospace Center (DLR) SC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *   http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
#ifndef TASK_H_
#define TASK_H_
#include "impl/task_Impl.h"
#include "taskUtils.h"
namespace Tasking
{
/**
 * A task performs a single execution if all inputs of the
 input array are activated or one input
 * marked as final is activated. To implement the body of
 the task, the method execute has
 * to be overridden. To simplify creating of a task with
 all its inputs the template class
 * TaskProvider exists, which provides an instance of the
 input array for all incoming inputs of
 * the task.
 *
 * The purpose of this class is the reactive and concurrent
 processing on incoming events or data
 * packages. For example a task implementation can be the
 reaction on an interrupt distributed by
 * an interrupt channel, the classification of an incoming
 message on a channel, or a further
 * computation step in a sequence of computation tasks.
 *
 * For a correct operation it is necessary to configure the
 class correctly. This means the inputs
 * are configured with the expected settings and connected to
 a channel and the inputs in the input
 * array are connected to this task by a call of the method
 construct or by using the template
 * class TaskProvider instead of Task directly.
 *
 * To combine several tasks to a group, the tasks should bind
 to a group with the class Group.
 * By default each task is scheduled without relationships to
 other task, which means that the
 * method reset is called directly after the task is executed
 and its inputs are synchronized. If
 * the task is bind to a group reset is called only when all
 tasks associated to a group are marked
 * as executed. By this, a subsequent activation can only
 happen when all tasks of the group are
 * executed.
 *
 * Each task has an identifier which shall unique. It can
 be _either_ a numeric id or a name of up to
 * 4 characters in length. Only use the respective setter/getter methods.
 *
 * @see TaskProvider
 * @see Group
 * @see InputArray
 * @see Event
 * @see Channel
 */
class Task
{
protected:
    /**
     * The identification of the task. It should always
     mapped to the first data member to find the
     * identification easy in a memory dump.
     */
    TaskId m_taskId;
public:
    /**
     * First initialization step and connect the task to the
     scheduler. The task is not fully
     * initialized until the second initialization step with a
     call to construct is done.
     *
     * @param scheduler Reference to the scheduler. It provide
     means to execute this task.
     *
     * @param policy Reference to the data structure needed for
     management of the task by the scheduler.
     *
     * @param inputs Reference to an array of inputs associated with this task.
     *
     * @param taskId Identification of the task. This identification
     is needed by extensions of the
     *                   Tasking framework to address a task or to identify
     the task for debugging. If not given,
     *                   an identification the number of constructor calls
     is given as identification.
     *
     * NOTE:
     *     It is the responsibility of the user to ensure uniqueness
     of the task id.
     *
     * @see taskId_t
     * @see construct
     */
    Task(Scheduler& scheduler, SchedulePolicy::ManagementData& policy,
    InputArray& inputs, TaskId taskId = 0u);
    /**
     * First initialization of task with a task name. The task
     is not fully initialized until the
     * second initialization step with a call to construct is done.
     *
     * @param scheduler Reference to the scheduler which
     performs this task.
     *
     * @param policy Reference to the data structure needed
     for management of the task by the scheduler.
     *
     * @param inputs Reference to an array of inputs associated with this task.
     *
     * @param taskName Null-terminated string specifying
     a name for this task. The name will be
     *                       truncated after 4 characters.
     Only a name _or_ a taskId can be used for
```

```
         *                        channel identification.
         * @see construct
         */
        Task(Scheduler& scheduler, SchedulePolicy::ManagementData& policy,
        InputArray& inputs, const char* taskName);
        /// Destructor needed by virtual methods
        virtual ~Task();
        /**
         * Connect a channel to an input of the task.
         *
         * @param key Identifications of the input which should connect to the channel.
         *
         * @param channel Reference to the channel to connect.
         The channel should have the type the task expect.
         *
         * @result true if the configuration of the input to the
         channel succeed. false if an error during the configuration
         * happened.
         */
        bool configureInput(unsigned int key, Channel& channel);
        /// @result True if all inputs are configured and connected to a channel.
        bool isValid(void) const;
        /**
         * A call resets the activation state of all task inputs.
         This method is called whenever a task was executed
         * by the associated scheduler or when the task belongs to
         a group all tasks of the group are executed.
         */
        virtual void reset(void);
        /**
         * Enquire the identification of a task
         *
         * @result The identification of type taskId_t for the task.
         *
         * NOTE:
         *     If a task name was assigned to this task the id
         *     will represent the numeric value of the
         4-character string.
         *
         * @see taskId_t
         * @see convertTaskIdToString
         */
        TaskId getTaskId(void) const;
        /**
         * Set a new name for a task
         *
         * @param newTaskName Null-terminated string specifying
         the new name
         *                        which will be set for the task.
```

```
     *                    The name will be truncated
     after 4 characters.
     *
     */
    void setTaskName(const char* newTaskName);
    /**
     * Set a new ID for a task
     * @param newTaskId The new ID which will be set for the task.
     *
     * @see taskId_t
     */
    void setTaskId(TaskId newTaskId);
    /**
     * Joining the task to a task group. The method is called
     by the group on calling join with a reference
     * to the task instance. The method should never use by
     an application software.
     *
     * @param p_group Reference to the task group.
     *
     * @result Reference to the implementation part of the task.
     * @see Group::join
     */
    TaskImpl& joinTo(GroupImpl& p_group);
protected:
    /**
     * Second initialization step of construction using the input array
     from outside the class task. The method is
     * called by the constructor of the specialized class to connect
     the task with the inputs. If the template class
     * TaskProvider is used, which is the preferred way to set up a task,
     will call this method in the constructor.
     */
    void construct(void);
    /**
     * Pure virtual entrance point for the processing of the task.
     An implementation of a task should override this
     * method with the task specific processing.
     */
    virtual void execute(void) = 0;
    /**
     * Initialize the task. This step is performed by calling the
     initialize method of the associated scheduler.
     * The method can override by the application programmer with
     further initialization steps.
     *
     * @see Scheduler::initialize
     */
    virtual void initialize(void);
    /**
     * Request the associated channel pointer connected to an input.
     This call simplify the cast to the corresponding
     * channel type.
     *
     * @tparam channelType Expected type of the channel.
     *
     * @param key Key to identify the input to request the channel.
     *
     * @return Pointer to the associated channel at input
     with the key or null pointer if input is not connected to any
     *             channel.
     */
    template<typename channelType>
    channelType* getChannel(unsigned int key) const;
private:
    /// Forbid copy constructor
    Task(Task&);
    /// Implementation specific structure of task
    TaskImpl impl;
};
/**
 * Helper template to simplify set up of a task.
 *
 * @tparam numberOfInputs Number of inputs for the task
 *
 * @tparam Policy Scheduling policy type
 */
template<unsigned int numberOfInputs, class Policy>
class TaskProvider : public Task
{
public:
    /**
     * Constructor for a task with identification number
     *
     * @param scheduler Reference to the scheduler
     which performs this task.
     *
     * @param taskId Specify the ID number for a specific task.
     *
     * NOTE:
     *     It is the responsibility of the user to ensure
     uniqueness of the task id.
     * @see taskId_t
     */
    TaskProvider(Scheduler& scheduler, TaskId taskId = 0u);
    /**
     * Constructor for a task with identification number and a
     scheduling policy with settings.
```

```
         *
         * @param scheduler Reference to the scheduler which performs this task.
         *
         * @param settings Initial settings on the task for the scheduling policy.
         *
         * @param taskId Specify the identification for a specific task.
         *
         * NOTE:
         *     It is the responsibility of the user to ensure uniqueness
         of the task id.
         * @see taskId_t
         */
        TaskProvider(Scheduler& scheduler, typename Policy::Settings settings,
        TaskId taskId = 0u);
        /**
         * Constructor for a task with a name.
         *
         * @param scheduler Reference to the scheduler which performs this task.
         *
         * @param taskName Null-terminated string specifying a name for
         this task. The name will be
         *                          truncated after 4 characters. Only a name
         _or_ a taskId can be used for
         *                          channel identification.
         *
         * NOTE:
         *     It is the responsibility of the user to ensure uniqueness
         of the task id.
         * @see taskId_t
         */
        TaskProvider(Scheduler& scheduler, const char* taskName);
        /**
         * Constructor for a task with a name.
         *
         * @param scheduler Reference to the scheduler which performs
         this task.
         *
         * @param settings Initial settings on the task for the scheduling
         policy.
         *
         * @param taskName Null-terminated string specifying a name for
         this task. The name will be
         *                          truncated after 4 characters. Only a name
         _or_ a taskId can be used for
         *                          channel identification.
         */
        TaskProvider(Scheduler& scheduler, typename Policy::Settings settings
        , const char* taskName);
    protected:
```

```
    /// Inputs of the task
    InputArrayProvider<numberOfInputs> inputs;
    /// Policy data of the task.
    typename Policy::ManagementData policyData; // Typename is needed
    to see the management
    data of the specified policy
};
// ========= implementation part ==========
template<typename channelType>
channelType*
Task::getChannel(unsigned int key) const
{
    return impl.inputs[key].getChannel<channelType>();
}
// ----------------
template<unsigned int numberOfInputs, class Policy>
TaskProvider<numberOfInputs, Policy>::TaskProvider(Scheduler& _scheduler,
TaskId taskId) :
    Task(_scheduler, policyData, inputs, taskId)
{
    Task::construct();
}
template<unsigned int numberOfInputs, class Policy>
TaskProvider<numberOfInputs, Policy>::TaskProvider(Scheduler& _scheduler,
typename Policy::Settings settings,
                                                    TaskId taskId) :
    Task(_scheduler, policyData, inputs, taskId), policyData(settings)
{
    Task::construct();
}
template<unsigned int numberOfInputs, class Policy>
TaskProvider<numberOfInputs, Policy>::TaskProvider(Scheduler& _scheduler,
const char* taskName) :
    TaskProvider(_scheduler, getTaskIdFromName(taskName))
{
}
template<unsigned int numberOfInputs, class Policy>
TaskProvider<numberOfInputs, Policy>::TaskProvider(Scheduler& _scheduler,
typename Policy::Settings settings,
                                                    const char* taskName) :
    TaskProvider(_scheduler, settings, getTaskIdFromName(taskName))
{
}
} // namespace Tasking

#endif /* TASK_H_ */
```

Bibliography

[1] `https://ecss.nl/standard/ecss-q-st-40c-rev-1-safety-15-february-2017/` (cit. on p. 13).

[2] `https://github.com/lifting-bits/mcsema` (cit. on p. 14).

[3] `https://llvm.org/` (cit. on p. 15).

[4] `http://datarescue.com/idabase/` (cit. on p. 19).

[5] `https://sourceware.org/binutils/docs/binutils/objdump.html` (cit. on p. 20).

[6] `https://github.com/TRDDC-TUM/wcet-benchmarks/tree/master/benchmarks/crc` (cit. on pp. 21, 22, 24).

[7] `https://developer.arm.com/documentation/ddi0403/eb/` (cit. on pp. 26, 52).

[8] `https://github.com/LuaDist/libjpeg/blob/master/example.c` (cit. on p. 28).

[9] `https://www.spec.org/cpu2006/` (cit. on p. 77).

[10] Hazem Abaza. *Worst-Case Execution Time Analysis for C++ based Real-Time On-Board Software Systems.* Master's thesis. 2021 (cit. on pp. 13, 23, 29, 31–37, 39, 42–47, 55–57, 64, 76).

[11] Alfred V Aho et al. *Compilers: principles, techniques, & tools.* Pearson Education India, 2007 (cit. on p. 18).

[12] Frances E. Allen and John Cocke. "A program data flow analysis procedure". In: *Communications of the ACM* 19.3 (1976), p. 137 (cit. on p. 13).

[13] Anil Altinay. "Dynamic Binary Lifting and Recompilation". PhD thesis. UC Irvine, 2020 (cit. on p. 28).

[14] Anil Altinay et al. "BinRec: dynamic binary lifting and recompilation". In: *Proceedings of the Fifteenth European Conference on Computer Systems.* 2020, pp. 1–16 (cit. on pp. 14, 26–28, 77).

[15] 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. 42).

[16] Kapil Anand et al. "A compiler-level intermediate representation based binary analysis and rewriting system". In: *Proceedings of the 8th ACM European Conference on Computer Systems.* 2013, pp. 295–308 (cit. on p. 77).

[17] Bertrand Anckaert, Mariusz Jakubowski, and Ramarathnam Venkatesan. "Proteus: virtualization for diversified tamper-resistance". In: *Proceedings of the ACM workshop on Digital rights management.* 2006, pp. 47–58 (cit. on p. 27).

[18] Dennis Andriesse et al. “An in-depth analysis of disassembly on full-scale x86/x64 binaries”. In: *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 2016, pp. 583–600 (cit. on p. 27).

[19] Andrew W Appel. *Modern compiler implementation in C*. Cambridge university press, 2004 (cit. on p. 46).

[20] Arvind Ayyangar. *Static disassembly of stripped binaries*. State University of New York at Stony Brook, 2010 (cit. on pp. 19–21).

[21] Gogul Balakrishnan and Thomas Reps. “Analyzing memory accesses in x86 executables”. In: *International conference on compiler construction*. Springer. 2004, pp. 5–23 (cit. on p. 23).

[22] Gogul Balakrishnan and Thomas Reps. “WYSINWYX: What you see is not what you eXecute”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 32.6 (2010), pp. 1–84 (cit. on p. 23).

[23] Erick Bauman, Zhiqiang Lin, Kevin W Hamlen, et al. “Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics.” In: *NDSS*. 2018 (cit. on p. 77).

[24] Fabrice Bellard. “QEMU, a fast and portable dynamic translator.” In: *USENIX annual technical conference, FREENIX Track*. Vol. 41. Califor-nia, USA. 2005, p. 46 (cit. on pp. 19, 29).

[25] Aaron R Bradley and Zohar Manna. *The calculus of computation: decision procedures with applications to verification*. Springer Science & Business Media, 2007 (cit. on p. 45).

[26] David Brumley et al. “BAP: A binary analysis platform”. In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 463–469 (cit. on p. 29).

[27] Robert Brummayer. *Efficient SMT solving for bit-vectors and the extensional theory of arrays*. Trauner, 2010 (cit. on p. 45).

[28] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. “Detecting self-mutating malware using control-flow graph matching”. In: *International conference on detection of intrusions and malware, and vulnerability assessment*. Springer. 2006, pp. 129–143 (cit. on p. 14).

[29] 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. 45).

[30] Pavel ČADEK. “Symbolic Loop Bound Analysis”. PhD thesis. Masarykova univerzita, Fakulta informatiky, 2015 (cit. on p. 42).

[31] 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 p. 14).

[32] Eric Cheng. “Binary Analysis and Symbolic Execution with angr”. PhD thesis. PhD thesis, The MITRE Corporation, 2016 (cit. on p. 59).

[33] Brian Chess and Gary McGraw. “Static analysis for security”. In: *IEEE security & privacy* 2.6 (2004), pp. 76–79 (cit. on p. 13).

[34] *codevis*. `https://www.oreans.com/codevirtualizer.php` (cit. on p. 27).

[35] Christian Collberg, Clark Thomborson, and Douglas Low. *A taxonomy of obfuscating transformations*. Tech. rep. Citeseer, 1997 (cit. on p. 27).

[36] Christian Collberg, Clark Thomborson, and Douglas Low. “Manufacturing cheap, resilient, and stealthy opaque constructs”. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1998, pp. 184–196 (cit. on p. 27).

[37] 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. 15, 40, 44, 46).

[38] Giuseppe Desoli et al. “Deli: A new run-time control point”. In: *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002.(MICRO-35). Proceedings*. IEEE. 2002, pp. 257–268 (cit. on p. 17).

[39] Alessandro Di Federico and Giovanni Agosta. “A jump-target identification method for multi-architecture static binary translation”. In: *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. 2016, pp. 1–10 (cit. on pp. 27, 29).

[40] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. “rev. ng: a unified binary analysis framework to recover CFGs and function boundaries”. In: *Proceedings of the 26th International Conference on Compiler Construction*. 2017, pp. 131–141 (cit. on pp. 19, 27, 29, 77).

[41] Michael D Ernst. *Static and dynamic analysis: Synergy and duality*. 2003 (cit. on p. 18).

[42] James H Fetzer. “Program verification: The very idea”. In: *Communications of the ACM* 31.9 (1988), pp. 1048–1063 (cit. on p. 14).

[43] Ian Foster. *Designing and building parallel programs: concepts and tools for parallel software engineering*. Addison-Wesley Longman Publishing Co., Inc., 1995 (cit. on p. 32).

[44] Neville Grech and Yannis Smaragdakis. “P/Taint: unified points-to and taint analysis”. In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (2017), pp. 1–28 (cit. on p. 14).

[45] 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. 13, 32, 33, 35).

[46] R. Nigel Horspool and Nenad Marovac. “An approach to the problem of detranslation of computer programs”. In: *The Computer Journal* 23.3 (1980), pp. 223–229 (cit. on p. 26).

[47] Giacomo Iadarola et al. “Call graph and model checking for fine-grained android malicious behaviour detection”. In: *Applied Sciences* 10.22 (2020), p. 7975 (cit. on p. 14).

[48] M Irlbeck et al. “Deconstructing dynamic symbolic execution”. In: *Dependable Software Systems Engineering* 40 (2015), p. 26 (cit. on pp. 42, 43).

[49] Suman Jana et al. “Automatically detecting error handling bugs using error specifications”. In: *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 2016, pp. 345–362 (cit. on p. 14).

[50] Donald B Johnson. “Efficient algorithms for shortest paths in sparse networks”. In: *Journal of the ACM (JACM)* 24.1 (1977), pp. 1–13 (cit. on p. 57).

[51] 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. 53).

[52] Soomin Kim et al. “Testing intermediate representations for binary analysis”. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2017, pp. 353–364 (cit. on pp. 18, 19).

[53] James C King. “Symbolic execution and program testing”. In: *Communications of the ACM* 19.7 (1976), pp. 385–394 (cit. on p. 14).

[54] Mark H Klein and Thomas Ralya. *An analysis of input/output paradigms for real-time systems.* Tech. rep. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 1990 (cit. on p. 13).

[55] Christopher Kruegel et al. "Static disassembly of obfuscated binaries". In: *USENIX security Symposium.* Vol. 13. 2004, pp. 18–18 (cit. on p. 19).

[56] Jian Lin et al. "A value set analysis refinement approach based on conditional merging and lazy constraint solving". In: *IEEE Access* 7 (2019), pp. 114593–114606 (cit. on p. 23).

[57] Cullen Linn and Saumya Debray. "Obfuscation of executable code to improve resistance to static disassembly". In: *Proceedings of the 10th ACM conference on Computer and communications security.* 2003, pp. 290–299 (cit. on pp. 20, 21).

[58] Chi-Keung Luk et al. "Pin: building customized program analysis tools with dynamic instrumentation". In: *Acm sigplan notices* 40.6 (2005), pp. 190–200 (cit. on p. 20).

[59] 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. 35).

[60] Olaf Maibaum, Daniel Lüdtke, and Andreas Gerndt. "Tasking framework: parallelization of computations in onboard control systems". In: (2013) (cit. on pp. 31–33).

[61] Nicholas Nethercote. *Dynamic binary analysis and instrumentation.* Tech. rep. University of Cambridge, Computer Laboratory, 2004 (cit. on p. 17).

[62] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *ACM Sigplan notices* 42.6 (2007), pp. 89–100 (cit. on pp. 18, 20).

[63] Matija Novak, Mike Joy, and Dragutin Kermek. "Source-code similarity detection and detection tools used in academia: a systematic review". In: *ACM Transactions on Computing Education (TOCE)* 19.3 (2019), pp. 1–37 (cit. on p. 14).

[64] Richard J Orgass. "McCarthy J.. Towards a mathematical science of computation. Information processing 1962, Proceedings ofIFIP 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. 45).

[65] Tobias Pfeffer et al. "Efficient and safe control flow recovery using a restricted intermediate language". In: *2018 IEEE 27th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE).* IEEE. 2018, pp. 235–240 (cit. on pp. 59, 60).

[66] David A Ramos and Dawson Engler. "Under-constrained symbolic execution: Correctness checking for real code". In: *24th {USENIX} Security Symposium ({USENIX} Security 15).* 2015, pp. 49–64 (cit. on p. 30).

[67] Eicke-Alexander Risse et al. "Guidance, Navigation and Control for Autonomous Close-Range-Rendezvous". In: (2020) (cit. on p. 35).

[68] Kurt Schwenk and Daniel Herschmann. “On-board data analysis and real-time information system”. In: (2020) (cit. on p. 35).

[69] Yan Shoshitaishvili et al. “Sok:(state of) the art of war: Offensive techniques in binary analysis”. In: *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2016, pp. 138–157 (cit. on pp. 59, 60).

[70] Xujie Si et al. “Learning loop invariants for program verification”. In: *Neural Information Processing Systems*. 2018 (cit. on p. 14).

[71] Dawn Song et al. “BitBlaze: A new approach to computer security via binary analysis”. In: *International conference on information systems security*. Springer. 2008, pp. 1–25 (cit. on p. 48).

[72] Johannes Späth, Karim Ali, and Eric Bodden. “Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems”. In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–29 (cit. on p. 13).

[73] Nick Stephens et al. “Driller: Augmenting fuzzing through selective symbolic execution.” In: *NDSS*. Vol. 16. 2016. 2016, pp. 1–16 (cit. on p. 14).

[74] Xin Sun et al. “Detecting code reuse in android applications using component-based control flow graph”. In: *IFIP international information security conference*. Springer. 2014, pp. 142–155 (cit. on p. 14).

[75] Robert Tarjan. “Depth-first search and linear graph algorithms”. In: *SIAM journal on computing* 1.2 (1972), pp. 146–160 (cit. on p. 53).

[76] 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. 13, 32).

[77] 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. 35).

[78] Alexey Vishnyakov et al. “Sydr: Cutting Edge Dynamic Symbolic Execution”. In: *2020 Ivannikov Ispras Open 134 Conference (ISPRAS)*. IEEE. 2020, pp. 46–54 (cit. on pp. 15, 23).

[79] *vmpsoft*. `https://vmpsoft.com/` (cit. on p. 27).

[80] 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. 13).

[81] Chenxi Wang et al. “Protection of software-based survivability mechanisms”. In: *2001 International Conference on Dependable Systems and Networks*. IEEE. 2001, pp. 193–202 (cit. on p. 27).

[82] Ruoyu Wang et al. “Ramblr: Making Reassembly Great Again.” In: *NDSS*. 2017 (cit. on p. 26).

[83] Shuai Wang, Pei Wang, and Dinghao Wu. “Reassembleable disassembling”. In: *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 2015, pp. 627–642 (cit. on pp. 26, 27).

[84] Tielei Wang et al. “TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection”. In: *2010 IEEE Symposium on Security and Privacy*. IEEE. 2010, pp. 497–512 (cit. on p. 14).

[85] Liang Xu, Fangqi Sun, and Zhendong Su. “Constructing precise control flow graphs from binaries”. In: *University of California, Davis, Tech. Rep* (2009) (cit. on pp. 14, 23).

[86] Babak Yadegari et al. “A generic approach to automatic deobfuscation of executable code”. In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 674–691 (cit. on p. 27).

[87] Insu Yun et al. “{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing”. In: *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 2018, pp. 745–761 (cit. on p. 14).

[88] Bin Zeng. *Static Analysis on Binary Code*. Tech. rep. Tech-report, 2012 (cit. on pp. 17, 22, 23).

[89] Mingwei Zhang and R Sekar. “Control flow integrity for {COTS} binaries”. In: *22nd {USENIX} Security Symposium ({USENIX} Security 13)*. 2013, pp. 337–352 (cit. on pp. 26, 27).

[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 pp. 14, 23).