UNIVERSITÉ DE VERSAILLES SAINT-QUENTIN-EN-YVELINES

Université de Versailles Saint-Quentin-en-Yvelines **École Doctorale "STV"**

Une approche générique pour la définition de composants bas niveau d'analyse binaire multi-architecture

A generic approach to the definition of low-level components for multi-architecture binary analysis

THÈSE

présentée et soutenue publiquement le 02 Juillet 2014

pour l'obtention du

Doctorat de l'Université de Versailles Saint-Quentin-en-Yvelines

(spécialité informatique)

par

Cédric Valensi

Composition du jury

Directeur de thèse :	William Jalby	- Professeur, Université de Versailles
Président :	Denis Barthou	- Professeur, Université de Bordeaux
Rapporteurs :	François Bodin	- Professeur, Université de Rennes
	Henri-Pierre Charles	- Chercheur, CEA
Examinateur :	Jean-Christophe Beyler	- Ingénieur expert, Intel

LRC IT@CA

Exascale computing research

Remerciements

Voici à présent 6 ans que j'ai quitté le monde de l'entreprise pour rejoindre celui de la recherche. Après 5 années de thèse, me voici enfin arrivé à l'importante étape des remerciements, point d'orgue de la rédaction du manuscrit.

Je tiens à exprimer en tout premier ma reconnaissance à mon directeur de thèse M. William Jalby qui m'a accepté dans son équipe d'abord comme ingénieur de recherche puis en tant que thésard.

Je remercie ensuite mes rapporteurs, MM. Henri-Pierre Charles et François Bodin, pour leurs relectures de mon manuscrit et leurs précieux conseils.

Je remercie également le Président du jury Denis Barthou pour m'avoir fourni les problématiques initiales de ma thèse et m'avoir conseillé tout au long de celle-ci. Je remercie Jean-Christophe Beyler pour avoir accepté de faire partie de mon jury de soutenance, ainsi que pour son soutien lorsque nous travaillions ensemble au sein du laboratoire Exascale.

Je remercie également pour leur encadrement les chefs successifs des équipes auxquelles j'ai appartenu, Jean-Thomas Acquaviva pour ses conseils et ses encouragements, et Andrés Charif-Rubial pour m'avoir poussé à ajouter de nouvelles fonctionnalités sur MADRAS, même au prix de nuits blanches.

Je remercie aussi les membres de l'équipe d'analyse des performances pour leur collaboration, en particulier les 3 ingénieurs de recherche qui ont eu chacun à porter une part du fardeau qu'est la maintenance du code de MADRAS et MINJAG: Mathieu Tribalat, qui a réécrit le code du parseur ELF et affronté les 2000 lignes de la fonction scn_reorder, Jean-Baptiste Le Reste qui a eu à résoudre les problématiques plus complexes que prévu du patch statique et les sournoises GNU_IFUNC, et Hugo Bolloré, qui s'occupe de la mise à jour des grammaires binaires et des macros d'assemblages et de désassemblage, et a du plonger dans les méandres des règles de codage ARM, ou plutôt ses exceptions.

Je remercie tous les autres membres du laboratoire pour avoir supporté mes plaintes de plus en plus fréquentes au fur et à mesure que la thèse avançait ainsi pour les discussions autour de la machine à café et au Restaurant Universitaire (sur l'utilité de la thèse et la qualité de la nourriture dans le cas de Franck).

Une mention particulière à tous mes confrères et consœurs thésard(e)s, aussi bien pour leur accueil lors de mon arrivée pour les plus anciens que pour leur soutien mutuel; celles et ceux qui sont depuis devenus Docteurs, Souad (que je remercie au passage pour son modèle LaTeX de thèse), Stéphane, Julien, Emmanuel, José, Hafid, Yuriyi, Sylvain; et celles et ceux qui le seront bientôt, Sébastien, Jean-Baptiste, Asma, Zakariah, Aurèle, Nicolas, Vincent, Michel : bon courage à vous tous.

Je tiens aussi à remercier M. de Feraudy qui m'a conseillé lorsque j'ai décidé de rejoindre le monde de la recherche et m'a recommandé auprès de William Jalby.

Je remercie tous mes amis pour leur patience lorsqu'ils tentaient eux aussi de comprendre quel était exactement mon sujet de thèse et combien de temps j'allais encore y travailler et quand j'annulais mes rendez-vous au dernier moment pour cause de thèse (mention spéciale au Groupe du Jeu Dit, qui m'a laissé déserter la scène pour m'occuper de mon manuscrit).

Enfin, je tiens à remercier mes parents, pour leur soutien sans faille tout au long de la thèse, et mon frère Flavien, pour m'avoir initialement motivé à rejoindre la recherche par la visite de son laboratoire meublé d'appareils dignes d'un savant fou, puis pour m'avoir fourni les contacts nécessaires pour faire de ce projet une réalité.

À mes parents et mon frère

Résumé :

L'analyse de performance d'applications pour systèmes hétérogènes requiert de pouvoir gérer la diversité des architectures impliquées. L'analyse des applications au niveau binaire offre une précision optimale mais implique l'utilisation de désassembleurs et patcheurs. Ces outils doivent être capables de supporter les fréquentes évolutions et la variété des architectures impliquées dans ces systèmes. La plupart des désassembleurs et assembleurs utilisent une représentation hard-codée des architectures supportées, dont la maintenance est source d'erreurs et consommatrice de temps.

Cette thèse propose une méthode permettant la description des jeux d'instructions des architectures suivant le formalisme des grammaires. Cette approche permet de simplifier la mise en œuvre des encodeurs et décodeurs pour les nouvelles architectures et leurs mises à jours ultérieures. La représentation sous forme de grammaire permet aussi une adaptabilité optimale de la sortie des décodeurs.

Cette théorie est ensuite validée par la mise en œuvre d'un désassembleur basé sur l'analyseur syntaxique généré. Nous montrons que ce désassembleur répond aux besoins des outils d'analyse en permettant le support d'architectures multiples et en présentant les résultats sous un format unifié. Les tests ont montré que ce désassembleur générique est comparable en terme de vitesse et de précision aux outils hard-codés.

Nous présentons ensuite une application des résultats du désassemblage sous forme d'un outil de réécriture binaire. Cet outil utilise un assembleur basé sur l'encodeur généré depuis la représentation de la grammaire et permet d'effectuer des opérations de patch bas niveau sur les fichiers binaires. Les tests ont montré que l'instrumentation de codes basée sur ce patcheur permet d'obtenir un coût en temps réduit par rapport aux outils existants.

Le désassembleur et le patcheur sont fonctionnels et intégrés dans l'outil Madras qui est un élément essentiel de l'environnement Maqao.

Mots clés : Désassemblage, Décomposition analytique, Grammaires non contextuelles, Automates à états finis, Analyse de performances, Réécriture binaire

Abstract:

Performance analysis of applications running on heterogeneous systems requires the ability to handle the diversity of those architectures. Analysing applications at the binary level offers a better accuracy but implies the use of disassemblers and patchers. These tools must be able to follow the frequent evolutions and variety of the architectures involved in those systems. Most disassemblers and assemblers use a hard coded representation of the supported architectures, which is error-prone and time-consuming to maintain.

This thesis proposes a method for describing architectures instruction sets based on a grammar formalism. This approach allows to simplify the implementation of encoder and decoders for new architectures and their subsequent updates. The grammar representation also offers optimal customisation of the decoders outputs.

This theory is then validated with the implementation of a disassembler based on the generated parser. We show that this disassembler meets the needs of analysis tools by supporting multiple architectures and presenting its results under a unified format. Tests show that this generic disassembler is comparable to existing hard coded tools in terms of speed and accuracy.

We then present an application of the results of the disassembler under the form of a binary rewriting tool. This tool uses an assembler based on the encoder generated from the grammar representation and allows to perform low-level patching operations on binary files. Tests have shown that code instrumentation based on this patcher allows to obtain a reduced overhead compared to existing tools.

The disassembler and patcher are functional and integrated into the MADRAS tool which is an essential element of the analysis framework MAQAO.

Keywords: Disassembly, Parsing, Context-free grammars, Finite State Automata, Performance Analysis, Binary rewriting

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

The domain of High Performance Computing (HPC) saw access to higher performance be obtained at the price of an increased complexity in all steps of a program life cycle, from the compilation chain to the processor internal architecture. While performance improvement remains a critical need, this complexity makes this task even more difficult. The field of performance analysis aims at finding the best way to understand the behaviour of HPC applications in order to pinpoint their bottlenecks, then identify their causes and offer solutions for their removal.

A possible way of achieving this is through the analysis of the source code of applications. While this is one of the easiest methods, it is not the most precise nor reliable. In order to take advantage of the architecture abilities, compilers can perform extensive code transformations, such as loop unrolling or constant propagation, when generating the binary executable from the source code. These transformations may significantly alter the control flow of a program from its source representation and render most deductions made from the analysis of the source code alone incomplete at best. It may also be interesting to be able to identify which of those transformations were performed by the compiler, in order to detect more efficient alternatives that it missed because of an ambiguous implementation of the algorithm in the source code. Finally, the source code of an application may not be fully available for analysis, for instance because of confidentiality restrictions.

It is also not always possible to retrieve all useful information from a static analysis of the code. Some information, such as the timings of routines, can not be deduced from such an analysis without a significant error margin, if at all. It may therefore be useful to be able to modify the program, for instance by inserting probes, in order to retrieve additional information at run time. Performing such modifications in the source code can however impact the compilation process and thus the resulting executable, as the compiler will take the altered code into account when performing its optimisations. This may lead to performance very different from the original, thus voiding the information gained from those modifications.

It is therefore interesting to be able to directly analyse the assembly code generated by the compiler, since this is what will be actually be executed by the processor. Most compilers allow to generate assembly files instead of binaries from a source file. However, it is usually not possible to generate such a file for the whole program if it was compiled from multiple source files, which is the case for most real applications. This will not be possible either when the sources are not available. Additionally, while it is much closer than the source code to what will be actually executed by the processor, an assembly file may lack some information useful for analysis. For instance, the assembler may add padding instructions for alignment when converting the assembly code to binary, which could have a non negligible impact on the overall execution time. Also, some information found only by analysing the binary code, such as the length of binary instructions, may be needed by analysis, for instance to compare with the size of the instruction cache. Finally, if the code must

be instrumented, assembly will still be necessary after modification of its assembler code.

A solution for these constraints is to directly process the binary file, for analysis and for instrumentation, as illustrated in figure 1.1. This implies being able to decode the binary code in the file back into assembler code through disassembly, and to directly modify the binary file through patching.

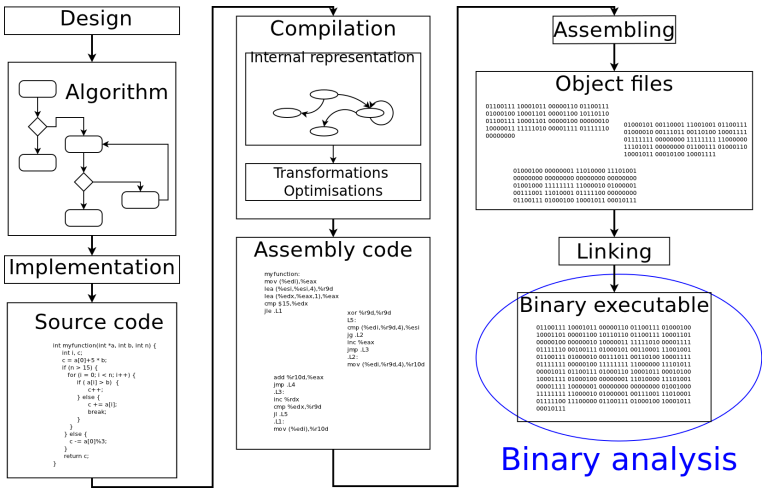

Figure 1.1: Location of binary analysis in the compilation chain of an executable

Disassembling presents multiple challenges, since a binary file, being intended solely to be executed by the processor, may lack some information necessary to analyse its contents, or require some additional processing to retrieve. The internal structure of binary executables can also make any modification complex to perform while keeping them functional, as executables, unlike linkable files, are not intended to be modified. Patching also requires being able to assemble instructions.

While not uncommon tools, most disassemblers return either a printed version of the assembly code or a simple abstraction, neither of which is sufficient for the purposes of performance analysis or instrumentation at the assembly level, which requires a detailed abstraction of the assembly instructions to allow fine-grained insertion of probes as well as rebuilding the instrumented binary. Existing disassemblers may not be available for every architecture involved in a system, require multiple compiled versions of the program to handle all of them, and need some customisation to adapt their output to the needs of the analysis to perform.

The solution is the implementation of a generic customisable disassembler and patcher. To handle HPC systems involving more than one architecture, it is necessary for such a tool to be able to support multiple architectures while presenting its results in a unified format to reduce the amount of adaptation required from the applications using it.

Supporting new architectures or keeping up to date with their evolutions can be costly in terms of development time, and negatively impact the implementation of actual analysis features. It is therefore necessary to be able to automatise the generation of architecture specific codes, while keeping an architecture independent interface with higher level tools. This implies being able to define abstractions of the binary encoding rules and of the assembly language that can be used to represent any architectures.

1.1 Contribution

In this thesis, we present a method for describing Instruction Set Architecture (ISA) encoding rules under the form of a context-free grammar with an architecture independent format. Implementing a new architecture or expanding an already recognised one is done by creating or editing the relevant grammar file, thus allowing to stay as close as possible to the format of the instruction list available from the architecture documentation. This is then used to generate a parser able to process binary instructions for the corresponding architecture.

We will then present an application of this parser as a multi-architecture disassembler functional for every architecture for which a description has been supplied. The output of the disassembler is an architecture independent representation of instructions, allowing analysis tools using it to remain agnostic with regard to architecture when performing standard computations such as control flow reconstruction.

We will finally describe an application for the representation of instructions returned by the disassembler under the form of a binary patcher. This contribution will include solutions to the challenges presented by the patching of binary executables such as the preservation of control flow.

1.2 Organisation

The outline of this dissertation is presented below.

Chapter 2 presents the context of our research, focusing on the requirements of analysis tools, the constraints of assembly languages, and briefly describing the overall structure of binary files.

Chapter 3 presents the challenges tied to the parsing of binary code, the related works on this subject, and our solutions for describing binary encoding formats as context-free grammars and generating the associated parser.

Chapter 4 focuses on disassembly, presenting its challenges, common solutions and related works, then our solution based on the multi-architecture parser generated from the previous chapter.

Chapter 5 focuses on patching, describing the challenges and common solutions for instrumenting files as well as related tools, then our choices for addressing these challenges using the output of the disassembler presented in the previous chapter.

Chapter 6 presents extensions of the tools implementing the principles of the previous chapters, aiming at increasing their performance and coverage.

Chapter 7 concludes this dissertation.

Annexes contain a detailed description of the tools implementing the concepts presented in this thesis.

Annex A presents Minjag, the tool allowing to process a grammar representing the ISA and generate the code for the corresponding parser according to the principles described in Chapter 3.

Annex B presents the API of Madras, **M**ulti **A**rchitecture **D**isassembler **R**ewriter and **As**sembler, which allows to disassemble and patch binary files according to the principles described in Chapters 4 and 5.

Chapter 2

Elements of binary analysis

Analysing binary files presents specific challenges, especially impacting the low level elements of the analysis chain. We will outline here the specificities of this field, focusing on the common operations performed by analysis tools and the associated requirements. We will also consider the structure of binary executable files, which share common characteristics across architectures and platforms that need to be taken into account when analysing and modifying them.

2.1 Analysis tools

Analysis tools aim at providing high-level static and dynamic information about the characteristics of an executable file in a unified format. As the following examples show, their operating mode and features vary, however their base operations rely on common elements. HPC Toolkit [34] performs sampling to retrieve the performance profile of an application, then correlates its results with information about the structure of code deduced from binary analysis. TAU [94] is a framework integrating multiple instrumentation tools using various techniques such as source code instrumentation, preprocessor or compiler patching, wrapper use and binary rewriting, and presents the results using different displays. HPCView [78] performs correlation of source code with data gathered from instrumenting or tracing and presents the results in a unified format.

2.1.1 Common operations

The operations performed by analysis tools can be broken down into the following categories:

- Rebuilding the structure of the application: Control Flow Graph (CFG), Call Graph (CG), Data Dependency Graph (DDG)
- Estimation of the performance from static analysis
- Instrumentation in order to time code sections, profile runtime values, or survey the code execution
- Presentation of the results in a unified format

Most tools either rely on a third-party tool for these tasks, such as PIN [67] or DynInst [41], or integrate hard-coded modules specific to their needs.

2.1.2 Additional constraints

HPC applications can involve multiple languages, compilers and architectures, sometimes in the same application. Since analysis tools aim at offering advices on how to improve existing applications, they should not impose constraints on the subject of

their analysis to be effective. The tools therefore need to support most versions of the elements involved in a compilation chain in order to ensure an optimal coverage.

Operating at the language level implies handling the most commonly used languages, such as C, C++ and Fortran. This is also important when presenting analysis results in order to correlate them with the source code.

Operating at the compiler level allows access to the intermediary representation it uses for generating the assembly code, which is a very powerful tool for retrieving the structure of a program as well as for modifying it for instrumentation. However, this implies some access to the code of the compiler or its ability to support plugins. While this is easy for Open-Source compilers such as GNU **gcc**, it can be more complicated for industrial compilers as Intel **icc**. It is also ineffective if the code was not compiled but directly written in assembly code.

Finally, operating at the binary level implies being able to support the multiple architectures possibly involved in a single application. Binary code also contains less information than other stages about the code behaviour, which makes the extraction of useful analysis data more complex.

2.1.3 Operating at the binary level

We will describe here the requirements induced by the operations described in 2.1.1 when operating at the binary level.

Building the CFG This task implies the ability to retrieve the list of assembly instructions present in a binary file, including all information relative to the flow. Since tools are expected to work on multiple architectures with different instruction sets, this information must be accessible in a format independent on the ISA. It is therefore necessary to be able to correctly disassemble files and return an architecture independent abstraction of the assembly code they contain.

Static analysis This task requires additional knowledge concerning the behaviour of assembly instructions. This knowledge can be as basic as the operation performed by the instruction and the type and size of data affected, but can also include information not easily available from the constructor documentation, such as the number of cycles needed by the processor to execute the instruction or its latency.

Instrumentation This task requires being able to edit a binary file to alter its behaviour at a low granularity while ensuring that the overall execution of the program is not affected, unless that is the desired effect.

Presenting the results In order to offer meaningful information, this requires being able to establish a correlation between the binary code present in the executable with its source code, since the latter is usually what the developers are able to update to improve performance according to the results of the analysis.

2.2 Assembly language

We will now describe the specificities of assembly language and its associated binary representation, focusing on the challenges they present to analysis and parsing.

Assembly is a low-level language directly representing the operations performed by the processor. Unlike compiled languages, assembly is specific to a given processor architecture, as it directly depends on the processor instructions and registers sets.

An assembly statement is an instruction. Instructions are composed of a mnemonic, which is a command to be executed by the processor, and a list of operands. Operands can be registers, memory addresses or immediate values. Mnemonics can be roughly categorised between the following operations:

- A numerical operation between the operands
- The affectation of a value, immediate or stored in a register or memory address, to another register or memory address
- Branching of the control flow of the program to a given address

Assembly language use labels to reference the destination of branch instructions. Special directives also allow to declare some labels as functions; this is especially useful for declaring functions intended to be invoked from a different assembly source file, or in the code of a shared library.

2.2.1 Binary encoding

The encoding rules for an architecture specify how to translate assembly instructions into binary code, also called assembling. Those rules usually establish a one-to-one correspondence between both representations, although in some cases different instructions can represent an identical operation, and as such be indifferently encoded as one or another. For instance, an instruction whose mnemonic represents the swapping of values between both of its operands can accept any order between its operands, and thus different encodings, for the same operation. Some mnemonics can also be homonyms of each others, leading to different assembly instructions having the same coding. Finally, some architectures may define macro mnemonics, representing multiple instructions, which are assembled into their respective binary encodings. Conversely, alias mnemonics can be defined, representing a specific variant of an instruction, and assembled as such.

In binary code, branch instructions do not use labels to reference their destination, but instead use the address of the destination instruction. The only use of labels in binary code is for relocations tables (cf. 2.3.2.3).

There are no separations between instructions in binary code. Depending on the architecture, encoded binary instructions can be of different lengths in bits.

2.2.2 Execution

Instructions in a program are executed by the processor in the order into which they appear in the file, until a branch instruction is encountered. The *instruction pointer*, or *program counter*, represents the virtual address of the instruction to be executed by the processor. It is used implicitly by branch instructions to address their destination. It is usually handled as a register, which allows to use it in the expression of memory addresses and access or edit its value with specific instructions.

Most assembly languages include a *no operation instruction*, usually called `nop`, that does not perform any operation. Such an instruction can be used as filler to align code or to temporarily stall the processor and ensure data dependencies

are broken. Compilers may however achieve these goals through other instructions whose performed operation do not actually affect the control flow nor the data set, such as switching the content of a register with itself.

2.2.3 Addressing

A branch instruction can reference its destination through either *absolute addressing*, using the address of the destination, or *relative addressing*, using the offset between the branch and its destination. *Direct branches* store this reference as an immediate operand while *indirect branches* store it in a register or memory operand.

References to addresses in memory are based either on an absolute value, the value stored in one or more registers, or the sum of both. Some architectures may also allow to reference addresses through the offset from the current instruction; such a mode is mainly used for referencing memory locations defined in the executable, as the address of dynamically allocated variables would be unknown before runtime and may need a relocation. As noted in 2.2.2, these addresses may be represented as being based on the instruction pointer.

Direct addressing is easier to handle for analysis tools as the referenced addressed can be deduced from the address of the referring instruction.

2.2.4 Overview of different architectures

Below is a quick presentation of some of the most used architectures, focusing more closely on those derived from the Intel x86 architecture as it offers particular challenges in terms of disassembly and patching and is widely represented in HPC.

2.2.4.1 IA-32

The IA-32 architecture [19, 17, 18], also called x86, i386 or x86-32, is a CISC architecture used by the Intel processors since the 8086 version and by AMD processors [1, 2, 3, 4, 5]. The architecture counts approximatively 400 different mnemonics.

IA-32 assembly instructions accept zero to two operands. Operands representing memory addresses can be expressed as the sum of an immediate value with the value stored in a base register and the value of an index register multiplied by a scalar.

The binary encoding of instructions vary between 1 and 15 bytes. Among the factors causing an instruction length to vary are the 1-byte *legacy prefixes*, of which an instruction can accept representatives of up to four different families in any order, with a given prefix possibly appearing more than once. Another factor is the length over which immediate values and memory displacements are encoded, which vary between 1 and 4 bytes depending on their ranges. Finally, the type of operands and the expression of memory addresses can impact the instruction length as well.

One particular specificity is that most direct branch instructions exist in two variants of different length, one coded on 5 bytes with a 32-bits signed offset range and the other coded on 2 bytes with an 8-bits offset range.

Figure 2.1 presents an overview of the structure of a binary IA-32 instruction.

2.2.4.2 Intel 64

The Intel 64 architecture, also known as AMD64 or x64, is an evolution of IA-32 to support 64 bits processors. It ensures backward compatibility with all previous versions of the processors and as such defines more than 1000 different mnemonics.

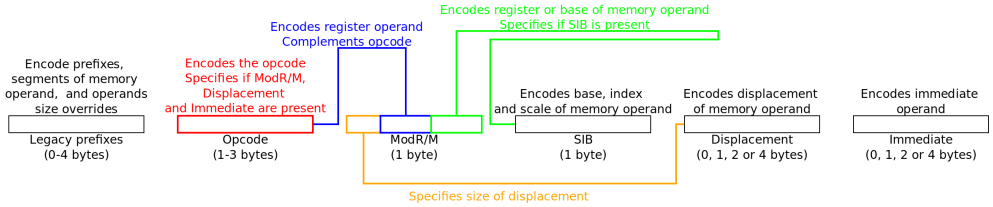

Figure 2.1: Overview of the structure of an encoded IA-32 instruction, describing the relationships between the various bytes involved in the encoding.

The Intel 64 architecture allows access to 16 64-bits general purpose registers instead of 8 for IA-32. It is possible to access only the 32, 16 or 8 least significant bits of those registers, thus emulating registers with this size, and ensuring backward compatibility for codes using such registers.

Although most IA-32 instructions exist in Intel 64, the encoding of some of them may be invalid or have a different meaning. For instance, one possible IA-32 coding for the `inc` and `dec` mnemonics (allowing to respectively increment or decrement a register) is used in Intel 64 for a prefix allowing to access additional registers. Processors supporting Intel 64 can be switched to a "legacy mode", effectively operating as 32 bits processors, through a sequence of instructions accessible only by the operating system in protected mode.

The Intel 64 architecture undergoes frequent evolutions, with new instructions appearing every 6 months and whole new instruction subsets, such as SSE and AVX, every two years. Instruction subsets can contain hundredths of new instructions and new encoding rules for representing them as well.

The SSE extensions introduced 16 128-bits vector registers, and instructions accepting up to three operands. The AVX extension extended those registers to 256 bits, with instructions accepting up to four operands. The latest extension AVX2 allows to access non-contiguous memory addresses to perform gather or scatter operations on vectors. The upcoming AVX-512 extension will increase the number of vector registers to 32 and extend them to 512-bits, and add specific mask registers.

Intel 64 instructions use an additional optional 1-byte prefix (REX) to encode that an instruction uses 64 bits operands, and to access the additional registers (identifier ≥ 8). The SSE extension uses some of the legacy prefixes to distinguish between opcodes, making them mandatory for some encodings. The AVX extension uses another 2 or 3 bytes prefix (VEX), to extend the functionalities of the REX prefix and access the new 256 bits registers. The AVX-512 extension will use a new 4-bytes prefix, EVEX, to encode its new features.

All Intel 64 direct branch instructions use relative addressing. The architecture also allows to address memory locations relatively to the current instruction pointer.

Figures 2.2 and 2.3 describe the uses of the REX and VEX prefixes in the binary encoding of Intel 64 instructions.

2.2.4.3 Intel Xeon Phi coprocessor

Intel Xeon Phi coprocessor [22, 21], also known as MIC or K1OM, is the latest Intel multiprocessor architecture. It is based on Intel 64 processors without the SSE nor AVX extensions, and as such no access to the 128 or 256-bits vector registers, but offers access to 32 512-bits vector registers. Xeon Phi defines 500 different mnemonics.

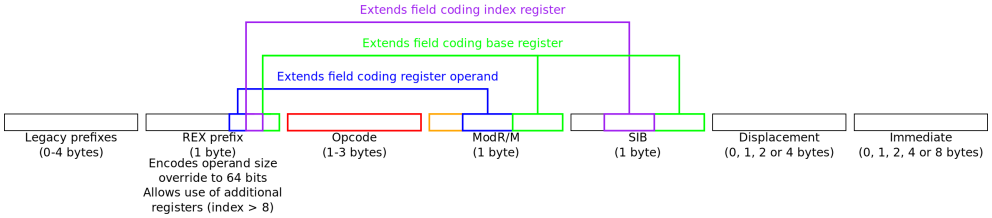

Figure 2.2: Overview of the structure of an encoded Intel 64 instruction, focussing on the use of the REX prefix.

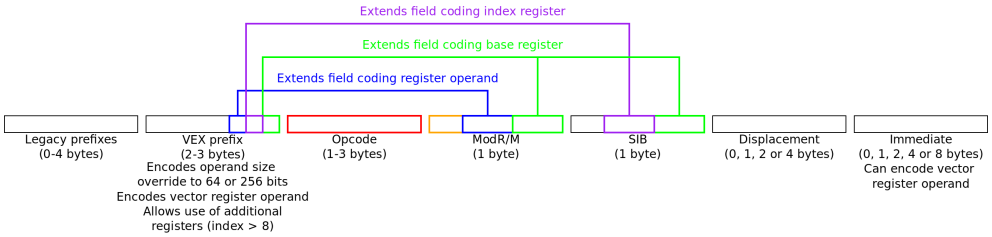

Figure 2.3: Overview of the structure of an encoded AVX Intel 64 instruction, focussing on the use of the VEX prefix.

The memory or vector register operands of a Xeon Phi coprocessor assembly instruction accept a modifier to tweak their use. Such a modifier can either be a flag specifying conversions to apply to the operand, a cache line eviction hint, or a mask, defined in a special register, specifying which elements in a vector will be affected by the operation.

Some conditional branch instructions in Xeon Phi can accept two operands, representing the destination and the condition to apply to the branch.

The binary encoding of Xeon Phi instructions obeys the same rules as Intel 64. A 4-bytes prefix (MVEX) is used to encode the new features of the architecture. Another difference with Intel 64 is the possibility of compressing offsets used in memory addresses by dividing the encoded value by a given factor.

2.2.4.4 Intel Itanium

The Intel Itanium architecture [20], also called IA-64, is a parallel Intel architecture distinct from x86, although Itanium processors allow to run IA-32 applications. The Intel Itanium instruction set numbers approximately 200 mnemonics. The architecture allows parallelism at the instruction level, and offers specialised mechanisms for handling conditional execution of instructions. The use of Intel Itanium processors in HPC applications saw a decrease in the recent years.

Intel Itanium instructions accept at least three operands, and use a predicate for identifying if the result of the instructions shall be kept or discarded. Instructions are grouped into bundles, composed of three instructions and a template specifying the type of instructions it contains.

In binary, all instructions are encoded on a fixed length of 41 bits, and the templates on five bits, thus making the encoding of a bundle 128 bits long.

2.2.4.5 ARM

The ARM architecture [90, 12, 9, 10, 11, 7] is a RISC architecture designed and licensed by ARM Holdings and used by various processors. It is mainly used in mobile systems but also involved in ongoing HPC projects such as the Mont-Blanc project [25]. The architectures undergoes frequent evolutions, with new versions released on a yearly basis. ARM processors support three instruction sets: ARM, which defines 32-bits instructions, Thumb, which defines 16 or 32-bits instructions, and A64 (from ARMv8 onwards), which defines 64-bits instructions. ARM instruction sets number approximately 200 different mnemonics.

Instructions from the ARM and A64 instruction set are always encoded on a fixed length of 32 bits and aligned on a four bytes boundary. Most ARM instructions are prefixed with a condition specifying whether the instruction must actually be executed. When present, this condition is always encoded on the five first bits. The binary template of the instructions depends on the type of operations performed.

Instructions from the Thumb instruction set are encoded either on 32 or 16 bits and aligned on a two bytes boundary. The first five bits of an instruction allow to identify its size. Branch instructions exist in the 16 and 32 bits version.

In binary code, both instruction sets overlap: encoded instructions from one can be decoded as a different instruction from the other. A same ARM executable file can contain blocks of instruction from the ARM and Thumb instruction sets. In this case, a specific branch instruction is used to notify the processor of the change of architecture. This mechanism, also called *interworking*, can also be used to switch the processor to the ThumbEE set, which uses a slightly different instruction set from Thumb, or to the Jazelle state to execute bytecode programs. A64 uses different encoding rules than either the ARM or Thumb instruction sets. A64 processors can also execute 32-bits code but the switch is done at the exception level.

ARM contains instructions for setting the value of the instruction pointer register, allowing to branch the control flow without using an explicit branch instruction.

2.2.4.6 Power ISA

Power ISA [29], evolved from the PowerPC, is a RISC architecture used by IBM and Motorola processors in home computers and in HPC. It offers 32 and 64 bits modes. The instruction set is stable and number approximately 900 mnemonics.

All Power ISA instructions are 32 bits long and must be word aligned. A field in an instruction coding may be split over multiple locations inside it. The template of an instruction binary encoding depends on the type of operations it performs.

2.3 Binary executable

An executable is a binary file with a specific format allowing the operating system to identify how to load it into memory and execute it. The same format is usually used for *relocatable files*, which are used as intermediary files generated by a compiler and used by the linker to generate an executable, and library files, which contain common shared code usable by executables. This format can also be used for representing dump files, generated after a program crash for debugging.

Dynamic executables rely on external libraries for the definition of some functions or variables, while *static executables* are stand-alone containing all information

needed for running them. A relocatable file contains compiled code and information used by the linker to generate a functional executable from it and other related files.

2.3.1 General structure

Executable files commonly contain the following elements:

- Format identifier ("magic word")
- Header describing the structure of the file
- Binary code to execute
- Variables used by the code
- Directives for the OS on how to load and execute the file
- Relocation tables
- Information on how to invoke external functions for dynamic executables
- Optionally:
  - Debug information, added if requested by the compiler
  - Labels, depending on compiler settings.

In most binary formats, files are broken down into *sections*. A section contains one given type of data, such as code, variables or relocation, and other specific information, such as the virtual addresses at which it must be loaded when executing the file and the rights to set on the corresponding memory segment. It is possible for a section not to be loaded into memory when running the program if its content is not needed at runtime, for instance in the case of debug information. Sections are identified in a header specifying the offsets in the file containing their data.

Some formats also use the concept of *segments*. In these cases, segments are used by the loader, while sections are used by the linker when building the file.

2.3.2 Contents

Apart from executable code, binary files commonly contain information which can be useful for further static analysis.

2.3.2.1 Symbols

Binary files can contain string *symbols*, which are associated to an address and possibly a type characterising what they represent. They can be used to reference specific locations in the file such as the beginning of a function or a variable. Most of those symbols are however not needed by an executable file, and as such may be removed without affecting its behaviour. The most important exception to this are relocations (see 2.3.2.3), which explicitly need a symbol. When generating binary files, compilers usually keep only a small subset of the labels appearing in assembly code, mainly restricted to function and variable names.

It is also important to note that, while a binary format may define specific types for identifying the uses of a symbol, those are not mandatory either. It is therefore

possible for a binary file to contain symbols actually corresponding to the beginning of functions, but being identified with a generic non indicative type.

The compilers for certain languages, like C++ or Fortran, use the concept of name mangling, where the symbols stored in a binary file are not the same as those used for a variable or function name as they appear in the source code, but are instead run through an encoding (mangling) algorithm. In this case, it is necessary to decode the names found in the file to retrieve the original names. The algorithm used to mangle the names depends on the language and the compiler used to generate the file. [58] presents some of those algorithms, which are not always available and may require some reverse engineering to retrieve.

2.3.2.2 Variables

An executable can contain variables, usually those that may be accessed from anywhere in the program (global variables) or have a fixed value, such as a character string or a branch table used by a switch operation; dynamically-allocated variables are not present in an executable. Variables that can be read or written are usually in a different section or segment than those that are read only, allowing to set the appropriate rights on it at run time.

Some systems may adopt a special behaviour with regard to variables whose value is not initialised at compilation time. Those variables are defined in a special section that does not take any space in the file, but whose attributes specify a size in memory. The loader is responsible for allocating the necessary space for those sections when loading the file in memory.

2.3.2.3 Relocation

Relocation is the process of updating a reference address in a file after it was generated. It is usually performed through a relocation table which references locations in the code or data and the symbol to which they are linked. When the address of the symbol is known, the reference is updated accordingly. It is mostly used by the linker during compilation; a compiled relocatable file contains relocation tables for all symbols defined in other files, allowing the linker to resolve references to the code or data corresponding to those symbols in the final linked file.

2.3.2.4 Referencing an external source

Most recent file formats allow to reference code or variables defined in an external file. The most common use for this is the invocation of functions defined in a shared library. The address at which the external reference is loaded in memory is not known when the executable is generated, so a relocation table (as described in 2.3.2.3) must be used. References to an external source are usually redirected to a specific section linked to a relocation table. Depending on the system and settings of the executable, the addresses can be filled either during program loading or when they are accessed for the first time.

2.3.2.5 Debug information

Debug information is not needed for a standard execution of a program, but is used when running it through a debugger utility. It is usually stored in a separate section not loaded into memory at runtime. The format of those sections may be

independent from the format representing the executable. Debug information aims at allowing the debugger to establish the relation between the source code and the binary file, and as such can contain correspondences between binary addresses and lines in the source code, or between memory locations and variable names.

2.3.3 Common binary formats

We present here a quick overview of the most important binary formats. We will focus more closely on ELF as it is the format with which we worked more extensively.

2.3.3.1 The ELF format

ELF (Executable and Linkable Format [14, 76]) is the format used by the Unix and Linux operating systems. It is used for object files, executable files, library files, and core dump files. ELF is an evolution of the `a.out` format.

The contents of an ELF file can be represented as a table of either sections or segments, each being identified in a distinct header. The section header is mandatory in an object file while the program header is mandatory in an executable file or a library. While the section header is optional in an executable file according to the standard, compilers routinely keep it in executable files. A segment usually encompasses multiple sections, but the standard does not specifies that the boundaries of a segment must match with those of sections. An ELF file always contains a general header identifying the file type and the offsets in the file of the sections and/or segments headers. Figure 2.4 represents the structure of an ELF file.

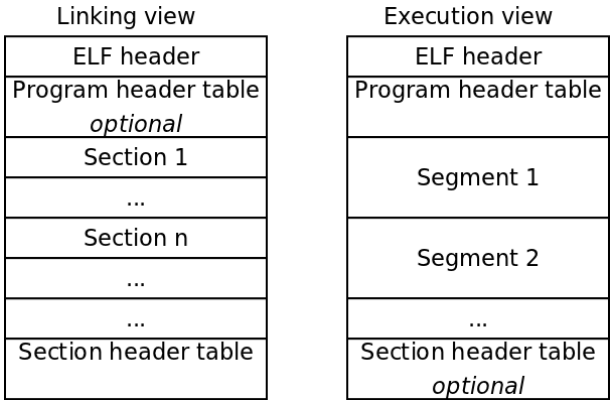

Figure 2.4: Representation of the structure of an ELF file, distinguishing between relocatable and executable files. Source: [14].

Segments Segments contain the virtual address at which the corresponding bytes from the file must be loaded into memory when executing it, as well as the rights to set on those addresses. An ELF executable usually contains the following segments:

- A segment containing the program header,
- A segment describing the interpreter to use when executing the file,
- A readable and executable segment containing the code and read-only variables
- A readable and writeable segment containing read-write variables.

Other segments contain information relative to external libraries for dynamic files.

Sections Sections in an ELF file follow the general description (cf. 2.3.1). Unlike segments, sections can be named. The standard does not impose a strict constraint on sections names, and more than one section can have the same name. However, compilers tend to reserve names for sections containing specific data. Common sections include:

- Executable code sections: "`.text`", "`.init`" and "`.fini`" contain respectively the main, initialisation and termination codes
- Data sections: "`.rodata`", "`.data`" and "`.bss`" contain respectively read-only variables, read-write variables and uninitialised variables
- Relocation sections are named as the section they reference prefixed with "`.rela`"
- String table: "`.strtab`", "`.shstrtab`" "`.dynstr`" respectively contain strings used for labels, section names and dynamic symbols.
- Offset tables: "`.got`" and "`.got.plt`" contain addresses used for referencing dynamic functions or variables.

Accessing dynamic functions A special mechanism allows ELF files to perform on-demand relocation, also called *lazy binding*, by retrieving the address of a dynamic function only when it is first needed, allowing to reduce the invocations of the dynamic loader. Calls to external functions point to a stub, which retrieves the address at which the function is loaded from the Global Offset Table (GOT) and redirects the flow there. The GOT is defined in the binary file, each cell initialised with the address of instructions in the stub allowing to invoke the dynamic loader for the corresponding function based on its name. When the dynamic loader is invoked for a function, it also updates the associated cell in the GOT with the function address, thus ensuring further calls to this function directly branch to it. Executable code for the stubs is usually present in the "`.plt`" section, with the "`.got.plt`" section containing the GOT. Figure 2.5 presents a description of this process.

Debug information ELF file can contain debug information in specific sections that not loaded when executing the program. These sections use DWARF [31], an independent format not tied to ELF, and the debug information they contain depends on the compiler version and options used. Common information stored includes the compiler name and version, the association between line numbers and instructions addresses, the original function names (without the mangling if present), and function parameters, start addresses, and local variables.

2.3.3.2 Other formats

The following binary formats are used on other operating systems.

a.out This format [6], used on older Unix systems, is one of the first formats for binary files. It has been now mostly replaced by the COFF then ELF formats in the Unix/Linux environment. `a.out` files are broken down into sections, containing the executable code, symbols, relocations and data. A header contains the sizes of the various sections, and begins with a branch instruction to the entry point of the

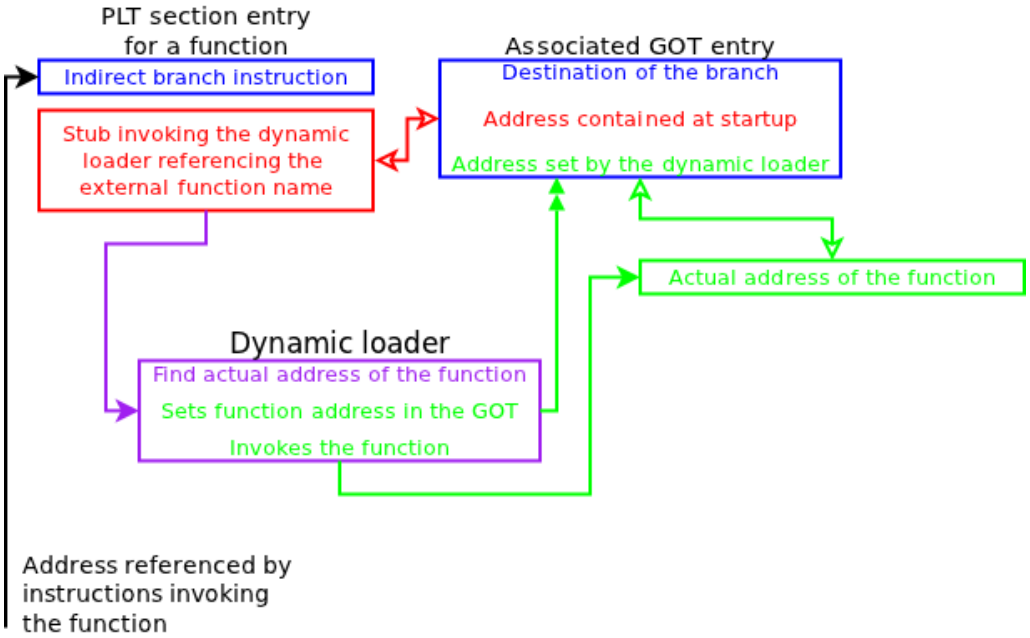

Figure 2.5: Schematic Description of the handling of references to dynamic functions using lazy binding. The indirect branch instruction in the PLT entry is always executed when the dynamic function is invoked from the code, and branches to the address stored in the GOT entry. The following stub is only executed once, and allows the dynamic loader to link this call site to a function name which it will use to find the corresponding address. The string and relocation tables used for performing this link are not represented here. The address of the dynamic function depends on the address at which the corresponding library was loaded into memory.

executable code. The loader ensures that the program does not run past its size by setting a break at the end of the text and data sections. Debug information can also be stored in special entries of the symbol sections.

COFF The Common Object File Format was introduced on more recent Unix systems, before being replaced by ELF. It is used by Windows in conjunction with PE and some of its variants are used by IBM AIX (XCOFF [32]) or Texas Instrument [13]. COFF files are broken down into sections. The format allows for an optional second file header, which is used when the file is executable to specify the program entry point and the sizes to load in memory. A COFF file can also contain a special section storing line numbers for debugging.

Windows PE The Portable Executable format [84] [24] is used by Windows-based operating systems. It is an evolution of the COFF format and is occasionally referred to as COFF/PE. A PE file contains a MS-DOS header, used to display an error message if the file is run on a system unable to support it. This header also contains the offset in the file of the PE header. PE files are broken down into sections, identified in a section table referenced in the PE header. Debug information is stored in the read only data section. Dynamic functions are invoked in a mechanism similar to the lazy binding used in ELF files described in 2.3.3.1.

Mach-O The Mach Object format [28] is used by systems based on the Mach kernel ([33]), the most prominent being Apple Mac OS X. Like ELF, Mach-O is an evolution of the `a.out` format, and both share similar concepts. Mach-O files are broken down into sections and segments, a segment containing zero or more sections. The header also contains load commands allowing to specify additional information about the file, such as the shared libraries sed in a dynamic executable. Multiple Mach-O files can be combined into a multiple architecture binary, containing files compiled for different architectures.

AIF The ARM Image Format [8] is a simple format used for executables, while ALF (ARM Object Library Format) is the corresponding format used for libraries. AIF files use a header specifying the entry point of the executable, a code image and a data image. An AIF executable can relocate itself after being loaded, using a special self-relocation code and a relocation list stored in the code image.

Chapter 3

Generation of a generic binary decoder and encoder

Recognition of the binary encoding format of the architecture for which the binary file has been compiled constitutes the first step in the implementation of a disassembler and assembler. As mentioned in Section 2.2.4, for some architectures this format can be subject to significant and relatively frequent changes, which must be reflected in the implementation of the associated encoder and decoder. It is therefore essential to ensure that the update of an existing architecture is simplified enough to allow frequent modifications at a minimal cost.

Another constraint is tied to the increasing use of heterogeneous systems in HPC applications, where multiple architectures can be involved, for instance in the case of processors coupled with accelerators. In that case, a complete analysis of the whole system requires being able to recognise the instruction sets of all the components it involves. To simplify the analysis chain, it is therefore useful to be able to handle multiple architectures with the same tool, which requires for the implementation of new architectures not to be overly costly.

In order to minimise the architecture specific parts of the analysis chain, it is also interesting to keep a unified higher level representation of the instructions. Analysis tools may need specific additional information, requiring for a disassembler to be modular enough to allow easy customisation of its output.

Since we are aiming at providing generic parsers with multiple possible uses, it is best to separate the description of the ISA from the actions performed upon the successful parsing of an instruction. This allows for instance to use the same representation of an architecture in multiple tools serving different purposes, such as plain disassembly or simulation, and to rely on the same representation to create the associated encoder. Some analyses may also require additional information not deducible from binary code, such as characteristics specific to static analysis or profiling; to avoid looking them up during processing, it can be interesting to load these during disassembly into the representations of instructions for better performance. It can also prove useful to be able to trigger specific actions after the successful disassembly of an instruction, for instance when performing a simulation of the executable. For these reasons, it is essential to easily dissociate the process of parsing the binary code from the actions performed upon a successful parsing.

In this chapter, we will first present common solutions tied to parsing binary code in Section 3.1 and its main challenges in Section 3.2, then our solution as a context-free grammar in Section 3.3 and its implementation in Section 3.4.

3.1 Parsers and architecture representation

We will briefly cover here the main principles of parsing and grammars, and present other works concerning the representation of binary architectures.

3.1.1 Parsers

Parsers are generally used to process files with a strict formatting, for instance source files written in a programming language, and rely on a grammar for specifying the format of the files to parse. They are mainly used for parsing text files, but can are also be used to describe strict binary formats, as presented in [98].

3.1.1.1 Context-free grammars

A grammar is composed of a set of *symbols* and *productions*, a production being a possible form of a given symbol expressed as a sequence of symbols. Symbols are distinguished between those without productions, called *terminals*, and the others, called *nonterminals*. The symbol whose production represents the entire grammar is called the *start symbol*. We will also use the term *token* to identify a terminal symbol representing any possible value over a given length.

For instance, in a grammar representing a programming language, the terminals are the keywords of the language; an example of nonterminal would be the symbol representing a function definition, and the start symbol the symbol representing a whole source file. Tokens would be the names of variables, as they are terminals without a fixed value.

A parser for a given grammar attempts to recognise, or *reduce*, its symbols in the file it is processing. Bottom-up parsers, also called LR-parsers, attempt to match each symbol starting from the lowest ones in the production hierarchy, then work they way up to the start symbol. A *LR(0) parser* only needs the symbol it is parsing to reduce it. Lookahead parsers, also called *LR(x) parsers*, need to check further symbols in a production to successfully reduce a symbol. A parser is usually implemented as a Finite State Automaton (FSA).

A grammar allows to associate a *semantic action* to a given production, which will then be executed upon reduction of the corresponding symbol from this production. This is used for instance to build an internal structure corresponding to the recognition of a pattern.

3.1.1.2 Parsing Expression Grammars

Parsing Expression Grammar (PEG) [60] are composed of a set of symbols and rules. A rule is close to a production, but can be defined using regular expressions. PEG also allow to specify preference between rules. Packrat parsing [59] is an optimisation allowing to implement parsers using memoization and backtracking to speed up parsing.

If used to disassemble a binary instruction, a PEG algorithm would attempt to match the whole instruction recursively. This would lead to inefficient parsing, as the topmost function for disassembling an instruction would attempt to recursively match all the symbols in a rule representing a possible expression of an instruction before switching to the next. Memoization may not even be useful here as the symbols appearing in the various rules could differ. Also, in the case of architectures where instructions have variable length, it would be necessary to try such a recursive parsing for each possible length of an instruction.

3.1.1.3 Procedural parsing

Procedural parsing [35] [104] consists in interleaving the parsing process with the evaluation of the results, thus allowing to control the parsing from the grammar itself. It is useful for handling complex rules that are not easily expressed using a declarative description. It can also be used for improving performance by restricting the possible upcoming choices based on predictions depending on what has been parsed so far.

In our context, this method would allow more flexibility in the handling of complicated encoding rules, such as those present in the ARM architecture. It would also simplify the handling of parsing errors by allowing to specify more precisely the backtracking rules to apply when a parser state can not find a successor matching with the current input. Finally, it could also reduce the time needed for parsing instructions by allowing to optimise some matching operations depending on the overall parsing progress.

This would however imply a more complex grammar, which would then increase the time needed for implementing new architectures or updating existing ones, thus going contrary to our goal of reducing the workload for those implementations. It could also lead to markedly different parsing processes, requiring more implementation to obtain a unified output. Another drawback is that it would make the generation of an encoder and decoder from a single grammar more complex to implement. We consider that declarative grammars are able to represent the encoding rules for all binary architectures and therefore offer a good compromise between coverage and implementation time.

However, since procedural parsing may prove useful for handling specific cases, the implementation of the parser generation from the grammar should allow future evolutions including this method.

3.1.2 Related work

Although the descriptions of ISA encoding rules are usually hard coded in disassemblers and assemblers, thus avoiding the need to describe binary instructions in a unified format, there have been other attempts at representing the binary encoding rules of an architecture under such a format. Other works describing instruction sets aim at facilitating higher level manipulations.

3.1.2.1 SLED and GDSL

SLED [86] and GDSL [93] are alternative approaches using structured languages to represent instructions of different architectures. In both cases, instructions and their parts are represented as patterns, allowing to define the encoding and decoding rules for the symbol. The source of the associated functions is generated by translating matching patterns into nested case statements expressed in the target programming language.

SLED allows to define groups of instructions as patterns in addition to more fine-grained elements. GDSL uses a monadic approach where the different parts of an instruction must be declared and typed. The actions performed upon recognising a pattern must also be specified in the description of the patterns, making them dependent on the formalisation of the binary architecture into the language.

Both of these approaches suffer partly from the same drawbacks as the PEG approach, since the resulting decoder would attempt to recursively match all sub elements of a given symbol before switching to the next expression, thus leading to poor performance when using the corresponding parser as a disassembler.

3.1.2.2 LLVM

LLVM [71] is a compiler framework allowing to handle the inner components of a program using a common, low-level, internal representation using a neutral instruction set. It is used by an increasing number of compilers, including the GNU compiler **gcc** [95], for which it was initially written [70].

LLVM also offers components for building target specific back ends and front ends. This includes the LLVM code generator, which allows to translate the internal representation into the assembly or binary code of a given architecture.

LLVM back ends use tables for defining the specificities of instructions with regard to the framework internal representation, including their coding. It is therefore possible to use these tables to generate a disassembler or an assembler for the corresponding architecture. However, the format used for the tables does not cover all possibilities, requiring architecture specific code to be added for handling the assembly and disassembly process. This is for instance the case in ARM, where the absence of opcodes forces some instructions coding to be defined in specific associated source code.

3.1.2.3 ISDL

ISDL [64] is a language allowing to describe architectures for use by retargetable compilers intended to be used by embedded systems. The language represents the ISA as a context-free grammar, which can be processed by Lex and Yacc [72] to generate an assembler. The language is however not intended to be used to allow the generation of a disassembler.

3.1.2.4 SALTO

SALTO [88] is a retargetable framework for building optimisation tools, including analysis and instrumentation. It requires a machine description of the architecture, which can be more or less detailed depending on the need of the tools to build. The internal workings of SALTO rely on this description and are therefore agnostic with regard to the architecture.

SALTO is intended to work on assembly files, either for analysis or for instrumentations. Its machine description may not be used directly for building encoders and decoders.

3.1.2.5 Machine Description Languages

Machine Description Languages are formalisms used to represent instruction sets, focusing on their assembly syntax, authorised formats, and behaviour. These languages include nML [61] [56], MDS [52], or Sim-nML [85]. Although they may use a grammar formalism, their primary purpose is not the description of encoding rules.

An example of building a disassembler from the Sim-nML representation is described in [50]. The disassembler uses a bit by bit matching tree, with possible backtracking, which is inefficient in terms of performance.

3.2 Parsing challenges

One of the most important challenge of parsing binary code stems from the fact that there is no separation between binary instructions, such as spaces in text. On architectures where instructions are of variable length, this prevents the detection of the bytes composing an instruction before decoding it. This can make the identification of symbols during parsing more complicated.

While processors contain embedded decoders allowing them to parse binary code, those are highly specific and optimised for the relevant architecture. Our approach instead aims at offering a generic parser functional for all types for architectures. For those reasons, it is best to use an approach fitting all cases instead of adopting different behaviours depending on the properties of the architecture such as the length of instructions being fixed or not.

We will detail here the challenges brought by the use of a grammar for describing the encoding rules of architecture.

3.2.1 Constraints on grammar

The specific nature of binary code prevents the identification of symbols during parsing through simple means like the detection of a specific terminal.

A direct application of the standard methods for building grammars would be to consider bits as terminals, with possible values being `0` or `1`. Since the encoding formats of binary instructions can contain bit fields of any value, for instance when encoding an immediate operand, a token representing a bit of indeterminate value would also have to be used. However, this definition of terminals would lead a standard parser to attempt detecting bits one by one, slowing down the lexical analysis process.

Since the parser must be able to disassemble binary code for any architecture, it must remain agnostic with regard to the structure of an instruction. It is therefore not possible to presume of the size and position of the important fields inside the encoding of an instruction or to consider that all terminals will always have a fixed length, like a byte. It is not possible either to hard code the looking up of a given field inside an instruction to retrieve specific information concerning the disassembly of the whole instruction.

Another constraint is keeping the structure of the grammar as close as possible to the one employed in the ISA description, in order to make updates easier. Because of the format of some documentations, it is therefore possible that the grammar will contain symbols whose production can overlap with one another.

For instance, architecture documentations can contain two instructions `Insn1` and `Insn2` respectively coded as `0100xxxx` and `01001111`. This can mean either that `Insn2` is a special case of `Insn1` or that the last 4 bits of `Insn1` can never take the value `1111`, for instance if they correspond to the encoding of an operand for which `1111` is not a valid occurrence. This last case can also happen without an instruction such as `Insn2` existing. In the grammar, the symbol representing an instruction would then contain two overlapping productions, as it would be inefficient to create separate cases for all the possible values of the coding of `Insn1`.

3.2.2 Architecture specific challenges

The various architectures of interest to the HPC community present different example of the challenges of disassembly.

3.2.2.1 Challenges of the Intel architectures

A particular challenge of the Intel 64 and Xeon Phi coprocessor architectures (cf. 2.2.4) occurs with the legacy prefixes, which are four families of 1-byte optional prefixes. They can appear in any order and impact the behaviour of the instruction they prefix, but can also be used to differentiate some instructions. Also, while this has no impact on execution, the same prefix can be added multiple times to an instruction by the compiler, usually for alignment purposes. When parsing an instruction, it is therefore necessary to be able to recognise these prefixes independently of their order or possible repetition.

Another complexity concerns the ModR/M byte, which is used in Intel instructions to encode up to two operands, and is potentially followed by another byte, the SIB byte, and a numerical value coded on 1 to 4 bytes, both concerning the representation of an address operand. The presence of these following bytes is deduced from the coding of the ModR/M byte, making it necessary for the parser to analyse it in order to recognise the need to read those additional bytes to complete the decoding of the instruction.

Another challenge may arise when the documentation contains pseudo-mnemonics corresponding to a special use case of some instructions. This is for instance the case in the Xeon Phi coprocessor instruction manual for the vector comparisons instructions `VCMPPS/D`, for which an immediate operand specifies the type of comparison to perform between the other operands. The documentation defines pseudo-mnemonics for each possible comparison, such as `VCMPEQPD/S` or `VCMPLTPD/S`, corresponding to a given value of the immediate operand. This gives one example of the case of a production being a restricted case of another.

As noted in 2.2.4.2, Intel 64 is also a good example of architectures whose evolutions require significant updates to existing disassemblers, as each instruction set adds new encoding rules. For instance, the SSE extension brought the use of legacy prefixes as being mandatory for some instructions, while the AVX extension increased the size of the instruction set by 10% and added a new multibyte prefix mandatory for its instructions, as well as the possible use of the field normally used for storing immediate values to represent a register operand.

3.2.2.2 Challenges of the ARM architecture

The ARM architecture presents different challenges tied to its use of fixed length instructions, as this allows the encoding rules to use any bit in an instruction coding to distinguish it from others.

A significant number of ARM instructions may not accept some operand values or combination of operands; for instance multiple instructions do not accept register `pc` (index 15) as an operand. The encoding rules for ARM instructions may however allow to use the coding that would correspond to the instruction using this operand to represent a completely different instruction instead; in the case of instructions not accepting register `pc` as operand, this would mean that setting the field containing the encoding of the register operand to the binary value `1111`, which would be the

index of the pc register, would have the resulting coding encode another instruction. This can also occur for instructions not accepting a combination of operands, such as having two identical register operands. It is also possible for different instructions to accept a very similar encoding, distinguished only by a bit located near the end of the expression, while only some of those instructions present restrictions with regard to the values of operands.

Therefore, a formalisation of the encoding rules as a grammar needs to take into account those different possibilities, while avoiding duplication of lines as it could lead to a combinatorial explosion.

3.2.3 Additional information

The parser being intended for use by analysis tools, it is important to ensure that its output is able to provide the most complete possible information regarding the architecture. Every information deducible from the documentation and common to all architectures should therefore be available to allow most analyses to remain architecture independent. This information should be added to the grammar for simplified maintenance by keeping it as a single entry point.

3.3 Representation using a grammar formalism

We will describe here our solutions for representing the binary encoding rules of instructions as a grammar. We will also present the algorithm for building the associated parser as a FSA and an improvement of this algorithm allowing to handle more grammars. We will also present the algorithm of the parser.

In the remainder of this document, we will use the term *bit field* for any combination of bits of fixed or unfixed value appearing in the expression of binary encodings. The character `x` in the expression of such a bit field represents a bit whose value is not fixed, or *unfixed bit*. When referring to individual bits in a bit field f, the notation $f[i]$ represents the bit of rank i, the leftmost bit being of rank 0.

3.3.1 Concepts

The lack of priority between the expression of instructions allows the use of a context-free grammar. However, we will have to be able to handle the cases of symbols containing unfixed bits whose expression can overlap with symbols expressed with less unfixed bits. This case may occur not only for whole symbols but for parts of the bit fields appearing in their productions. We handle this by adding a notion of priority between bit fields.

3.3.1.1 Terminal symbol

When building the FSA associated to the parser, we will consider bit fields of any length as terminal symbols. This means that, for the purpose of building the FSA, a terminal symbol may encompass multiple symbols as they appear in the grammar.

For instance, let us consider the production: `A` $\mapsto$ `0000 t 111 S`, where `t` is a token of size 4, and `S` a nonterminal symbol. When building the FSA, the production will be considered as: `A` $\mapsto$ `0000xxxx111 S`, with `0000xxxx111` being handled as a single terminal symbol.

3.3.1.2 Matching bit fields

We will be using the concept of matching bit fields, especially for transition values. Two bit fields are said to be matching if both can take identical values. More formally, bit fields f_1 and f_2 match if:

$\forall i$, either $f_1[i] = f_2[i]$, or $f_1[i] = x$, or $f_2[i] = x$.

3.3.1.3 Binary fields ordering

We will add a notion of priority through a partial order for matching bit fields, according to which a bit field will be superior to the bit fields that represent one or more of its possible values. More formally, the order is defined through the following rules:

- For single bits, $0 < x$ and $1 < x$
- For f_1 and f_2 two bit fields, l being the length of the shortest one:

  $f_1 < f_2 \iff \forall i,\ f_1[i] \leq f_2[i]$ and $\exists i < l / f_1[i] = x$ or $f_2[i] = x$

For example, if bit field f_1 has value `1100` and bit field f_2 has value `11xx`, then $f_1 < f_2$. Bit field f_3 with value `10xx` has no order relation with either f_1 or f_2 as it does not match with either of them.

This partial order can be said to follow the imprecision of bit fields, as bit fields with more unfixed bits on the considered length are of higher order.

We also make the postulate that, for t_1 and t_2 two bit fields to be ordered, at least one of the following propositions is true:

- $\exists i < l \ / \ t_1[i] \neq t_2[i]$ with $t_1[i]$ and $t_2[i] \in \{0, 1\}$
- $t_1 < t_2$, or $t_2 < t_1$

This means we suppose no bit fields such as `x11x` and `0xx0` would appear when we have to order them. For such a case to occur, an architecture would need to contain two different instructions with the same beginning, but at one point one of them would be storing indefinite values in its coding where the other stores fixed bits, and vice versa. This would prevent a successful parsing by the processor decoder.

3.3.1.4 Semantic actions

The semantic actions are defined for each production and represent the operations to execute when performing the corresponding reduction. A semantic action appears as a single function whose parameters can either be constants or values retrieved from the reduction. This allows to choose the language and implementation of the action independently of the grammar, and therefore to implement multiple different uses of the parser from a single grammar file.

When the symbol represents an instruction, some of the macro parameters are mandatory and represent characteristics of the instruction common to all architectures. These parameters include the following information:

- Instruction mnemonic
- Subset to which the instruction belongs if the architecture uses more than one
- Family of the instruction. This regroups the following information:

- Type of operation performed
- Possible impact on the control flow.
- Conditional nature of the instruction
- Vectorial nature of the instructions
- Size of manipulated data

These parameters can be safely ignored when implementing the action if the parser is used to perform basic disassembly, but are essential for more detailed analyses such as the reconstruction of the CFG.

3.3.2 FSA building algorithm

The algorithm for building the FSA is close to the standard, as described for instance in [36], but including the concepts defined in Section 3.3.1.

3.3.2.1 States generation

The automaton is a set of *states* and *transitions* between states. It can be seen as a graph whose nodes are the states and edges the transitions.

A state corresponds to every grammar productions that could be in the process of being reduced at this stage of the parsing. It contains a list of one or more *items*, which are a combination of one possible production for a symbol from the grammar and a position inside this production that we will design as the *item step*. This list contains every possible productions in the grammar to the reduction of which the state corresponds. For example, the item `A` $\mapsto$ `B. 10 C` represents the step of reducing the production `A` $\mapsto$ `B 10 C` after having reduced the symbol `B`.

Transitions are identified by their value, which can be any grammar symbol (using the definition of terminals from 3.3.1.1) or an empty value. As a convention when describing the automaton and its generation, we will consider states to contain a list of transitions, which are all the transitions leading away from this state, and transitions to contain the state to which they point.

For a *reduction state*, the step of the item it contains is at the end of the production. Reaching such a state means the FSA can reduce the corresponding symbol. For instance, the state containing the item `A` $\mapsto$ `B 10 C.` means the `A` symbol can be reduced from its `B 10 C` production.

For a *shift state*, none of the items it contains have their step at the end of the production. Such a state must contain at least one transition.

The states are built recursively from the state corresponding to the first step of the production of the grammar start symbol as described in algorithms 1 and 2.

3.3.2.2 Coherence tests

Once all the states are created, they are tested for uniqueness in order to reduce the automaton. Two states are considered equal if they contain the same list of items, which means they correspond to the same step in the decoding of an instruction. If a state S_2 is found equal to another state S_1, S_2 is removed and all transitions pointing to it are redirected to S_1.

The states are then checked for overlapping transitions. Two transitions over terminals and belonging to the same state are said to overlap if the bit fields representing their respective values are identical over the shortest length between both.

Algorithm 1 Automaton states generation.

```
Require: grammar
states_list ← empty
first_item ← "grammar↦.start_symbol"
first_state ← NewState()
AddToList(first_state.item_list, first_item)
AddToList(states_list, first_state)
ExpandState(first_state)  {ExpandState is described in algorithm 2}
for each state in states_list do
  for each item in state.item_list do
    if IsNotEndStep(item) then
      next_symbol ← NextStepSymbol(item) {B if item is A↦.B C}
      next_item ← NextStepItem(item) {A↦B.C if item is A↦.B C}
      if ListContains(state.transition_list, next_symbol) then
        AddToList(found_transition.next_state.item_list, next_item)
      else
        new_state ← NewState()
        AddToList(new_state.item_list, next_item)
        new_transition ← NewTransition()
        new_transition.value ← next_symbol
        new_transition.next_state ← new_state
        AddToList(state.transition_list, new_transition)
      end if
    end if
  end for
  for each transition in state.transition_list do
    ExpandState(transition.next_state)
    if ListContains(states_list, transition.next_state) then
      DeleteState(transition.next_state)
      transition.next_state ← found_state
    else
      AddToList(states_list, transition.next_state)
    end if
  end for
end for
```

More formally, using the notation $t^l_{a\to b}$ to represent a transition of length l from state a to state b, if l and L are two transition lengths such that $l \leq L$:

$T^L_{s\to s_L}$ overlaps $t^l_{s\to s_l} \iff \forall i < l,\ T^L_{s\to s_L}[i] = t^l_{s\to s_l}[i]$.

In this case, new transition $\tau^{L-l}_{s_l\to s_L}$ is created with a value composed of the rightmost $L-l$ bits from $T^L_{s\to s_L}$. The original transition $T^L_{s\to s_L}$ is then deleted.

For example, if a state contains transition $t^2_{s\to s_1}$ over bit field `00` and transition $t^5_{s\to s_2}$ over bit field `00111`, transition $t^3_{s_1\to s_2}$ over bit field `111` is created while transition $t^5_{s\to s_2}$ is deleted.

Finally, coherence checks are performed on the FSA states:

- Transition values in a shift state must be unique. For the purpose of this test, unfixed bits are considered to be distinct from fixed bits, so for instance the values `00x0` and `00xx` are considered different.

Algorithm 2 Automaton state expansion.

Require: `state`: State to expand with at least one `item` in `state.item_list`

```
 1: for each item in state.item_list do
 2:   next_symbol ← NextStepSymbol(item)
 3:   if IsNonTerminal(next_symbol) then
 4:     for each production in grammar where next_symbol is the head do
 5:       new_item ← NewItem(production) {A↦.B C for A↦B C}
 6:       if IsEmpty(production) then
 7:         new_state ← NewState()
 8:         AddToList(new_state.item_list, new_item)
 9:         new_transition ← NewTransition()
10:         new_transition.value ← EmptyValue()
11:         new_transition.next_state ← new_state
12:         new_transition.length ← ∞
13:         AddToList(state.transition_list, new_transition)
14:         AddToList(states_list, new_state)
15:       else
16:         AddToList(transition.next_state.item_list, new_item)
17:       end if
18:     end for
19:   end if
20: end for
```

- Reduction states must contain only one item and no transition.

The uniqueness of transition values should be ensured by construction and the handling of overlapping transitions. The existence of a reduction state containing more than one item corresponds to a *multiple reduction conflict* and is due to the grammar allowing identical binary expressions to represent different symbols. Similarly, such a state containing at least one transition corresponds to a *shift/reduce conflict*, which is due to a binary expression representing a symbol and part of another. These two types of conflict are due to ambiguities in the grammar, which must be fixed accordingly.

3.3.2.3 Example

We will consider a simplified grammar whose start symbol `A` accepts 4 productions, one of them involving another nonterminal `B`. Two tokens, `c` and `d`, of respective size 4 and 1 bit, are also used. The grammar is expressed as follows, with the semantic actions not represented:

```
%token 4 c
%token 1 d
B:
11 d d
;
A: 1100 c
| 1101 000 d
| 1101 B
| 1100 1100
```

;

We will use simple graphs to represent automaton states, using the transitions as edges. In the first iteration of automaton generation, whose corresponding graph is displayed on figure 3.1, the first state contains the four following transitions:

```
T1: 1100xxxx -> S1
T2: 1101000x -> S2
T3: 1101     -> S3
T4: 11001100 -> S4
```

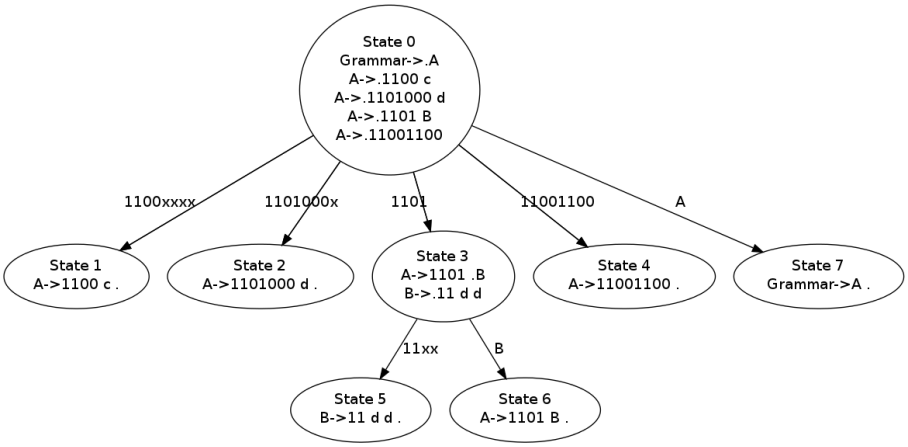

Figure 3.1: First iteration of the FSA generated from the sample grammar.

Since transition $T2$ overlaps transition $T3$, it is deleted during the second iteration and a new transition over `000x` is added to $S3$. The transitions are then ordered using the relative order described in Section 3.3.1.3, leading to the less precise value of $T1$ to be moved after $T4$.

The final transitions for state $S0$ are then:

```
T4: 11001100 -> S4
T1: 1100xxxx -> S1
T3: 1101     -> S3
```

The final version of the automaton is shown on figure 3.2.

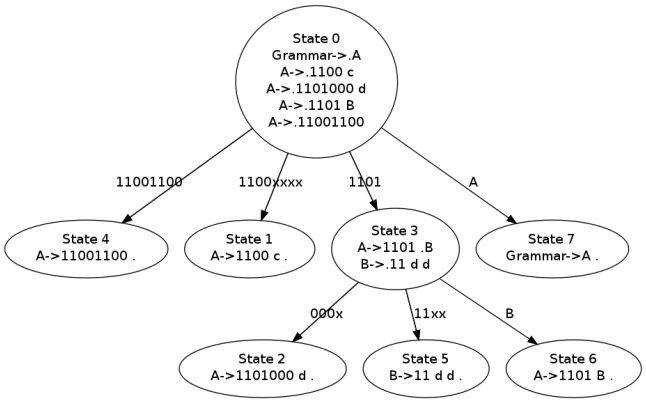

Figure 3.2: Second iteration of the FSA after splitting overlapping transitions.

3.3.3 Parsing algorithm

The parsing algorithm is close to the standard of a FSA built for a LR(0)-parser. The main difference is that the order we imposed on transitions must be the order into which they are tested when deciding which state will follow the current one. This ensures that the more general transitions, with the most unfixed bits in their respective values, will be tested last, and constitute as such an "else" case to the more precise transitions.

3.3.3.1 Storing transitions

The transitions in a state are distinguished between those on a reduced grammar symbol and those on a binary value. Transitions over reduced nonterminal symbols are stored into direct tables indexed on unique identifiers of the symbols, containing the identifier of the next state associated to this transition value. If a state does not contain a transition over a symbol, the corresponding cell in the table contains a special identifier indicating a match failure. In the case of transitions over binary values, the use of a global hash table for speeding up the process was not directly possible here since transitions need to be tested in a given order. In order to ensure better parsing performance, it was necessary to find an alternative to avoid having to test each transition one after another.

The solution involved matching transitions byte after byte, allowing the use of direct tables. To achieve this, transitions over values in a given state are broken down into a series of sub values of identical length for a given index in the transition. The length of the sub values at a given index is the shortest value between 8 bits and the difference between this index and the transition length immediately superior.

A sub value is identified by its value and its mask, which specifies which bits in the value are fixed (the corresponding bit in the mask is set to 1 if the bit is useful, 0 otherwise). Sub values of identical length, index, and preceding sub values are grouped into a table.

A sub value references either a pointer to the table of the sub values that could follow it, or the identifier of the next state if the sub value was the last in a transition.

A list contains all the sub values that could take a given value because of the masks, in the same order the corresponding transitions were ordered in the state.

A table of sub values is a direct hash table containing lists of sub values. Sub values containing unfixed bits appear in multiple cells of the table.

3.3.3.2 Post-parsing actions

In addition to semantic actions, which are performed after reducing a given production, the parser offers to define actions to execute after the successful parsing of a whole word. When the parser is used in a disassembler, a parsed word is an assembly instruction, and it is possible to know precisely which variant from the instruction set at the time of the parser construction. This allows to define actions depending on the mnemonic and operand types of the instruction, which can be used to complement the representation of the instruction or execute specific actions.

3.3.3.3 Parsing a stream

During the parsing of a stream, the automaton processes the bits from the current offset. Once the parsing is complete, either because a word has been parsed or a

parsing error occurred, the automaton is reset and prepares to decode the next word.

In a parsing error occurred, the minimal size of an instruction for the given architecture is automatically skipped before resuming the parsing, otherwise the parser attempts to decode the bits immediately following the end of the previously decoded word. It is however possible to reset the parser anywhere in the stream before parsing the next word. This is useful for instance if the parser is used in a disassembler that does not process the file sequentially or if parsing errors must have a special handling.

When a word is successfully parsed, its associated post-parsing action is automatically triggered if it exists.

The parsing of a word is described in detail in algorithms 3, 4 and 5.

Algorithm 3 Parsing of a word with the FSA.

Require: `InputStream`: Byte stream to parse
Require: `MinWordSz`: Minimum size in bits of a word for this grammar

```
1: parse_error ← false
2: EmptyStack(states_stack)
3: EmptyStack(semantic_actions_stack)
4: Empty(symbol_values)
5: post_parsing_action ← None
6: reduced_symbol ← None
7: while StackTop(states_stack) ≠ final_state do
8:   if IsShiftState(StackTop(states_stack)) then
9:     ProcessShiftState(StackTop(states_stack)) {Cf. algorithm 4}
10:  else
11:    ProcessReducState(StackTop(states_stack)) {Cf. algorithm 5}
12:  end if
13: end while
14: if parse_error == false then
15:   for each action in semantic_actions_stack do
16:     ExecuteAction(action)
17:   end for
18:   ExecuteAction(post_parsing_action)
19: else
20:   SkipsBits(InputStream, MinWordSz)
21: end if
```

3.3.4 Extended FSA building algorithm

There are cases where a LR(0) parser can not be built if the grammar contains symbols whose productions are distinguished from one another by a terminal following nonterminals with matching productions. Such a grammar, while valid, will produce shift/reduce conflicts.

Let us consider the following productions:

```
A: B 00 | C 11
B: 01 | 10
C: 01 | 11
```

Algorithm 4 Handling a FSA shift state.

Require: `state`: Shift state to process
Require: `reduced_symbol`: Previously reduced symbol

```
1: if reduced_symbol ≠ None then
2:   if TransitionMatch(state.symbol_transitions, reduced_symbol) then
3:     StackAdd(states_stack, matching_transition.next_state)
4:     reduced_symbol ← None
5:   else
6:     parser_error ← true
7:   end if
8: else
9:   subtable ← state.sub_values
10:  next_state ← Unknown
11:  while next_state == Unknown do
12:    test_bits ← ExtractBits(InputStream, subtable.values_bitlength)
13:    if input_stream.bitlength < subtable.values_bitlength then
14:      parser_error ← true
15:    end if
16:    if TransitionMatch(subtable, test_bits) then
17:      if NumberElements(matching_sublist) == 1 then
18:        submatch ← true
19:        subvalue ← matching_sublist.value
20:      else
21:        if TransitionMatchInList(matching_sublist, test_bits) then
22:          submatch ← true
23:          subvalue ← first_matching_value
24:        else
25:          parser_error ← true
26:        end if
27:      end if
28:    else
29:      parser_error ← true
30:    end if
31:    if subvalue.next_state ≠ None then
32:      next_state ← next_state
33:    else if subvalue.next_subtable ≠ None then
34:      subtable ← next_subtable
35:    end if
36:  end while
37:  if parser_error == false then
38:    StackAdd(states_stack, next_state, test_bits)
39:    SkipsBits(InputStream, test_bits.bitlength)
40:  end if
41: end if
```

These productions are coherent and can be expanded without conflict into the values: `0100`, `1000`, `0111`, `1111`. However, when building an FMA as described in Section 3.3.2.1, the transition on the `01` bits in the state generated from the

Algorithm 5 Handling a FSA reduction state.

Require: `state`: Reduction state to process

```
1: for each element in states.reduction do
2:   if IsToken(element) then
3:     value ← element.bits
4:     StoreToken(symbol_values, element.token_name, value)
5:   end if
6:   StackRemoveTop(states_stack)
7: end for
8: StackAdd(semantic_actions_stack, states.reduction.semantic_action)
9: if state.post_parsing_action ≠ None then
10:  post_parsing_action ← state.post_parsing_action
11: end if
12: reduced_symbol ← states.reduction.head
13: if reduced_symbol = start_symbol then
14:   StackAdd(states_stack, final_state)
15: end if
```

productions of the `A` symbol will produce a shift/reduce conflict, as it will not be possible to know whether the `B` or `C` symbol must be reduced without checking the following bits.

This case occurs in binary architectures such as Intel Itanium, where instruction bundles are of a fixed 128-bits length, and differentiated on the last 5 bits, while possibly containing overlapping values and symbols in the previous bits. It is also the case in ARM architectures, especially if the grammar uses different nonterminal symbols to represent the fields encoding register operands for which certain values are excluded.

These kind of conflicts can be resolved with the use of an extension of the LR parser performing the validation of a transition by looking more symbols ahead, which is similar to the LR(x) parser. Such an extension is harder to implement for binary code because of the variable length of the terminal symbols, and is usually not possible for architectures where instructions are of different lengths. The adopted solution attempts to use an agnostic approach allowing the same algorithm to be used on grammars representing all types of architectures.

3.3.4.1 New concepts

The extended parser redefines some of the concepts used in Section 3.3.2.1.

States A state is here uniquely defined as a list of items and a number of bits matched ahead of the items steps. The step of an item can also be located inside a terminal. It is considered that all bits preceding the items step have been successfully matched, while the bits tested ahead are not contiguous to the position of the step and may not be contiguous with one another. For shift state, a mask of bits to test is calculated from the items it contains, which will be used to generate its transitions over terminals. The tested bits of a state are generated from the tested bits of the preceding state updated by the bits from the transition value leading to it.

Transitions A transition over a terminal can now contain gaps between the bits. They correspond to the testing of only some of the bits following the position of the parser. The transitions over terminals from a given state all contain the same number of bits and gaps and are generated from the mask of bits to test for this state applied to its list of items.

Symbol sizes The implementation relies on the computing of the size of symbols. The size of a symbol is the sum of the sizes of all the terminal and nonterminal symbols of its productions. A symbol is flagged as being of variable size either if two of its productions have different sizes, or if at least one symbol in one of its productions is of variable size.

3.3.4.2 States generation

The main difference in this version of the parser generation resides in the algorithm for expanding states. The derivation of the items they contain is performed recursively until a valid mask of bits to test is found for all items. Items are distinguished between *non derivable items*, for which the symbol following the step is a terminal, and *derivable items*, for which this symbol is a nonterminal. Non derivable items are always taken into account for generating the mask, while derivable items are used only if they would not cause the mask to be emptied and are derived otherwise.

This mask is calculated by comparing bits between the step and the first nonterminal of unfixed size or the end of the production in all items contained in the state in order to find those that have a fixed value for every item. Derivable items are included in this calculation only if they contain at least one such eligible bit at the same index from the step as the other items. The generation of the mask from the list of items obeys the following rules:

- Nonterminals of fixed size occurring in at least one item are represented in the mask by gaps (ignored bits) of the same size.
- Bits fixed in all items are marked as to test in priority.
- Bits unfixed in at least one item are removed if the mask contains at least one bit marked as to be tested in priority.

Since transitions over terminals are all generated from the same mask, the handling of overlapping transitions described in 3.3.2.2 is unnecessary.

A shift/reduce state, containing a single reduction item and at least one transition, is valid in the extended algorithm if the transitions over terminals do not cover all possible values reachable with the state mask. It is handled as a shift state defaulting to a reduction state if none of the transitions match.

The expansion of states for the extended parser is described in algorithm 6.

3.3.4.3 Storing transitions

The breakdown of transitions over terminals as described in 3.3.3.1 is modified to ensure that the most significant parts of transitions are tested first, instead of matching bytes sequentially. This is done by detecting among all transitions in a state the bits whose value is unfixed or ignored for a minority of transitions, then using this order to build the bytes used for breaking down transitions into sub values. The storing of sub values into tables is otherwise unchanged.

Algorithm 6 Extended automaton state expansion.

Require: `grammar`: The grammar for which the automaton is being built
Require: `state`: State to expand with at least one `item` in `state.item_list`

```
1: testmask ← EmptyValue()
2: derivable_list ← EmptyList()
3: for each item in state.item_list do
4:   if IsDerivable(item) then
5:     AddToList(derivable_list, item)
6:   else
7:     testmask ← UpdateMask(testmask, item)
8:   end if
9: end for
10: repeat
11:   derivation_occurred ← false
12:   OrderList(derivable_list) {Order by increasing distance from the step to
      first fixed bit and decreasing distance between step and end of production or
      first nonterminal of unfixed length}
13:   for each item in derivable_list do
14:     if UpdateMask(testmask, item) == EmptyValue() then
15:       for each production in grammar where NextSymbol(item) is head do
16:         new_item ← NewItem(production) {Same derivation as described in
            algorithm 2}
17:         if IsDerivable(new_item) then
18:           AddToList(derivable_list, new_item)
19:         else
20:           testmask ← UpdateMask(testmask, new_item)
21:         end if
22:       end for
23:       RemoveFromList(derivable_list, item)
24:       derivation_occurred ← true
25:     else
26:       testmask ← UpdateMask(testmask, item)
27:     end if
28:   end for
29: until derivation_occurred == false
30: state.testmask ← testmask
```

3.3.4.4 Example

Let us consider the following grammar:

```
Start: A B 01 | 000000 | C
A: 00 | 01
B: 00 | 10
C: 111111 | 110110
```

The parser generation algorithm described in 3.3.2.1 would cause a shift/reduce conflict, as an input beginning with bits `00` could be interpreted either as the beginning of the `Start->000000` production or as the `A->00` production.

With the extended parser generation algorithm, the first generated state will contain the following items:

```
Start->.A B 01
Start->.000000
Start->.C
```

Using symbol '?' to represent a bit to be tested, the generation of the mask of bits to test for this state will take the following steps:

1. Initialisation from the non derivable items to the value `000000`
2. Merging with derivable item `Start->.A B 01` results in value `____0?`
3. Merging with derivable item `Start->.C` would blank the mask: the item is derived and excluded from further calculations
4. Merging with newly added item `C->.111111` results in final value `____??`

The final version of the automaton is shown on figure 3.3.

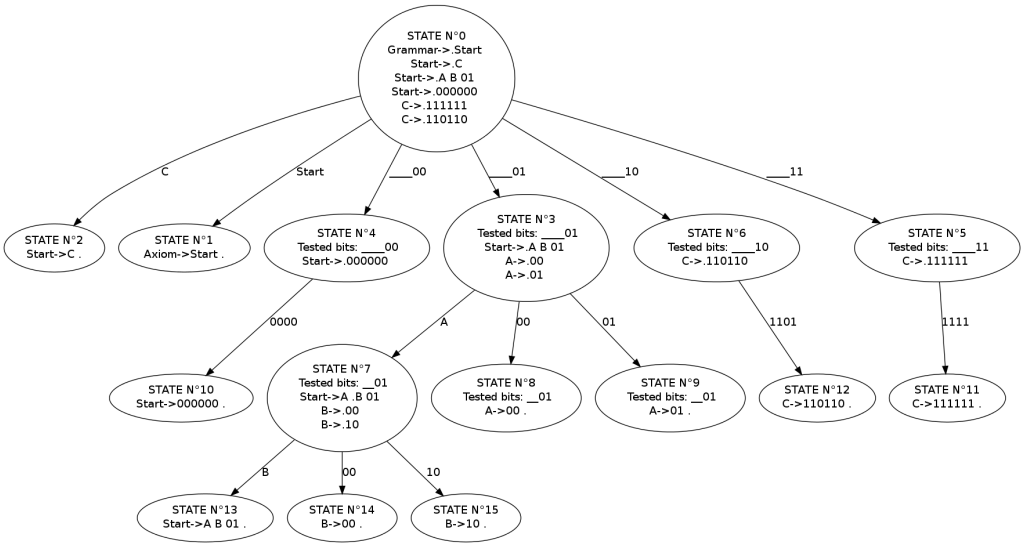

Figure 3.3: Automaton generated through the extended algorithm.

3.3.4.5 Extended parser execution

The execution of the extended parser is close to the description presented in 3.3.3.3.

The main difference consists in the extraction of bits from the input stream, which may involve the retrieval of distinct bits or groups thereof and not immediately following the current position of the parser.

Hybrids shift-reduce states are handled as shift states, with an additional step consisting in saving the parser state to a stack. A subsequent failure to match any of the transitions will cause the reloading of the parser state from this stack and the execution of the reduction from the shift/reduce state instead of raising a parser error. This stack is emptied after the successful parsing of a word.

3.4 Minjag

In order to validate the principles detailed in Section 3.3, we implemented the tool Minjag, which allows to parse a grammar and generate the associated parser according to the algorithms described in that section.

Minjag includes a script allowing to build a grammar from the list of instructions of a given architecture through simple transformations. In order to keep the grammar as a single entry point for defining the architecture, additional information about the architecture is also stored into the header of the grammar file. This information includes the names and types of registers used by the architecture, and the maximum and minimum sizes in bits of an instruction, which may not be possible to deduce from the grammar if it contains recursive symbols. Minjag then parses the grammar and builds an FSA for a LR parser following the extended generation algorithm described in 3.3.4, then generates the code needed to execute it, defining the states and transitions of the FSA as C structures.

Minjag also parses the semantic actions, which must follow the constraints described in 3.3.1.4. It then generates definitions of C macros for each semantic action, and provides a list of those definitions among the generated files. The generated FSA is built so that, if a macro is defined, its associated code will be executed whenever the corresponding semantic action must be triggered. This is used to define actions to perform during disassembly without impacting the grammar or the parsing process, allowing to define a grammar file for a given architecture once and be able to adapt its associated parser to different applications.

Minjag provides a list of macros allowing to define the post-parsing actions if needed (cf. 3.3.3.2). It also generates files defining structures characterising the architecture, allowing to easily retrieve the names of mnemonics, types and size of registers, or additional information about the behaviour of instructions. Finally, Minjag generates the code for an encoder for the architecture, described in 3.4.3.

The grammar format accepted by Minjag is described extensively in annex A.3. It is close to the Backus-Naur format [77] used for context-free grammars.

Minjag was able to process grammars representing the Intel 64, Intel Xeon Phi coprocessor and ARM architectures. Table 3.1 presents general metrics on the description of the currently existing architectures.

Architecture	Intel 64	Intel Xeon Phi	ARM
Lines in instruction list	2,398	1,194	1,512
Lines in grammar	6,082	3,082	1,491
Reduction states	5,950	2,406	1,625
Shift states	4,019	1,468	2,916
Shift/reduce states	2	2	6
Total states	9,971	3,876	4,547

Table 3.1: Characteristics of the grammar and resulting FSA for the Intel 64, Intel Xeon Phi coprocessor, and ARM architectures.

3.4.1 Specificities of the Intel architectures

Minjag is able to process a grammar addressing the particular challenges of Intel architectures presented in 3.2.2.1.

Legacy prefixes are represented through a separate symbol for each family. Instructions for which a prefix of a given family is mandatory are grouped under the same symbol. The productions corresponding to the encodings of those instructions do not contain any such prefix. These symbols are then used in the productions of higher-level symbols corresponding to encodings containing one prefix family. By combining these symbols, it is possible to represent the random order of those prefixes. Since the grammar supports left recursion, it is also possible to handle multiple instances of an identical prefix. Below is a simplified example using only two prefix families, where the productions corresponding to encodings of prefixes or instructions and the semantic actions are not represented.

```
Start:Legacy1 Insn_Legacy1 | Legacy2 Insn_Legacy2 | Insn_NoLegacy;
Insn_Legacy1: Legacy2 Insn_Legacy1_2 | Insn_NoLegacy;
Insn_Legacy2: Legacy1 Insn_Legacy1_2 | Insn_NoLegacy;
Insn_Legacy1_2: <instructions for which both families are mandatory>;
Insn_NoLegacy: <instructions for which both families are optional>;
```

The partial order of matching bit fields allows to directly represent the opcodes that are restricted cases of other instructions as well as the ModR/M byte with a minimal level of complexity.

Below is a simplified extract of the grammar for Intel 64:

```
%token <3,b> reg
%%
template: Legacy3 Insn       #[ FULLINSN($1,$2) ]#
| Legacy3 template #[ FULLINSN($1,$2) ]#
| Insn             #[ FULLINSN($1) ]#;
MemModRM: 00 reg RMSIB_00  #[ REG_MEM($1,$2) ]#
|01 reg RMSIB_01 #[ REG_MEM($1,$2) ]#
|10 reg RMSIB_10 #[ REG_MEM($1,$2) ]# ;
RegModRM: 11 reg RMSIB_11  #[ REG_REG($1,$2) ]# ;
Insn: 00010000 RegModRM #[ INSN(ADC,ADD,NA,REG($1),REG($1)) ]#
|REX 00010000 MemModRM #[ INSN(ADC,ADD,NA,REG($1,$2),MEM($1,$2)) ]# ;
```

In this example, the `reg` symbol is declared as a token of length 3 with significant bit first endianness. The `template` symbol is the start symbol of the grammar. It indicates that the `Insn` symbol accepts an optional prefix, defined as the `Legacy3` symbol (not represented in this extract), possibly repeated. The semantic actions associated to the `Insn` symbol productions correspond to instructions. In the example, the lines present describe the `ADC` mnemonic, which belongs to the family of `ADD` operations, and the `NA` constant indicates it has no impact on the control flow.

The shift/reduce states allow to handle the case of macro instructions representing multiple instructions. Those states contain a reduction corresponding to the first instruction, while the transition corresponds to the next instructions in the macro. During parser execution, matching on the transitions will result in the continuing processing of the macro instruction, while a matching failure will result in the reduction of the first instruction of the macro only. The reduction of the next instructions in the macro only is performed by another state.

Figure 3.4 presents a simplified version of a generated FSA containing an example of the handling of a macro instruction. The instructions used in this example are `FWAIT` with coding `0x9B`, `FNINIT` with coding `0xDBE3` and macro instruction `FINIT` with coding `0x9BDBE3`, all from the x87 instruction subset.

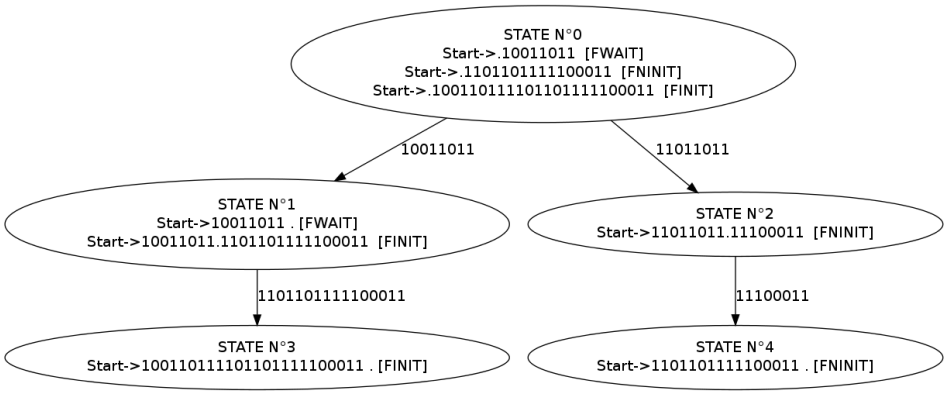

Figure 3.4: Part of an automaton handling instructions `FWAIT`, `FNINIT` and the macro instruction `FINIT`. State 1 is a shift/reduce state; if the transition over value `1101101111100011`, which would lead to the reduction of symbol `10011011110110111111100011` corresponding to macro instruction `FINIT`, fails, symbol `10011011` (`FWAIT`) is reduced instead.

3.4.2 Specificities of the ARM architectures

The extended FSA generation algorithm allows Minjag to process a grammar addressing the particular challenges of ARM architectures presented in 3.2.2.2.

Bit fields that do not accept some values, used to represent the encoding of operands for which certain values are excluded, are represented as different symbols containing all possible values of the bit fields as productions. For instance, the symbol representing the 4 bits field for coding a register operand for which the `pc` register (index 15) is not a valid option contains for productions all 4-bits expressions except `1111`. Those symbols were used appropriately in the productions representing instructions for which an operand or combination of operands was not valid.

The possibility for transitions to contain non contiguous bits allows Minjag to handle encodings distinguished by bits located near the end of the production.

3.4.3 Assembler generation

The format chosen for the grammar also allows to use the same file to generate the source files of an encoder which can be used to build a simplified assembler. It functions on the same principles as a packrat parser [59].

3.4.3.1 Building the encoder

The encoder is built by reversing the behaviour of the grammar semantic actions and productions.

A *reversed semantic action* is responsible for checking whether a given input matches with what the action represents in the grammar, and for extracting from it the information needed to build the binary expression.

Encoding rules are defined for each grammar symbol and consist in a list of reversed semantic actions representing its possible productions. If the architecture allows for multiple instruction sizes, this list is ordered by increasing size of the resulting binary expression to emulate the behaviour of mainstream assemblers.

A special mechanism handles the case where the input of the encoder is not a production of the start symbol of the grammar, which occurs for instance with our

grammar for Intel 64 due to the way legacy prefixes were handled. In that case, a path to the start symbol is built, and an *upward encoding rule* is created from the corresponding list of reversed semantic actions.

Finally, MINJAG generates for each instruction a list of reversed semantic actions consisting in the possible encodings for this instruction, which will be used as entry point. This allows to handle an instruction to be encoded as another grammar symbol, using this list as its encoding rules.

3.4.3.2 Encoder execution

When run as an assembler, the encoder input is a structure representing the instruction to be assembled. The list of encoding rules associated to this mnemonic is retrieved, and the encoder attempts to process its input following the same algorithm as for encoding a symbol. This consists in attempting to match the input with each of the encoding rules, then complement the encoding with the data returned by the reverse semantic action and encoding the nonterminals from the associated production. This recursive process is detailed in algorithms 7 and 8.

Algorithm 7 Encoding a symbol.

Require: `symbol`: Representation of data to encode as a grammar symbol

```
1: encoding ← null
2: for each rule in symbol.encoding_rules do
3:    if RuleMatches(rule, symbol) then
4:       ApplyRule(rule, symbol, encoding) {Cf. algorithm 8}
5:    end if
6: end for
7: if encoding ≠ null then
8:    for each rule in symbol.upward_encoding_rules do
9:       ApplyRule(rule, symbol, encoding) {Cf. algorithm 8}
10:   end for
11: end if
12: return encoding
```

The result is a structure representing the binary encoding of the instruction, or a null value if it was not found to be matching a valid instruction for this architecture.

3.4.4 Grammar checks and debugging

During its parsing of the grammar, MINJAG performs a certain number of checks to ensure that basic consistency constraints are obeyed by the grammar and automatically handles some of them to allow avoiding common errors during FSA generation.

In particular, MINJAG automatically detects identical productions of a given symbol. These cases are logged, and those duplicates are discarded from the lists of productions when building the parser. The associated semantic actions are however preserved and taken into account when generating the encoder as described in 3.4.3. This allows to support the case of different mnemonics or instructions performing identical operations by eliminating the redundant expressions from the parsing options while supporting all of them for encoding.

Algorithm 8 Applying a successful reverse semantic action.

Require: `symbol`: Representation of data to encode as a grammar symbol
Require: `encoding`: Partial encoding of the symbol

```
1: for each element in rule.production do
2:   if element.type == terminal then
3:     AddCoding(encoding, element)
4:   else if element.type == nonterminal then
5:     element_code ← EncodeSymbol(element) {Cf. algorithm 7}
6:     if element_code ≠ null then
7:       AddCoding(encoding, element_code)
8:     else
9:       encoding ← null
10:    end if
11:  end if
12: end for
13: return encoding
```

Minjag performs additional coherency tests on the grammar and logs the results. These include the detection of symbols not reachable from the start symbol, which causes a warning, and of symbols appearing in productions that are undefined as either nonterminals or tokens, which causes an error. Errors are also logged if the consistency checks performed on the resulting FSA (cf. 3.3.2.2) fail.

Finally, Minjag can generate a file describing the states of the generated automaton, including the transitions and items each of them contain, along with the associated grammar lines. This file can be useful for debugging the grammar when FSA consistency checks fail by allowing to pinpoint the overlapping productions causing shift/reduce or multiple reduction errors.

3.4.5 Exhaustive tests of architecture representation

The list of instructions the grammar represents can be subjected to similar transformations to generate test files containing exhaustive combinations of the architecture mnemonics and operands. This allows to build a library of all possible instructions existing in the architecture. Such a library allows to serve two purposes:

- Check the validity of the instruction list
- Check the validity of the generated grammar

Checking the instruction list can be necessary as documentations can contain errors or imprecisions. By generating assembly files and attempting to run them through an assembler like GNU **gas**, it is possible to detect by catching assembly errors the instructions that were incorrectly represented in the list, either in their coding or in their acceptable operands. Further tests could involve the generation of executables from these assembled files in order to detect instructions that were assembled into code actually not supported by the processor.

The validity of the grammar can be checked by attempting to disassemble the assembled files with the parser and comparing the result with the original assembly input. Parsing errors or divergences with the assembly code will signal errors in the representations of instructions in the grammar. Comparisons with the assembly

input requires being able to detect special cases such as homonyms, since different mnemonics with identical coding will be disassembled as the same mnemonic in each instance, causing a difference with input without the parser being faulty.

For architectures accepting a large number of instructions and/or operands per instruction, the complete list of possible instructions can be excessively large. In these occurrences, the generation needs to be restricted to cases significant for the purposes of testing; it is for example not necessary to generate instructions whose encoding differs from others by a clearly identified number of bits, such as the encoding of a register operand.

As an example, such a significant subset of the possible instructions in the Intel 64 architecture numbers approximatively 400,000 instructions, while the equivalent subset for the Xeon Phi coprocessor architecture, which counts more instructions using 4 operands, numbers more than 1,200,000 instructions.

3.5 Conclusion

We have presented here a solution for representing instruction sets as a context-free grammar. The format used and the associated algorithm for generating the parser allow to take into account the specificities of binary code. This provides the means for automatically generating parsers for any architecture from a simple text description of the format and to keep them up to date with the architecture evolutions. The use of a grammar also allows to adapt the generated parser for different applications while keeping the same representation of the architecture. We also presented a functional implementation of these principles as the Minjag tool. Grammars describing the Intel 64, Intel Xeon Phi and ARM architectures have been written and validated by Minjag.

In the next chapter, we will present an application of the parser as a disassembler intended to be used as an entry point by analysis tools. This allows to test the validity and accuracy of the parser as well as its capacity to be customised to provide additional information as needed.

Chapter 4

Disassembly of binary files

Disassembling a file consists in the translation of its binary code into assembly code. In order to facilitate subsequent analysis of the code, this implies being able to retrieve not only the instructions but also any additional information available in assembly files and that was preserved in the binary file. For instance, the retrieval of labels can allow to have a first insight into function boundaries; however, as noted in 2.3.2.1, not all symbols or assembly directives appearing in a compiler-generated assembly file are preserved when generating an executable.

The operations performed by a binary disassembler can be broken down into the following steps:

1. Parsing the file to extract the executable code as well as any additional information useful to further code analysis.

2. Parsing the binary code of an instruction and return the corresponding assembly instruction.

3. Parsing a binary stream to return the list of instructions it contains and complement it with any additional information available in the file.

Step 1 is done through a parser of the binary format used for the file. As the implementation of such a parser is straightforward provided the complete specification of the relevant format is available, we will not focus on this part and consider that all information present in the file is available during the disassembly process. Section 2.3.3 presents an overview of most binary formats.

Step 2 requires the implementation of a parser for the architecture for which the binary code is defined. Chapter 3 described how to generate the code of such a parser from a representation of the architecture. Using this code for a disassembler simply requires defining the semantic actions of the parser to build structures representing instructions and containing all information available in the grammar.

Step 3 consists in the successful application of the parser to the extracted binary code, correlated with any additional information available in the file. This task can be complicated if the binary code is interleaved with data not representing instructions, which is possible even though binary formats separate code sections from data sections. This will be the main subject of this chapter.

Since we intend the disassembler to be used by analysis tools, special care must also be brought to its performance in terms of speed and precision. Avoiding excessive disassembly times is challenging as analysed programs can contain millions of instructions but necessary to prevent the disassembly stage from becoming the bottleneck of a potentially lengthy analysis process. Precision is also a crucial factor, as erroneous instructions could mislead analyses about the behaviour of the program. Another constraint is the ability to offer an architecture independent abstraction of the assembly code by ensuring that the base output format of the disassembler covers the elements common to assembly languages of different architectures.

In this chapter, we present the challenges of disassembly and our solutions. We will first focus on the challenges from the application of a parser on a binary stream in Section 4.1, then describe common algorithms addressing these issues and present existing disassemblers in Section 4.2. We will then present our solutions in Section 4.3, and their implementation and performance in Section 4.4 and 4.5.

4.1 Disassembly challenges

As noted in 2.2.1, the encoding of binary instructions does not include a separating character between instructions. Therefore, for architectures accepting instructions of different lengths, the only way for identifying the beginning of an instruction is to successfully disassemble the previous one. If the parser used for recognising individual instructions fails to correctly decode one of them, it may not successfully identify its end, and thus attempt to disassemble the next instruction while still in the middle of the coding of an instruction. Such an operation would lead the disassembler to return at least two erroneous instructions or parsing errors.

4.1.1 Interleaved foreign bytes

Parsing errors can occur if the parser encounters binary data not representing assembly instructions intertwined with the regular binary code. Although binary formats specify separate sections for executable code and other data, there is nothing to prevent a binary file to mix them. In fact, since the only constraint for the execution of a program is that the control flow is not interrupted and the executable code is correctly loaded to memory, it would be possible for the code to be spread anywhere in the file. In such a case, branch instructions are responsible for keeping the control flow from reaching addresses not containing executable code.

Since disassemblers operate statically, possibly with no information on the instruction set other than their binary coding, they may not detect the role of those branches and attempt to disassemble bytes that do not actually represent instructions. This can cause the disassembler to return parsing errors when attempting to process those areas, or to incorrectly disassemble correct instructions that followed them as the parser missed their beginning. Conversely, since data bytes can potentially take any value, the disassembler could be able to successfully parse such a code into an instruction. It would then return an incorrect list of instructions containing elements that are not actually part of the executable code.

Such disassembly errors may not be detected easily as some binary formats, especially Intel x86, exhibit properties allowing disassembly to be self-repairing, as demonstrated in [81] and illustrated in [74]. This means that, if the parser failed to correctly identify the beginning of an instruction due to one of the cases described above, it will eventually resynchronise with their proper boundaries and return subsequent instructions correctly, provided it is still in an area containing executable code. Architectures where the instructions have a fixed length, such as ARM or Power ISA, usually also impose constraints on their alignment, ensuring that the length of any foreign bytes interleaved with code allows to preserve the alignment of the next valid instructions. If these bytes happen to match the coding of valid instructions, they will be erroneously decoded as instructions without raising any parsing errors. It is therefore possible for the disassembler to return little to no parsing errors while encountering ranges of executable code interleaved with other

foreign bytes, making an automated detection of such areas complex.

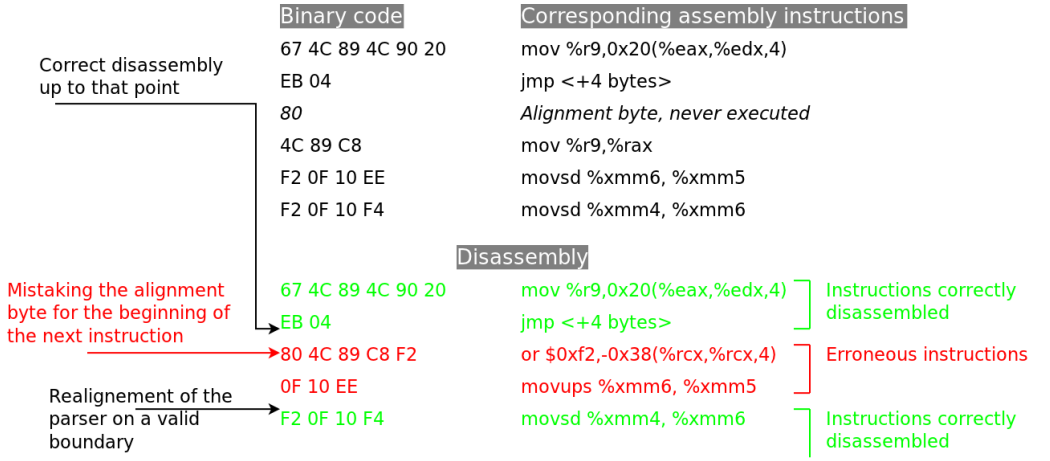

Figure 4.1: Example of a disassembly being thrown off course and returning erroneous instructions. The code used is Intel 64 with instructions of different binary length.

Figure 4.1 presents how a disassembler processing sequentially can return erroneous results but no parsing errors when attempting to disassemble binary code not representing instructions, in this case only a single byte long. The self-repairing properties of Intel 64 can also be observed on this example, as the disassembler is correctly realigned at the end of the second erroneous instruction. This example also demonstrates the impact such errors can have on further analyses, as the disassembler returns here a non-existent memory access and an instruction with the right operands but an incorrect mnemonic.

In most cases, the foreign bytes interleaved with encoded instructions represent data used by the executable. A code could for instance keep the value of variables close to the instructions using them for optimisation purposes.

A common occurrence for this is represented by jump tables, which contain the different possible values for the destination address of an indirect branch (cf. 2.2.3 and [47]). Such tables allow to avoid multiple conditional branch instructions since only a calculation of the index of the cell in the array is needed to identify the target. This mechanism can for instance be a straightforward compilation of a `switch` statement. Some compilers, like the Intel C compiler **icc**, use a table directly present in the code of customised versions of standard functions that are automatically added to the generated executables.

Another reason for the presence of foreign bytes in executable code is the use of padding to enforce alignment constraints. Compilers usually achieve this by inserting legitimate instructions (`nop` or similar), but they could potentially use any other binary value if the control flow has to skip these bytes, such as in the example in figure 4.1.

4.1.2 Obfuscated code

Some binary codes may have been purposefully altered to hamper their disassembly, usually for code protection or security purposes. The code is then said to have been obfuscated; [74] references some known methods for achieving this. Code obfuscation can also refer to the process of making the assembly code itself more challenging to

analyse, but we will not cover this here.

A common obfuscation method is the addition of "junk" bytes to the code, reproducing the symptoms of foreign bytes interleaved with executable code. A refinement of the method consists in choosing those bytes so that disassemblers become desynchronised with the actual code and return the longest possible sequences of incorrect instructions. Figure 4.2 present an example of how an obfuscated code can mislead a disassembler into identifying an instruction very different from the actual one.

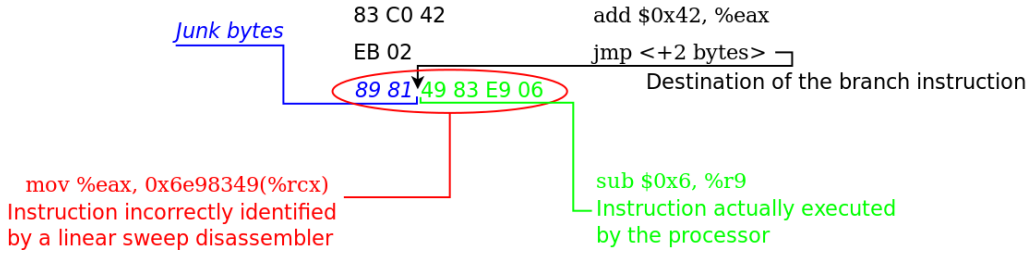

Figure 4.2: Example of an obfuscated Intel 64 code. If the disassembler misses the `jmp` instruction used to skip the junk bytes, it will process them and the following instruction as a single instruction very different from the actual one.

Other obfuscation methods consist in mixing executable code inside the structure of the binary file, for instance using reserved and unused bytes in headers to store instructions, or obfuscating the binary format itself so that sections in the file are harder to identify while leaving the necessary information for execute it. [80] details some of those methods. They do not affect the disassembly itself but make the retrieval of the binary executable code in the file more complex.

4.1.3 Self rewriting code

Another method commonly associated with obfuscation is the use of self rewriting code, which overwrites itself with binary data corresponding to the encoding of the actual instructions to execute. A self rewriting code does not prevent disassembly, but causes the disassembled instructions to differ from those that will be actually executed, and only a thorough analysis of the disassembled code may allow to identify the actual instructions. Such an analysis may not even be possible without a full simulation, for instance in the case of a code modified at each iteration of a loop. Self rewriting code may also appear in a non obfuscated code to reduce its size while effectively storing multiple versions in a single file. Figure 4.3 presents an example of self rewriting code.

4.1.4 Overlapping instructions

Another method that may also be used in non obfuscated codes consists in using overlapping instructions. This method is especially relevant with instructions sets of variable length, like Intel x86. If the coding of an instruction overlaps with another, it is possible to encode the largest in the file and select the instruction to execute through branches pointing to the relevant part in the coding. The most common use of this is for instructions using prefixes to distinguish between different opcodes. Branching the flow after the prefix results in the instruction not using it being executed. This method is close to self rewriting code in its effects, in that the code can be successfully disassembled but does not match the code actually executed.

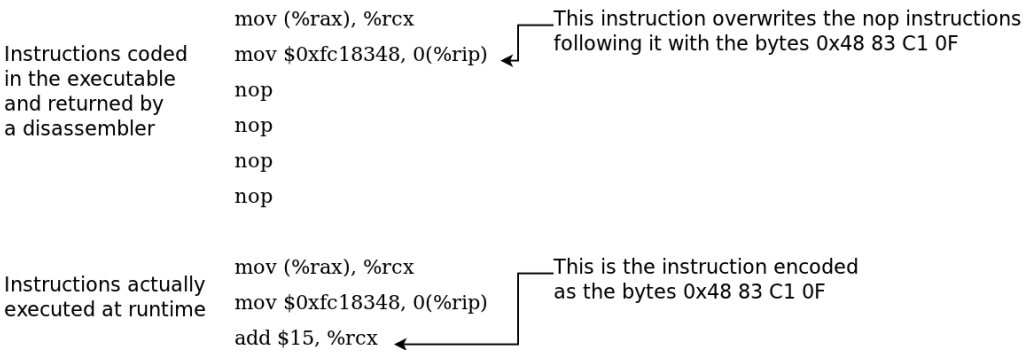

Figure 4.3: Example of a simple self rewriting Intel 64 code for a single instruction.

Figure 4.4 presents an example of this technique executing a different instruction in the first iteration of a loop than for the following iterations.

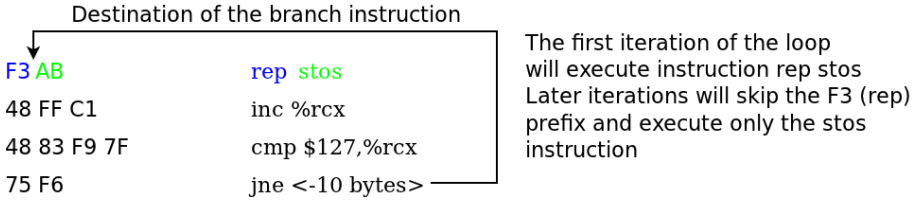

Figure 4.4: Example of the use of overlapping instructions in Intel 64 code involving the `stos` and `rep stos` whose opcodes differ only by a prefix.

4.1.5 Output format

One last challenge of the disassembler concerns its output format. Since the disassembler is chiefly intended to be used by analysis tools, it is important to ensure that its output allows to cover most assembly languages and offers standard information needed by analysis algorithms. Assembly languages present common characteristics: an instruction is composed of a mnemonic with a variable number of operands, each operand being either a register, a memory reference, or an immediate value. Since mnemonics or register names and types vary between architectures, the output representation must be able to provide information as to the instruction operations and impact on the control flow without relying on names.

4.2 Disassembling principles and related work

We will describe here the common principles involved in disassembly and present some existing disassemblers.

4.2.1 Disassembly methods

There are two main approaches for disassembling a file, *linear sweep* and *recursive traversal*, described in [91] and [73]. Since both methods present complementary strengths and weaknesses against disassembly challenges, disassemblers aiming at an extensive coverage of disassembled codes and at overcoming obfuscation techniques usually implement a combination of the two. Self rewriting code (cf. 4.1.3) is

not handled by either method, as in this case only an extensive analysis of the disassembled code allows to correctly identify the code to be actually executed.

4.2.1.1 Linear sweep

Linear sweep disassembly consists in a sequential parsing of the binary code, assuming that encoded instructions follow each other without interruption. Figure 4.5 presents an example of the parsing of binary code using this method.

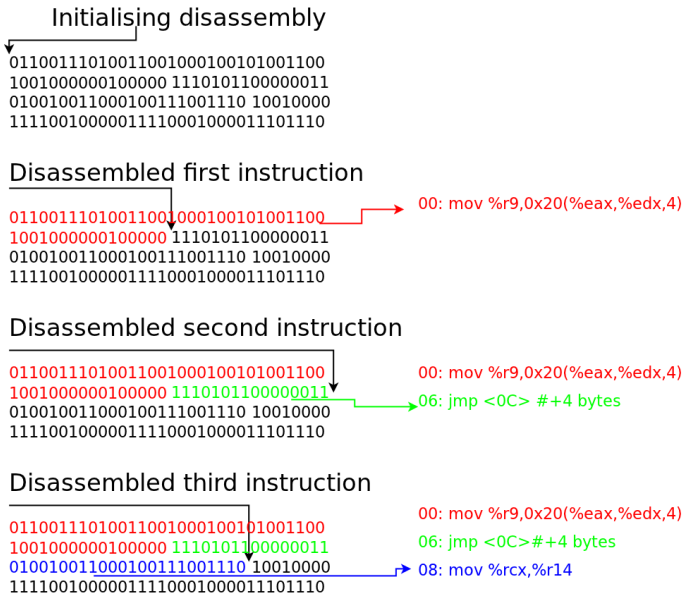

Figure 4.5: Example of a linear sweep disassembly on Intel 64 code.

Linear sweep ensures that all the binary code extracted from the file is parsed, but is unable to overcome most of the challenges described in Section 4.1.

For instance, foreign bytes interleaved with code will throw a linear disassembler off course and generate parsing errors or cause it to report erroneous instructions, as described in figure 4.1. This method is also vulnerable to most obfuscation techniques, especially those involving the insertion of "junk" bytes.

4.2.1.2 Recursive traversal

Recursive traversal disassembly consists in the parsing of the binary code following the control flow. A disassembler using this method performs a linear sweep of the code until a branch instruction is encountered, at which point it attempts to resume its parsing starting from the instruction addressed by the branch. Figure 4.5 presents an example of the parsing of binary code using recursive traversal.

This method allows to overcome the problems presented by foreign bytes interleaved with code, but relies upon the successful identification of instructions affecting the control flow, which implies a more detailed knowledge of the architecture, and of the targets of those instructions. This is especially challenging in the case of indirect branches (cf. 2.2.3), whose target is not immediately deducible from reading the assembly code, requiring an analysis of the preceding instructions to be identified. This analysis can also become exponentially complex if multiple execution paths lead to such a branch. Recursive traversal may also be unable to correctly detect

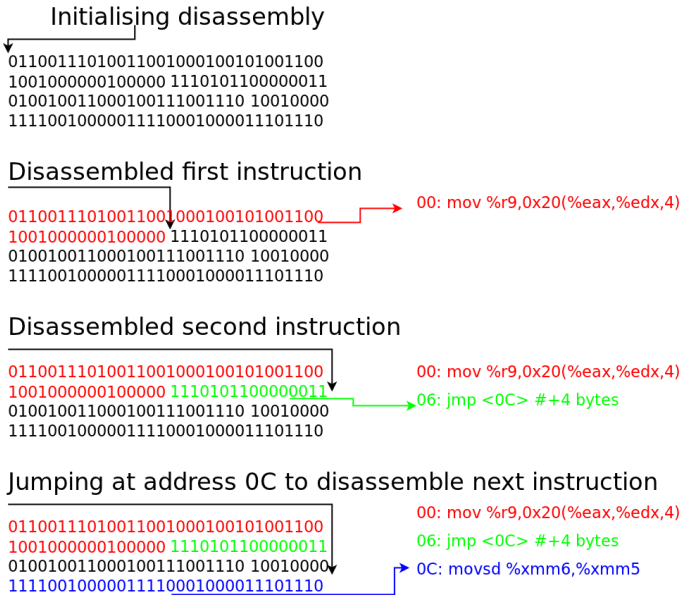

Figure 4.6: Example of a recursive traversal disassembly on Intel 64 code.

overlapping instructions (cf. 4.1.4) without specific handling and miss one of the possible instructions.

This method can also be more slower and complex to implement than a linear sweep, for instance to handle conditional branches, where the control flow can reach two different addresses. Some obfuscation techniques specifically target this weakness by using fake conditional branches, where only one destination is actually used and the other points to code intended to cause further disassembly errors. Other such techniques involve modifying the return address of instructions branching to another subroutine, which are normally expected to return the flow to the instruction following the branch.

4.2.2 Existing disassemblers

Disassemblers are commonly used either as standalone tools to print the disassembled contents of a file, or as part of a more complex application to retrieve the assembly code contained in a binary file before further analysis.

Almost every disassembler use a hard coded recognition of the ISA description, meaning the code of the disassembler itself needs to be updated when adding or updating an architecture.

4.2.2.1 Disassemblers using linear sweep

Most standard disassemblers use a linear sweep algorithm for performance purposes. While this is not always described explicitly in their documentation, it can be easily deduced by observing the disassembly errors returned when attempting to process interleaved foreign bytes. Such disassemblers usually flag parsing errors and do not feature any mechanisms for detecting erroneous instructions.

objdump This standalone disassembler [27] is included in most Linux distributions, handling all architectures supported by this operating system but restricted to an architecture per build. It allows to disassemble executable, relocatable or

shared ELF files (cf. 2.3.3.1) or `a.out` files (cf. 2.3.3.2) and print the assembly code they contain, using an ELF parser to retrieve executable code and its associated virtual address. Instructions are printed to output as soon as they are disassembled. **objdump** also extract labels and prints them alongside the disassembled instructions at the address with which they are associated. It is also able to retrieve and print debug information if present.

As a plain linear disassembler, **objdump** does not feature a mechanism for detecting data mixed with code and simply flags any parsing errors. It does not present any easily usable API for using its results in a more advanced context than printing disassembled code.

XED This disassembler and assembler [44] is developed by Intel for the IA-32/Intel 64 architectures and is available for the Linux, Windows and Mac OS operating systems. It is able to parse binary files for the ELF, WindowsPE and Mach-O formats to retrieve executable code and associated virtual addresses as well as labels. A recent version allows to disassemble files for the Xeon Phi coprocessor architecture but needs to be executed on a system using this architecture. Unlike **objdump**, **XED** offers an API allowing to pilot the disassembly and access its output.

A drawback of **XED** is its restriction to the Intel architectures, which is unlikely to be lifted due to its developing company.

Others standalone disassemblers **ndisasm** [26], **udis86** [96] and **distorm** [49] are disassemblers for the IA-32/Intel 64 architectures. They do not include a parser for the binary format, preventing them from retrieving additional information such as labels and more importantly the boundaries of the code to disassemble, which must be provided externally. **udis86** and **distorm** offer to access the disassembled instructions through an API. **distorm** is developed with a particular emphasis on performance, using a prefix tree to look up the possible matches for a binary stream.

Our tests showed (cf. section 4.5.3) that these disassemblers are not always up to date with regard to the latest evolutions of the architectures, which can prove problematic when analysing code using instructions from those subsets.

llvm disassembler The LLVM framework [71] contains a disassembler similar in functionality to **objdump**. The disassembler is available for the architectures covered by the framework, which include ARM, IA-32, Intel 64 and PowerPC. The code of the decoders used by the disassembler, while relying on data present in tables using a global unified format, is architecture specific.

Debuggers Debuggers usually implement a disassembly feature allowing advanced users to view the assembly code being executed.

The GNU debugger **gdb** [54], offers to disassemble any range of addresses from the executable being debugged. The disassembler may report erroneous instructions if the given starting address is not located at the beginning of an instruction. It is also possible to disassemble whole functions provided the file contains debug information allowing to detect their boundaries.

4.2.2.2 Disassemblers using other methods

The following disassemblers use an improved algorithm, usually recursive traversal, occasionally complemented with linear sweep, in order to extend their coverage. As such, they may trade present lower performance in terms of speed while aiming at a better accuracy.

DynInst This static and dynamic binary instrumenting tool [41, 40] allows to analyse and instrument ELF or XCOFF PE files for the IA-32, Intel 64 and Power ISA architectures through a C++ API. Disassembly is done following a recursive traversal algorithm, the API functions allowing to choose if the scope of disassembly must extend to the whole code or a single function. The results of a disassembly are presented as a CFG, with the smallest unit being a basic block. It is also possible to extend the definitions of structures used to represent CFG elements to add information. A use case is described in [81], which presents methods for identifying functions entry points through pattern recognition and relies on **Dyninst** to perform actual disassembly.

PEBIL PMaC's Efficient Binary Instrumentation Toolkit for Linux [79] is a tool used to instrument ELF executables compiled for the IA-32/Intel 64 architectures. Its first task when instrumenting an executable is attempting to disassemble it, flagging functions whose disassembly failed as ineligible for instrumentation. **PEBIL** uses the file symbol table to identify its functions, then processes each of them using a control-driven (recursive) disassembly. If this fails, **PEBIL** falls back to standard linear disassembly, flagging the function as incompletely disassembled if this also fails. A disassembly failure can be due to an unrecognised opcode or to the discovery of a branch instruction pointing either in the middle of an already disassembled instruction or outside of the current function boundaries while not being a `call` instruction. The latter can lead to discard valid functions, as some compilers, like the Intel compiler **icc**, may add functions invoking one another using such branches, which happens especially when compiling files for use with OpenMP. The former can lead to discard functions where overlapping instructions (cf. 4.1.4) are used.

PLTO This tool [92] allows to instrument and optimise executable files for the IA-32/Intel 64 architectures without recompilation. **PLTO** uses a linear sweep algorithm to disassemble a file. It then correlates the results with a check on the target addresses of branch instructions. If a branch or relocation is found to point to the middle of an already disassembled instruction, that instruction is marked as invalid, and the disassembly is resumed from the target. This behaviour may cause an instruction to be incorrectly flagged as invalid for codes where overlapping instructions are used (cf. 4.1.4), or if foreign bytes interleaved with code led to the erroneous disassembly of a branch instruction. **PLTO** also attempts to analyse jump tables to identify the destination of indirect branches.

UBQT University of Queensland Binary Translation [48] is a tool allowing to convert a binary executable from one architecture to another. The first step of the translation process is the disassembly of the binary into an internal representation. The decoding module uses a recursive traversal algorithm, following the destination

of branch instructions and storing other possible destinations in a queue. The addresses of indirect branches are recovered with a pattern recognition algorithm to identify the location of jump tables. An additional algorithm disassembles the parts in the code that have not been disassembled because they were not reached by any branch, discarding those areas that are found to contain illegal instructions. The boundaries of all disassembled areas are also checked against instructions performing register updates to see if their addresses appear as operands, in order to resolve simple indirect branches.

IDAPro This is a licensed disassembler and decompiler [16], available for most available processors and binary formats. Its professional edition is required for handling 64-bits architectures. IDAPro is available as a standalone application and through a scripting language allowing to develop modules performing tasks on disassembled files. An example of the use of IDAPro is presented in [87] as part of a complete platform able to disassemble and analyse IA-32/Intel 64 executable files.

objconv This object file converter [57] allows to convert library files between binary formats. It is able to parse binary object files and disassemble the code they contain, and also print the disassembled code. **objconv** also perform symbol analysis on the files, and is able to identify some of the foreign bytes interleaved with code as such. It supports IA-32/Intel 64 and Xeon Phi coprocessor architectures.

Our tests showed that the impact of the additional analysis performed by **objconv** can significantly hamper its disassembly speed (cf Section 4.5.1).

Others disassemblers Ďurfina et al. [100] present a decompiler relying on a disassembler. It uses an architecture description language to represent the instructions and their binary coding, and allows to include additional information to the disassembled instructions for use by the decompiler. The decompiler is functional for the Sony PSP console, but does not seem to have been ported over other architectures.

Theiling et al. [97] present a solution for building the CFG of a binary executable by identifying routines in the code then using a bottom-up algorithm to retrieve the branch destinations. It uses the TDL language [65] to represent assembly instruction characteristics, but decoders are hard-coded.

RevGen [45] is a disassembler based on the LLVM [71] framework. It parses binary code and returns the result using the LLVM internal representation, which can then be used for further analysis. **RevGen** is still at an experimental stage and does not handle whole binary files.

4.3 Performing disassembly

Our solution relies on the principles presented in Chapter 3 to generate and maintain the architecture specific code used for parsing instructions. A standard parser for the binary file format is used to extract the binary code to disassemble, the associated virtual addresses, and the function labels. Debug information is retrieved if present using the appropriate parser for its format. Each section potentially containing code is then parsed and a list of structures representing the assembly instructions is built.

Labels retrieved from the binary file are stored in separate structures and linked to the instructions found at the address to which they are associated. The disas-

sembler attempts to identify the instructions targeted by direct branch instructions and stores them in the representation of the instructions.

Since the disassembler is chiefly intended to be used by other tools for further analysis or patching, the main concern is ensuring good performance. Therefore, a linear sweep algorithm has been chosen to perform the disassembly. If a parsing error is detected, the instruction is flagged as bad and the parsing resumes after skipping the smallest instruction length for the given architecture. However, additional metrics allowing to detect potential errors are available and additional procedures can be enabled to handle them.

4.3.1 Disassembly errors

The disassembly process can encounter two types of errors: *parsing failure* and *erroneous disassembly*. A parsing error occurs when the parser is unable to match the binary data with a valid encoding. This case is easily identified since the parser returns an error, allowing to flag the bytes as not corresponding to a valid instruction and skip them. An erroneous disassembly occurs when the parser recognises a valid instruction that is not actually encoded in the file. This error is much harder to detect, as everything appears to be normal from the disassembler perspective.

Disassembly errors of both types can be due to an incomplete or erroneous representation of the ISA, either because of an error when creating the grammar or it being incomplete due to an evolution of the architecture. This can cause valid instructions to fail to be recognised or to be mistakenly disassembled as other instructions. These errors can however be mostly prevented by performing the exhaustive tests of the parser described in 3.4.5 to detect incoherences in the grammar representation of the ISA.

The other main cause of disassembly errors is the presence of foreign bytes interleaved with executable code. As mentioned in 4.1.1, these may not always cause a parsing error, as they can happen to match valid instructions and be disassembled as such. In this case, the disassembler may return a list of instructions without raising any parsing error but that will nonetheless contain erroneous instructions. This may impact further analyses, especially if one such erroneous instruction is a branch, which could cause the CFG to be incorrect.

We will describe here some of the checks and methods available to detect and possibly fix some errors. The possibility remains however that erroneously disassembled instructions remain unnoticed in the final disassembled file.

4.3.1.1 Sanity checks

The disassembly process can perform some preliminary detection of potential erroneous disassembly with minimal impact on its overall performance.

One of them is a simplified metric allowing the detection potential areas of interleaved foreign bytes. A counter is increased for each consecutive parser error and decreased after an instruction is successfully parsed. This enables the detection of instructions surrounded by disassembly errors, which could therefore correspond to foreign bytes that matched with the coding of regular instructions. This allows to establish heuristics for discarding some of these instructions, possibly requesting another pass of the disassembler on a limited number of bytes in order to recover actual code that was erroneously disassembled because of its proximity to the boundaries of the foreign bytes.

Another metric offers information about the flow and can be used for detecting errors as well as for further flow analyses. When scanning the disassembled instructions for associating direct branches to the instructions they branch to, unconditional branches or return from subroutine instructions are detected. All instructions following such branches are then flagged until the destination of a direct branch is met. This will in effect flag all instructions that can not be reached from a direct branch and help identify instructions that are potentially disassembly errors, blocks accessed through an indirect branch, or dead code used for instance for padding. Establishing this metric also needs to take into account the fact that a direct branch instruction could be the result of an erroneous disassembly error, potentially leading instructions to be incorrectly flagged as reachable, since the branch instruction found addressing them does not actually belong to the code. A correlation with the results of the previous check is therefore necessary in order to potentially discard some branch instructions.

4.3.1.2 Handling errors

In addition to checks, actions can be directly taken during disassembly or upon its completion to fix some errors.

Labels overlapped by disassembled instructions This action relies on the labels identified as marking the beginning of a function, either from the binary file or the debug information. If the address associated to such a label is found to be in the middle of a disassembled instruction, this instruction is considered as erroneous. It is then flagged as a parsing error and the parsing is resumed from the address of the label. This allows to handle erroneous disassembly errors during parsing, but is only effective near the beginning of functions. Padding is however often added between functions for alignment purposes, so these locations have a higher risk of disassembly errors. This method could be extended to include direct branches pointing to the middle of instructions, however those could also be the result of overlapping instructions as described in 4.1.4. This needs therefore to be correlated with the confidence in the validity of the branch instruction and the capacity of the bytes addressed by the branch to be disassembled as a valid instruction. The results of this correlation can reveal an erroneous disassembly of either the instruction or the branch, or simply a case of overlapping instructions. The latter case can be handled by specific flags in the representation of the branch. A similar method consists in detecting labels identifying variables in the binary file or the debug information but pointing to a section supposed to be executable code. However, not all binary formats define such label types and debug information may not be present.

Other methods for detecting or fixing errors are available through further analysis upon completion of disassembly. Since they impose additional processing, they can negatively impact performance, imposing a trade-off between speed and precision.

Analysing error ratios One such method involves the overall count of parsing errors encountered during disassembly. The ratio of errors over the total number of instructions exceeding a given threshold means either that the file being disassembled uses too many unrecognised instructions, which can be caused by an obsolete parser or by using the parser for the wrong architecture, or that the section being disassembled does not actually contain executable code. Different actions can then

be triggered depending on the threshold reached, including a critical error for the highest ratios (above 1%). A more fine-grained approach involves keeping track of the error ratio over basic blocks or functions, allowing to single out those containing higher error ratios. Those blocks can therefore be either discarded from the final disassembly result if they are also found to be unreachable from direct branches or simply flagged as unreliable, thus complementing the check described in 4.3.1.1.

Detecting suspicious instructions While erroneous disassembly errors are not directly identifiable, those due to interleaved foreign bytes can be identified through additional heuristics. These heuristics include the detection of instructions using operands significantly different from those used by the surrounding instructions, such as a register unused by any other instruction in the current block or function. Another such heuristic involves the detection of groups of nearly identical instructions following each other, as the result of disassembling a series of foreign bytes with a uniform value can result in such groups. However, a valid code could also legitimately contain such series of instructions, so their type needs to be taken into account to identify if this make sense. Pattern matching can be used in this case.

Memory accesses to disassembled addresses Another, more specific, method involves the detection of instructions not dedicated to control flow but reading from memory addresses identified as belonging to the range of addresses being disassembled, therefore revealing the presence of data bytes. Further analyses are however necessary to identify the precise boundaries of the area containing those bytes. Instructions writing to such addresses can also reveal the use of self-rewriting code. This method needs to be correlated with the previous checks ascertaining the validity of instructions to ensure the detected instructions are not themselves resulting from an erroneous disassembly.

If either of the methods above allows to identify a block of instructions as not actually containing valid executable code, the disassembler can perform a second disassembly pass near its boundaries in order to retrieve valid instructions whose proximity to this block could have prevented to be correctly parsed. Invalid blocks can be marked as such and removed from the list of disassembled instructions for all further analyses.

4.3.2 Disassembler output

The output of the disassembler is a list of structures describing the instructions.

These structures contain every information that could be retrieved from their encoding correlated with additional characteristics encoded in the grammar, as well as generic information retrieved during the disassembly process. This information includes the following elements:

- Instruction mnemonic
- Instruction operands, including their type, size and role (read, write)
  - For architectures allowing this, the update status of the components of a memory address operand
- Instruction binary coding

- Instruction virtual address
- All information present in the grammar and described in 3.3.1.4

These structures also contain optional fields that can be used either to store additional information which can be loaded during the post parsing phase, or to expand the description of the instruction. This can be useful to represent instructions from architectures whose assembly language contains specific information. This is for instance the case for the Xeon Phi coprocessor, where every operand can be applied a mask register or conversion modifier that have no equivalent in other architectures.

4.4 MADRAS disassembler

We implemented the principles presented in Section 4.3 into the MADRAS tool (**M**ulti-**A**rchitecture **D**isassembler **R**ewriter and **As**sembler). MADRAS allows to disassemble an ELF binary file and return structures representing the assembly instructions and containing every available information from the grammar. It is also able to identify the destinations of direct branch instructions and retrieve additional data from the ELF file, such as function labels, and to parse the debug information in DWARF format if present to complement the description of instructions, for instance with their corresponding line numbers in the source code.

MADRAS relies on MINJAG to generate the architecture specific code of the parser. It is available as a standalone disassembler and fully integrated into the MAQAO framework [39].

MADRAS is kept up to date with the evolutions of the Intel 64 architecture and supports all extensions, up to and including AVX2, as defined in the latest version of the documentation (as of the writing of this dissertation). This is done easily by updating the instruction list extracted from the Intel documentation that MINJAG uses as a base to generate files. MADRAS also supports the Intel Xeon Phi coprocessor architecture. The implementation of disassembly for the ARM architecture is in the process of being tested as of the writing of this dissertation.

MADRAS is able to support macro instructions representing multiple instructions thanks to the shift/reduce states allowed by MINJAG in the generated parser.

4.4.1 Inner workings

The disassembler relies on an ELF parser to identify the sections containing binary code and retrieve the corresponding bytes. It returns structures defined in the base MAQAO framework representing the instructions. Figure 4.7 presents an overview of the disassembler operating mode.

4.4.1.1 Disassembler initialisation

At the beginning of a disassembly session, the disassembler retrieves the architecture of the file from the ELF format, then invokes the corresponding binary parser. An error is raised if the file is not defined for one of the supported architectures.

The labels extracted from the file are ordered by increasing associated address.

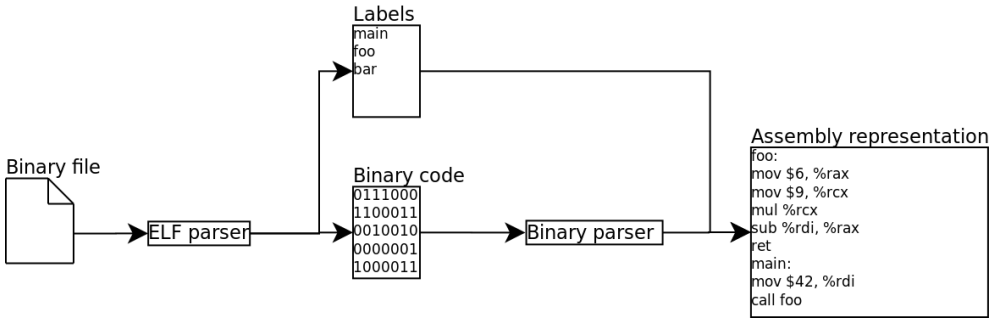

Figure 4.7: Description of the handling of binary files by the MADRAS disassembler.

4.4.1.2 Disassembly process

The disassembler invokes the FSA parser (cf. 3.3.3.3) on the bytes of each section of the ELF file containing executable code. The parser returns each parsed word as a structure representing the instruction and containing its mnemonic and operands. The operations performed on a parsed instruction are detailed in algorithm 9.

Algorithm 9 Parsing an instruction

Require: `insn`: A structure containing information filled by the parser.

```
1: insn.address ← Parser_GetCurrentAddress()
2: insn.encoding ← Parser_GetCurrentCoding()
3: AssociateLabel(insn)
4: if ∃ label / insn.address < label.address < insn.end_address then
5:    insn.end_address ← label.address
6:    insn.badly_disassembled ← true
7:    Parser_SetCurrentAddress(label.address)
8: end if
9: if Parser_Error() == false then
10:   if IsBranch(insn) then
11:      AddToTable(branches, insn)
12:   end if
13:   error_counter ← error_counter −1
14: else
15:   error_counter ← error_counter +1
16:   insn.badly_disassembled ← true
17: end if
18: if error_counter > 0 then
19:   insn.dubious ← true
20: end if
21: AddToList(insn_list, insn)
```

Each instruction is linked to the eligible label associated to the closest preceding address. A label is eligible for association if it obeys the following rules:

- It does not begin with a '.' character
- It is of type function if more than one label is associated to the same address.
  - A label is of type function if identified as such by the ELF parser or in the debug information if present

– If the debug information is present, a label not identified as a function by it is never eligible

If more than one label is eligible for a given address using the rules above, string comparison is used to break the tie in a deterministic manner.

4.4.1.3 Disassembler completion

When all sections have been disassembled, the instruction list is scanned once more to identify targets of branches and potentially erroneous instructions. This last pass is described in algorithm 10.

Algorithm 10 Second pass on disassembled instructions

Require: `insn_list`: List of disassembled instructions
Require: `branches`: Table of branch instructions indexed on referenced addresses

```
1: for each insn in insn_list do
2:   unreachable ← false
3:   for each branch in branches / branch.addressed == insn.address do
4:     branch.destination ← insn
5:   end for
6:   unreachable ← false
7:   if ∃ label / label.address == insn.address then
8:     unreachable ← false
9:   end if
10:  if unreachable == true then
11:    insn.unreachable ← true{insn is unreachable by a direct branch}
12:  end if
13:  if IsUnconditionalBranch(insn) then
14:    unreachable← true
15:  end if
16: end for
```

4.4.2 Parallel disassembly

Madras can also use multiple threads to disassemble files. In this mode, the binary code to disassemble is split, each part processed by a separate thread, and the resulting instruction lists are merged into a single one after all threads are finished.

The splitting of the file is performed following the identified function labels. When this is not possible, for instance if the file does not contain labels, the splitting is done arbitrarily, and the merging may require disassembling a second time the bytes at the boundaries between two parts to recover valid instructions that were split between two threads. The self-repairing properties of the Intel architectures ensure that a minimal number of instructions will have been affected this way. This is not necessary for architectures with fixed instruction length as it is then possible to split the code so as to ensure the parts length is a multiple of the instruction size.

The implementation of this mode is currently at an experimental stage.

4.4.3 Use in MAQAO

MADRAS is used as an entry point by the performance analysis tool MAQAO [23] to process binary executables. The structures generated by MADRAS to represent the instructions in a binary file are used by MAQAO to identify functions, loops and basic blocks, and build its Control Flow and Data Dependency Graphs. This is based on the resolution of direct branches by MADRAS and the identification of subroutine calls as well as the labels retrieved from the ELF file.

In MAQAO, the post-parsing actions of the MADRAS disassembler are used to fill additional information concerning instructions latencies and dispatches among processor ports, allowing MAQAO to perform static performance analyses at the loop level. The code quality analysis plug-in for MAQAO uses this additional information to build estimations of the cycles needed for the execution of loops or functions. In particular, it is able to identify the vectorisation ratio of loops and statically estimate their execution time in cycles. MADRAS can also access the debug information in the file if it is present, allowing MAQAO to reference the original code in its results.

MAQAO also uses the patching functionalities of MADRAS, which will be covered in the next chapter.

4.5 Disassembler performance

The performance of a disassembler can be judged on two criteria, speed and accuracy.

Disassembly speed is calculated from the time necessary to parse a whole file and return its contents, either as structures or in printed form. While this time obviously depends on the size of the disassembled file, the relation may vary depending on the content of the file, especially if the sections containing binary code are not the largest ones in the file. For instance, the size of a binary file may be due to a comparatively large data or debug section, neither of which has a significant impact on the disassembly process. Conversely, an important number of labels in a file, reflected by a large section containing them, will affect the parsing speed of the binary file but not its disassembly. In order to keep the focus on the disassembly itself, the metric used for characterising the disassembly speed will be expressed in number of disassembled instructions per second.

Accuracy measures the capacity of the disassembler to correctly parse the instructions of the architectures it supports, and to successfully disassemble whole files. The correct parsing of all instructions of an architecture can be tested by creating test files such as those described in 3.4.5, then attempting to disassemble them and compare with their source. As noted in 4.3.1, disassembling a compiled file with 100% completion is not always possible because of interleaved foreign bytes. We will therefore use the ratio of parsing errors over the total number of instructions for characterising the disassembler accuracy over regular files.

We compared MADRAS with other disassemblers among those presented in 4.2.2. All of those disassemblers use a hard coded implementation of the architecture specific parsing functions, which allows to optimise the handling of specific cases and thus to offer better performance than generated code. Since their behaviour and output vary, we had to adapt the normal operating mode of MADRAS accordingly in order to establish a valid comparison. We used for our tests some of the largest SPEC 2001 and 2006 benchmarks [30] as well as some of the exhaustive test files generated from the instruction list for the Intel 64 and Xeon Phi coprocessor architectures.

4.5.1 Testing context

We compared MADRAS with **objdump**, **XED**, **ndisasm**, **udis86** and **distorm**.

Since the MADRAS disassembler does not perform flow analysis in order to allow the tools using its output to choose the extent of additional analyses to perform, we excluded from the comparison the disassemblers for whom flow analysis is tied to the disassembly process, such as **DynInst**.

The decoder generated with GDSL was excluded from the tests since it failed for the files used for this benchmark, either crashing or never stopping. Separate tests using the test binary provided by GDSL showed that MADRAS disassembly speed was 60% higher than the GDSL decoder for this sample file.

objconv was also excluded as it complements its disassembly with analyses aimed at detecting interleaved foreign bytes, thus offering a better accuracy at the expense of speed. This additional processing, especially the handling of labels in the file, significantly slows down **objconv** for the large files which were used for the test. While its disassembly speed reached between 150% and 200% of the speed of MADRAS for the test files containing no labels nor foreign bytes, it dropped to 70% for the smallest SPEC files, and to less than 20% for files larger than 1Mb.

The behaviour of the disassemblers is summarised in table 4.1.

Disassembler	ELF parsing	Structures	Printing	Architectures
objdump	Yes	No	Yes	Intel 64, Xeon Phi
ndisasm	No	No	Yes	Intel 64
udis86	No	Yes	No	Intel 64
distorm	No	Yes	No	Intel 64
xed	Yes	No	Yes	Intel 64, Xeon Phi

Table 4.1: Actions performed by the disassemblers used for comparison with MADRAS. *ELF Parsing* means that the disassembler parses the ELF file to retrieve the boundaries of code to disassemble. *Structures* means that the disassembler builds structures representing instructions. *Printing* means that the version of the disassembler used for the test prints the instruction list.

Those among the tested disassemblers offering an accessible API were tweaked to be closest to the normal operation mode of MADRAS, which is disassembly without printing of the instructions; this is taken into account in table 4.1. In order to emulate the different behaviours of the other disassemblers to allow establishing a relevant comparison, the following modes were implemented in the MADRAS disassembler:

- ***Standard***: Allocating the structures representing instructions, printing them then freeing them. This is the standard behaviour of MADRAS when used as a simple disassembler.

- ***Print only***: Printing the instructions without allocating the structures representing them. This mode is only used for comparison, since it is unsuited for use by an analysis tool and prevents parallel disassembly.

- ***Mute***: Allocating the structures representing instructions and freeing them. This is the normal mode of operation for MADRAS when used as the entry point of another tool.

- ***Raw***: Disabling the parsing of the ELF file to retrieve the boundaries of the sections of code, which must be fed manually to the disassembler. This mode is to be combined with one of the three previous ones.

Table 4.2 presents the characteristics of the files used for those tests.

File	Architecture	File size (Mb)	Code size (Mb)	Description
Small	Intel 64	0,96	0,96	Test file
fma3d	Intel 64	3,78	1,75	SPEC2001
calculix	Intel 64	5	2,31	SPEC2006
gcc	Intel 64	9,02	2,56	SPEC2006
dealII	Intel 64	60,94	2,83	SPEC2006
Xalan	Intel 64	130,64	3,46	SPEC2006
tonto	Intel 64	33,27	5,81	SPEC2006
wrf	Intel 64	19,52	6,83	SPEC2006
gamess	Intel 64	18,2	10,55	SPEC2006
Large 1	Intel 64	11,95	11,94	Test file
Large 2	Intel 64	23,22	23,22	Test file
equake	XeonPhi	0,12	0,05	SPEC2001
art	XeonPhi	0,21	0,12	SPEC2001
ammp	XeonPhi	0,84	0,44	SPEC2001
swim	XeonPhi	0,96	0,57	SPEC2001
wupwise	XeonPhi	0,96	0,66	SPEC2001
mgrid	XeonPhi	0,95	0,68	SPEC2001
applu	XeonPhi	1,03	0,71	SPEC2001
apsi	XeonPhi	2,61	1,72	SPEC2001
galgel	XeonPhi	2,84	2,08	SPEC2001
fma3d	XeonPhi	4,62	2,35	SPEC2001

Table 4.2: Files used to test disassemblers performance. The occasionally significant differences between file size and code size are due to the presence of labels or debug information. The files described as *test files* contain sequences of instructions with various exhaustive combination of mnemonics and operands and contain no labels nor debug information.

With the exception of the comparison with **XED** for Xeon Phi (cf. 4.5.2.2), all tests were performed on a computer using a Sandy Bridge with 8 processors at 2.6 GHz and 20 Mb cache, running under RedHat. The Linux command `time` was used to time the disassembly executions. Each test was performed 30 times, with the resulting timing calculated from an average of the results.

4.5.2 Disassembly speed

All figures in this section compare disassembly speeds, so higher values are better. Files are ordered by increasing size of their respective code sections.

4.5.2.1 Comparing Madras operating modes

Figures 4.8 and 4.9 present a comparison of the disassembly speeds of the various Madras operating modes on Intel 64 and Xeon Phi coprocessor executables respectively. As expected, the modes printing instructions are slower due to I/O

accesses. The results show that allocating and deallocating the representations of instructions instead of simply printing them slows down MADRAS by 20% to 30% . The lowest performance is obtained with large files (more than 100 Mb) containing an important number of labels, which add a significant overhead to the disassembly process. This is evidenced by the comparison with the modes where the ELF file is not parsed, which can be up to 40% faster when dealing with large files (100 Mb).

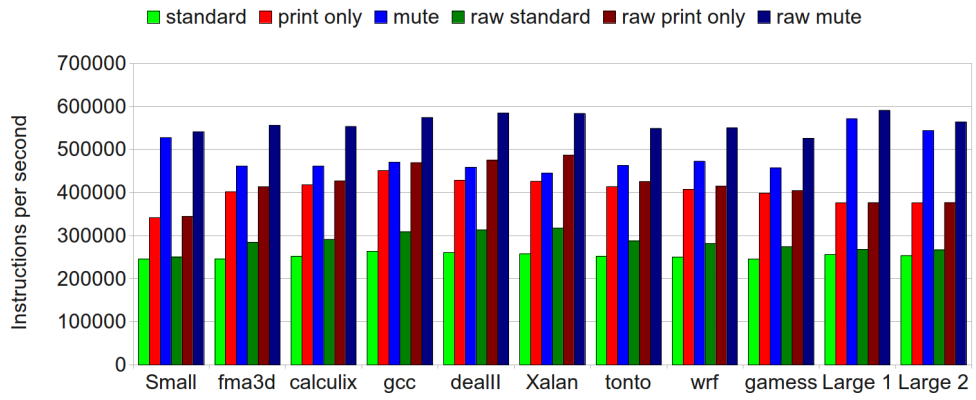

Figure 4.8: Performances of the MADRAS disassembler on Intel 64 files, for various operating modes. In *print only*, the instructions are printed directly without allocating structures, while for *mute*, the structures are allocated and destroyed but nothing is printed. Finally, in *standard*, the structures are allocated, printed and freed. *Raw* modes skip the parsing of the ELF file.

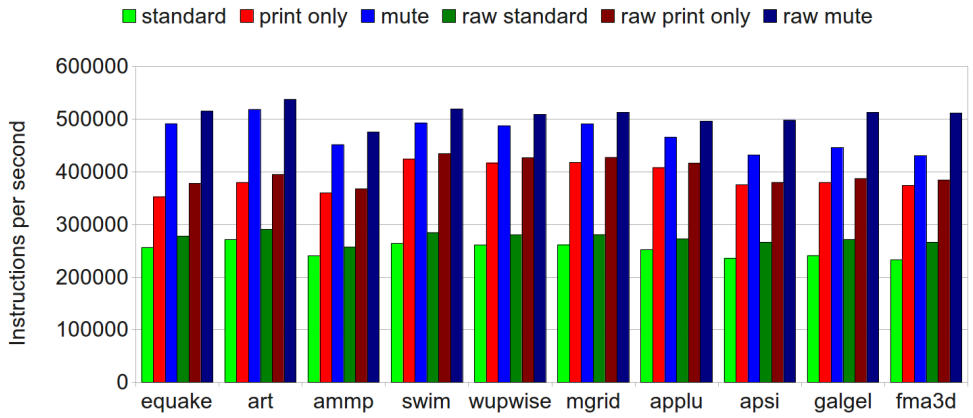

Figure 4.9: Performances of the MADRAS disassembler on Xeon Phi coprocessor files, for various operating modes. In *print only*, the instructions are printed directly without allocating structures, while for *mute*, the structures are allocated and destroyed but nothing is printed. Finally, in *standard*, the structures are allocated, printed and freed. *Raw* modes skip the parsing of the ELF file.

4.5.2.2 Comparison in print only mode

We compared the MADRAS disassembler in print only mode with **objdump** and **XED** on the Intel 64 and Xeon Phi coprocessor architectures.

The results of the comparison with **objdump** and **XED** for Intel 64 files are presented in figure 4.10. These tests show that the MADRAS disassembly speed is comparable to the speed of **objdump**, reaching between 90% and 110% of its speed. **XED**, being maintained by the processor manufacturer, is on average 20% faster than both of them, reaching its highest performance for files containing no labels, which could also mean that its ELF parser is more optimised.

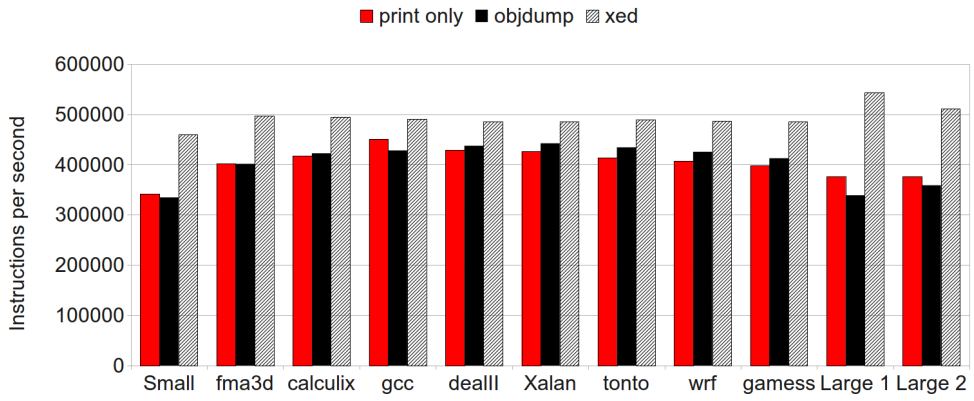

Figure 4.10: Performances of the MADRAS disassembler on Intel 64 files, compared with **objdump** and **XED**. The MADRAS disassembler directly prints instructions without allocating structures (*print only* mode).

The results of the comparison between MADRAS and **objdump** for Xeon Phi coprocessor files are presented in figure 4.11. A different compiled version of **objdump** was used for those tests, while the MADRAS disassembler was the same executable.These tests show a similar performance as the one observed for Intel 64.

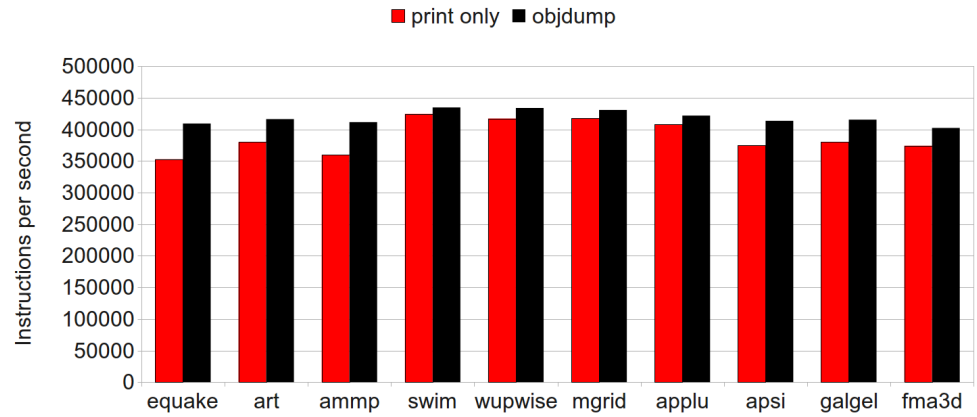

Figure 4.11: Performances of the MADRAS disassembler on Xeon Phi coprocessor files, compared with **objdump**. The MADRAS disassembler directly prints instructions without allocating structures (*print only* mode).

The results of the comparison between MADRAS and **XED** for Xeon Phi are presented in figure 4.12. As **XED** for Xeon Phi is only available on systems using this architecture, these tests were performed on such a system, using a version of the MADRAS disassembler compiled for it.

The performance of both tools on this system is different from Intel 64 as the cores are slower. Another reason is that a linear parser, whose code can not be easily vectorised, does not take full advantage of the Xeon Phi coprocessor architecture, especially when executed sequentially.

These tests show a better relative performance for MADRAS than observed on Intel 64, with MADRAS being on average 10% faster than **XED**.

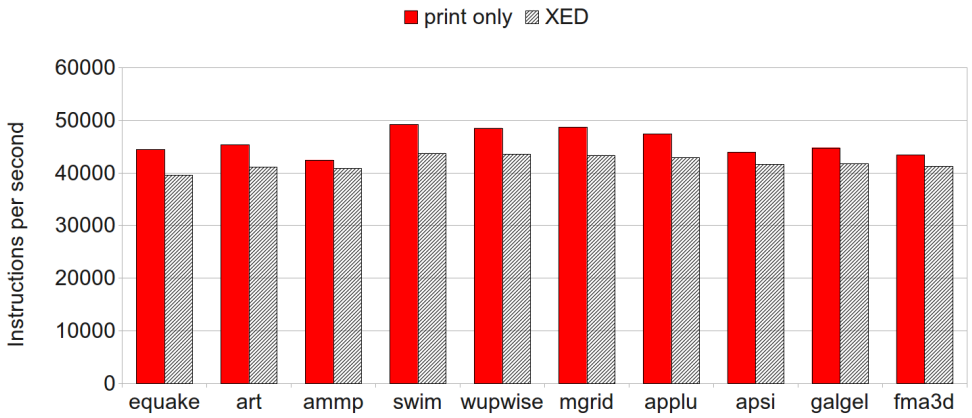

Figure 4.12: Performances of the MADRAS disassembler on Xeon Phi coprocessor files, compared with **XED**. The MADRAS disassembler directly prints instructions without allocating structures (*print only* mode). Unlike the others, those tests were performed on a Xeon Phi system, causing a lower speed for both tools.

4.5.2.3 Comparisons in raw mode

Figure 4.13 presents a comparison between the MADRAS disassembler in raw print only mode and **ndisasm**. Figure 4.14 presents a comparison between the MADRAS disassembler in raw mute mode and **udis86** and **distorm**. These tests were only performed for Intel 64 as these disassemblers does not support the Xeon Phi coprocessor architecture. The results presented show that the MADRAS disassembly speed is comparable to the speed of other hard-coded disassemblers. Only **distorm**, which is optimised for high performance, is significantly faster. However, some of the files caused it to crash during disassembly. This also occurred with **udis86** for different files.

4.5.2.4 Parallel disassembly

Figures 4.15 and 4.16 present the evolution of the disassembly speed of MADRAS in multi-threaded mode for the Intel 64 and Xeon Phi architectures respectively. The mute mode was used for those tests, as it is the standard behaviour of MADRAS and focuses on the stage of the disassembly process which benefits from multi-threading. The results show that the disassembly process presents acceptable scalability for up to 4 threads, but with moderate efficiency. As this mode is still in the experimental stage, this average performance can be due to flaws in its implementation.

Figure 4.17 presents a comparison of the MADRAS disassembly speed in 4 threaded mode with **udis86** and **distorm**. We observe that the multi-threaded

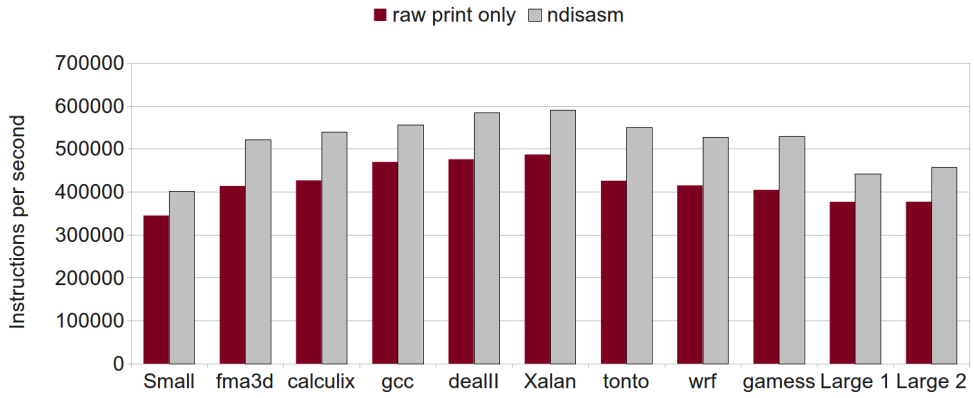

Figure 4.13: Performances of the MADRAS disassembler on Intel 64 files, compared with **ndisasm**. The MADRAS disassembler directly prints instructions without allocating structures and does not parse the ELF file (*raw print only* mode).

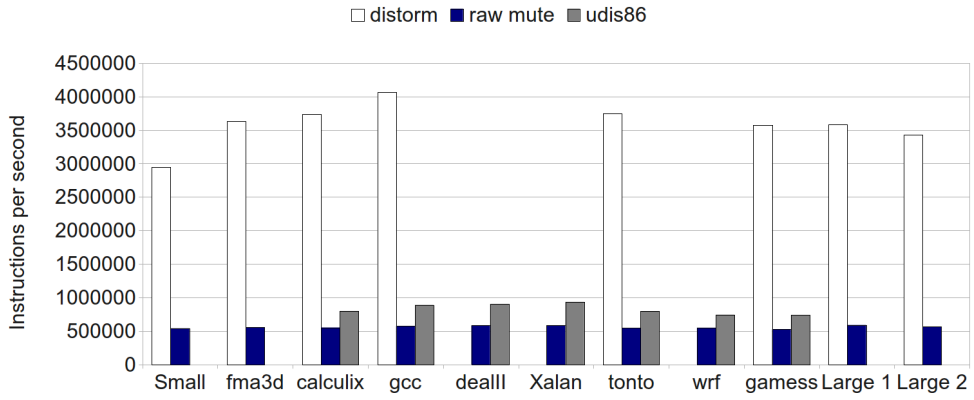

Figure 4.14: Performances of the MADRAS disassembler on Intel 64 files, compared with **udis86** and **distorm**. The MADRAS disassembler allocates structures representing instructions without printing them and does not parse the ELF file (*raw mute* mode). Missing values indicate that the tool crashed for all tests on this file.

mode, even in its experimental stage, allows MADRAS to overcome its handicap with **udis86**. These results justify further works on improving this mode.

4.5.3 Accuracy

The accuracy performance of the disassembler can be evaluated by matching its result with an assembly reference or by comparing its errors with other disassemblers.

The MADRAS disassembler is routinely tested for accuracy on the test files containing exhaustive combinations of mnemonics and operands for the supported architectures, using their sources as reference. These results show that the ratio of disassembly errors, including parsing errors and erroneously disassembled instructions, amounts to 0.099% for the Intel 64 architecture and 0.024% for the Xeon Phi architecture. Most of the errors are due to dubious cases of homonyms, such as instructions accepting different register types as operands with the same encoding.

Figure 4.18 presents a comparison of the disassembly errors between MADRAS

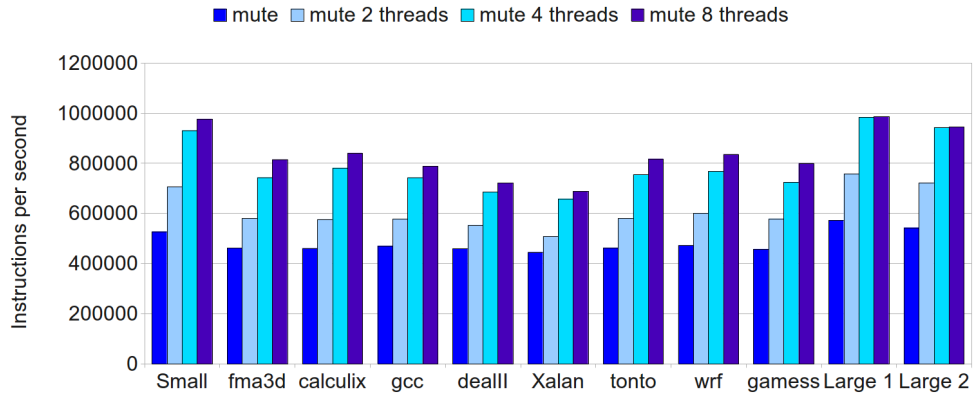

Figure 4.15: Performances of the MADRAS disassembler on Intel 64 files, for various number of threads. For this test, the *mute* mode was used for MADRAS.

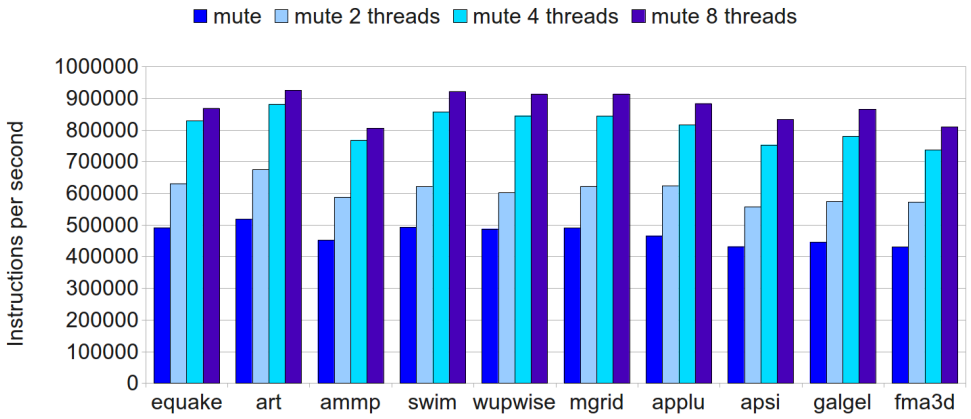

Figure 4.16: Performances of the MADRAS disassembler on Xeon Phi coprocessor files, for various number of threads. For this test, the *mute* mode was used for MADRAS.

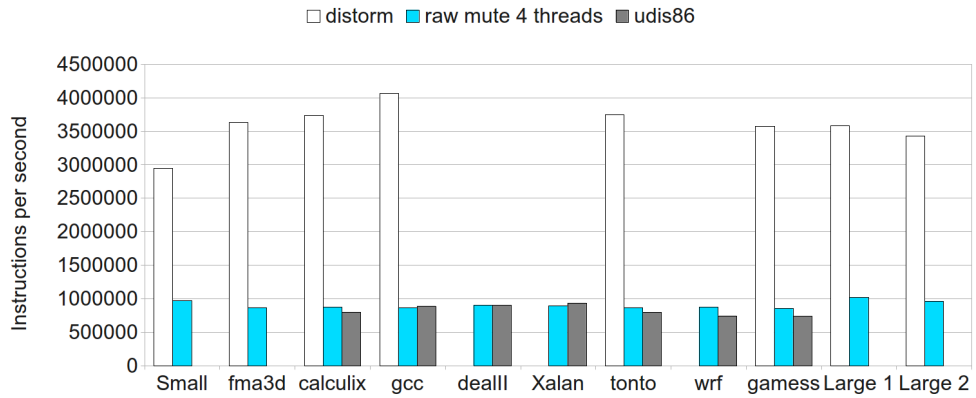

Figure 4.17: Performances of the MADRAS disassembler in 4 threaded mode on Intel 64 files, compared with **udis86** and **distorm**. For this test, the *raw mute* mode was used for MADRAS.

and other disassemblers. The errors detected here are exclusively parsing errors, as the detection of erroneous disassembly is much harder to perform without a reference. We had to exclude the test files from this comparison as they contained instructions from the latest instructions sets that were not supported by the other disassemblers, especially **distorm** and **ndisasm**, leading to comparatively high error ratios (3 to 17%) on those files.

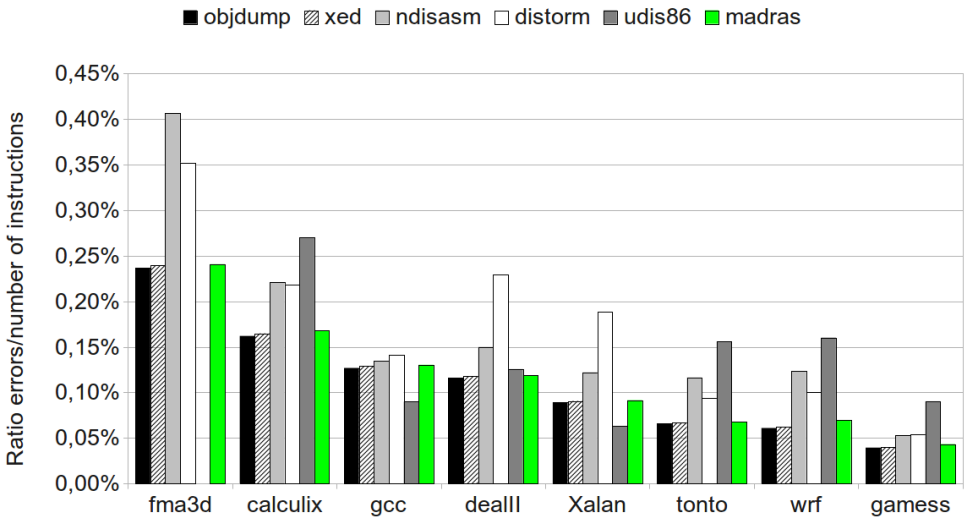

Figure 4.18: Comparison of error ratios between other disassemblers and Madras for Intel 64 executables. In this test lower values represent a better accuracy.

objdump, **XED** and Madras return less disassembly errors than the other disassemblers. Most of those errors are due to interleaved foreign bytes that could not be identified as such. **objdump** comes slightly ahead since it identifies lone opcode prefixes as such instead of reporting a disassembly error. Such prefixes have no significance and are ignored by the processor decoder anyway.

4.6 Conclusion

We presented here an application of the binary parser generated from the principles presented in the previous chapter under the form of a multi architecture disassembler. This disassembler is intended to be used by analysis applications and as such allows to customise its output to provide additional information about the disassembled instructions. The principles involved also offer means to detect parsing errors and erroneously disassembled instructions that can occur when processing compiled executables. The disassembler was implemented as Madras and tests showed its performance in terms of speed and accuracy to be comparable, if not better, than those of other disassemblers using a hard-coded representation of the architecture. Madras currently supports the Intel 64 and Intel Xeon Phi coprocessor architectures, with the ARM architecture being in the testing stages.

In the next chapter, we will present an application of the instruction list returned by the disassembler as a binary patcher allowing analysis tools to complement their results with instrumentation.

Chapter 5

Patching executables

Instrumentation consists in the retrieval of information from an application, such as timing or data profiling, during its execution. It is a crucial step in an analysis process, as it allows to retrieve data available only at runtime or whose deduction from a static analysis would require an exceedingly complex process. It presents however multiple challenges as the overall behaviour of the executable must not be impacted by the instrumentation.

A common mean to achieve this is through patching of the executable, which consists in its modification without recompilation. Instrumentation is then performed by patching the file to insert function calls or assembly code snippets allowing to retrieve the desired information. In addition, patching allows to perform other operations, such as modifying instructions to observe their impact on the overall performance of the application.

We will focus here on the patching methods involving a rewriting of the binary file. This approach allows to reduce the impact of the instrumentation stage in the analysis process by performing it only once and to impose no other constraint on the target system than those required for running the original executable. Its main challenge stems from the fact that, unlike object files, executables are not intended to be modified and could even contain specific countermeasures against this.

The process of patching a binary file can be broken down into the following steps:

1. Retrieving the code present in the file
2. Alter the assembly code as needed by the patching operation
3. Rebuild the file as a valid executable

Step 1 was covered in chapter 4. In this chapter, we will focus on the challenges and solutions to steps 2 and 3.

For the remainder of this chapter, the term "patch" will be used for any operation modifying a binary executable, while "instrumentation" is reserved for the specific modifications aiming at retrieving information from the executable at runtime, like data profiling, memory tracing, or timing. Thus, instrumenting a code can be done by patching it, but not all patching operations aim at instrumentation.

We will first describe the challenges of binary patching in Section 5.1. We will then present common solutions to those challenges as well as alternatives to binary rewriting, and some existing implementations in Section 5.2. Our solutions will be described in Section 5.3 and their implementation in Section 5.4.

5.1 Challenges of patching

The primary requirement of a patching operation is that the behaviour of the patched file must be identical to the original, except for some specific applications. For instance the output of the patched executable is expected to be identical to the

original for the same input. Most importantly, the patched program must not crash when run with its normal parameters.

This means that the control flow and data must be kept intact, or as close as possible to the original, and that the patching operation must not cause the appearance of forbidden or undefined operations. Another important requirement for patching is to avoid inducing important overheads in order to avoid excessive slowdowns of the patched files.

A prerequisite for correct patching is the successful disassembly of the file, or at least the parts concerned with patching. Disassembly errors can be pinpointed using the methods described in 4.3.1, and patch operations impacting code flagged as erroneous or dubious can be discarded to prevent accidental modification of interleaved foreign bytes. For the remainder of this chapter, we will be assuming that a patching operation never attempts to modify code containing disassembly errors.

5.1.1 Preservation of the control flow

When running an executable file, the processor uses the instruction pointer to identify the address of the instruction to execute. The bytes present at this address are decoded and the length of the decoded instruction is added to the instruction pointer. If the instruction is a branch instruction, its execution will update the instruction pointer with the address pointed to by the branch.

Erroneous addresses stored in the instruction pointer will cause the program execution to fail. If the address is located outside the segment allocated to the program for execution, this will cause a segmentation fault. If the address does not contain binary code that can be decoded into a valid instruction, an unknown opcode exception will be thrown by the processor; this happens for instance if the instruction pointer references an address halfway into the coding of a valid instruction. In this last case, if the code pointed to by the pointer happens to be successfully decoded into an instruction, the processor will execute at least one erroneous instruction, and may also lose synchronisation with the actual code and eventually encounter a binary code it is unable to decode. If the instruction pointer contains a valid address, but the code at this address is not meant to be executed at this point, the program behaviour will differ from the original, ranging from returning incorrect results to throwing a segmentation fault, for instance if the code attempts to access a memory address contained in a register set to a null value.

5.1.1.1 Impact of patching on branches

Patching involves the insertion and possibly deletion of instructions inside the flow of the executable, thus modifying its total length and the offsets between instructions. This can also happen when modifying an existing instruction in architectures defining different instruction lengths, as the modified instruction could be encoded with a different size. Since branch instructions rely on relative and absolute addresses to reference their destination in binary code, such changes can impact the referenced addresses. Figure 5.1 displays an example of branch instructions assigning wrong values to the instruction pointer because of an inserted instruction.

It is therefore necessary to ensure that branches in a patched file still point to valid instructions, and that those instructions are coherent with regard to the original control flow of the program. This concerns direct as well as indirect branches (cf. Section 2.2.3), and requires the identification of their destination.

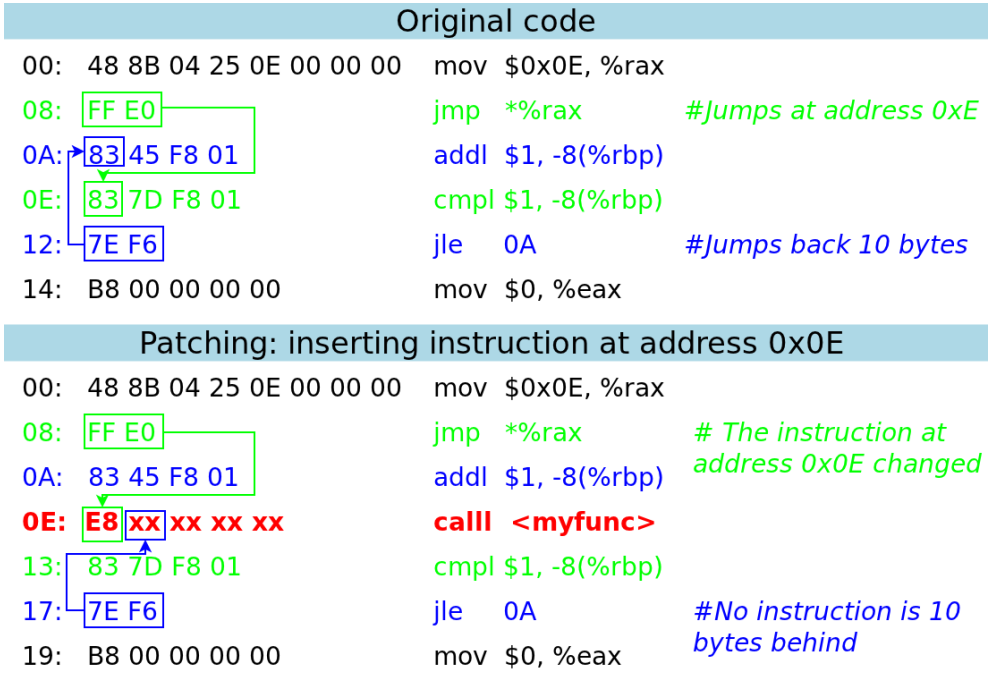

Figure 5.1: Example of cases (using Intel 64) where the insertion of a single instruction causes branch instructions to point to incorrect addresses. The branch at address 0x08 in the original code will address the inserted instruction in the patched code, which can cause the code execution to be altered. The branch at address 0x12 in the original code presents a more serious case, as it will address the second byte of the inserted instruction in the patched code, which may either cause an undefined opcode exception or the execution of an erroneous instruction.

5.1.1.2 Detection of branch destinations

While the destination of a direct branch can be easily identified statically, it is not always possible for indirect branches or may require advanced analysis. In particular, while direct branches have one possible destination, indirect branches are often used for addressing varying destinations depending on runtime data. It may be therefore necessary to identify all possible values that the branch operand could take to be able to update them.

In some cases, the value stored in the operand used by an indirect branch can be the result of a series of calculations, some of them possibly conditional. Identifying those values may be complex statically, since the number of possible paths can increase exponentially when attempting to trace the origin of a value.

Indirect branches can also use a switch table, where the branch operand references a cell in an array containing the address to branch to. This array is usually stored in the data section, and as such its boundaries may not be obvious. For instance, the cell index could be the result of an extensive computation, making even the base of the array hard to identify. The array can also be contiguous with other such arrays, making the analysis of its contents, for instance through pattern matching, inconclusive for the purpose of identifying its boundaries.

Finally, addresses can appear as immediate values in assembly code. This is the case for instance when passing a function address as parameter to another function. It is impossible to distinguish such an address from another immediate without some

heuristics, which need to take into account the fact that numerical values used as standard immediate operands may happen to match the address of a function in the executable, thus potentially leading to false detections of such cases.

5.1.2 Preservation of the data context

The execution of a program also depends on its runtime data environment: values stored in registers, on the stack, or in other memory locations. Code added to a file through a patching operation is liable to change this environment. Since this code may not be known during the patching operation, for instance if located in a dynamic library, it is not possible to predict which elements of the environment it will impact.

5.1.2.1 Saving register values

The conventions of the Architecture Binary Interface (ABI) specify which registers, also called scratch registers, must be saved before invoking a function. These registers can therefore be overwritten by a function without saving them beforehand, as the invoking function is expected to have done this if it was using them. However, when patching a code, a function call may be inserted anywhere, making the additional insertion of instructions for saving and restoring these registers necessary. In some cases, the process of invoking an external function itself may also overwrite some registers. Finally, it has been observed for executables built from a single file that the compiler may not have obeyed the ABI conventions when invoking internal functions for optimisation purposes.

Architectures also define specific registers storing flags containing the results of comparisons, which are used by conditional instructions. These flags may be modified by the inserted code, which could change the behaviour of further conditional instructions in the original code.

The patching operation must therefore include the insertion of instructions allowing to save and restore the contents of registers. This may however lead to significant overheads.

5.1.2.2 Saving the stack

The stack is an area of memory used for storing local data and passing parameters to invoked functions, depending on the architecture and the level of compilation optimisation. Its top level is usually identified by the stack pointer, but the code of a function may actually access any position inside the stack regardless of the address identified by the stack pointer. A detailed analysis of the code is necessary to identify the parts of the stack that are used, as the code added by a patching operation may access the stack and possibly overwrite values present on it.

For instance, depending on the ABI, an inserted function call may modify the stack, first when saving the calling context, then when passing parameters to the function, and finally by the called function itself. If the patched code was using data stored on the stack, the inserted code may overwrite it.

The stack must therefore be preserved before the inserted code is executed, either by saving its value or moving the stack pointer. [75] references the problems linked with the saving of the stack.

Some architectures may also assume that the stack pointer is aligned at the beginning of a function. If a function call is inserted in the middle of a function, the stack pointer may need to be realigned at this point to ensure the invoked function works properly.

5.1.2.3 Preserving variables references

The behaviour of the executable also depends on its variables, some of which are present in the executable and directly accessed by assembly instructions. These accesses may reference the offset from the current instruction or use a fixed value. In the latter case, extended analysis may be needed to identify those values as such. For instance, let us consider the following C instruction printing a string:

```
puts("Hello sweetie!");
```

It may be compiled into the following Intel 64 assembly instructions:

```
mov $0x57484f,%edi
call <puts>
```

In the resulting assembly code, 0x57484f is the address of the string to print in the data section of the executable and `edi` a register used for passing parameters to functions. Detecting that this immediate value is actually the address of a variable used by the `puts` function is not straightforward, even more so if the invoked function is defined in an external library unavailable for analysis at the time of patching.

The patching operation may therefore impact these addresses as well, if they caused the offset between the instruction and the accessed data or even the address of the data in the file to change. Detecting those references present the same problems as those described for the branch instructions in 5.1.1.2.

5.1.3 Handling dependencies of inserted code

For easier implementation, instrumentation or profiling is usually done by inserting calls to *ad hoc* functions that have been already developed and compiled separately, the alternative being the insertion of the corresponding assembly code, which could be complex depending on the instrumenting function.

If the format of the patched binary file allows it, those functions can be referenced in a shared library, and the patcher must insert the required instructions and references for invoking such a function. This usually implies the edition of the relevant relocation table stored in the binary file, and adding the appropriate architecture specific snippet of code.

In other cases, it may be necessary to add the whole body of the function to the file. This can occur for instance if the patched executable is supposed to be standalone and not requiring any external libraries. In that case, the patcher needs to ensure that all required functions are added to the file, including those needed by the inserted function and its dependencies.

5.2 Methods and tools for instrumentation

We will first present here alternative solutions for instrumenting files with their pros and cons as well as methods commonly used in binary rewriting, then describe other tools allowing to patch or instrument files.

5.2.1 Compiler-based instrumentation

One alternative method for instrumentation uses the compiler. The code modifications are performed during the compilation process, so that the compiled executable contains all required alterations. This allows to benefit from the compiler internal representation of the code to ensure that the modifications will not damage the control flow of the generated executable, while preventing the transformations performed by the compiler on the code to be impacted by the modifications, which is the main drawback from source instrumentation.

This method is used by the GNU profiler **gprof** [63, 15, 99] to add probes during processing by the **gcc** compiler. Another use of this is described in [101], where executables for the MIPS processor are instrumented using additional information provided by the linker to create relocation tables allowing to update all branches inside the code, while branches to data section are not updated as the inserted code is added in a gap in addresses between the code and data section.

The main drawback of this method is that it implies being able to modify the code of the compiler used, which may not be possible if the compiler used in the files to be analysed is neither open source nor allows the addition of plug-ins. It can also lead to an increased implementation workload if multiple compilers must be supported, as each of them would have to be patched accordingly. It also requires access to the application sources in order to recompile it with the appropriate options, which is not always possible. Finally, while such methods allows to relatively easily instrument files on a function or loop level, it may be more complicated to specify fine grained modifications, such as at an instruction level.

A similar method would consist in analysing the file in order to rebuild the intermediary representation used by the compiler from the binary code, then modify it to apply the needed patching operation and reassemble it. However, decompilation is a complex problem that may not always have a reliable solution, and, since the purpose of patching is usually to retrieve information about the behaviour of the compiled file, this intrusive mode of patching could extensively change the structure of the generated code, like source patching would, and invalidate the results.

5.2.2 Dynamic patching

Dynamic patching consists in the modification of a file during its execution, either by using a supervising thread or by altering the executable after it has been loaded into memory.

5.2.2.1 Thread supervision

The use of a supervising thread, such as the one allowed under Linux by the system call **ptrace** [66], allows to control the execution of another thread and change its runtime data context. Debuggers are a common application of this method. It can be used in order to perform instrumentation in a file while avoiding problems tied to the preservation of the control flow and the data environment, since all runtime variables can be accessed.

One drawback of this method is that the process of patching the file will have to be repeated for every execution of the application, adding its overhead to all of them. It also requires an additional thread to be executed, which may not be possible or require additional configuration from the environment, for instance in the case of

an already multi-threaded code. Another drawback is that the whole code may not be accessible through this method, preventing some of the more detailed analyses of the executable to take place before settling on the patching operations to perform. Finally, while well adapted for retrieving information during execution of the program, this method may not be adapted for more intrusive patching operations, such as modification or deletion of instructions.

A further example of this technique is demonstrated by Periscope [62], which performs a distributed thread supervision of multi process programs.

5.2.2.2 Patching in memory

Patching the representation of the file in memory allows to access the code after it has been loaded and all operations such as relocations have been performed. It also allows to perform multiple patching operations during the execution of the file, such as removing a given modification in a loop after a certain number of iterations.

It suffers the same drawbacks as the thread supervision method in that the overhead of the patching process is added to each execution of the file, while presenting the same challenges as static patching regarding the preservation of control flow.

5.2.3 Simulation

Another method consists in simulating the execution of the application or dynamically translating it in a process similar to Just In Time compilation while performing the required modifications. The main interest of this method is that, since the whole executable is reinterpreted, it allows to avoid all problems tied to the conservation of the control flow or data environment, as modifications brought by the patch operations are integrated in the execution as if they had been present at compilation without altering the overall structure of the program.

This is done however at the cost of an important overhead, since the translation process can be time consuming. It can also present issues when handling multi-threaded applications, as it requires being able to emulate multiple threads and their communications, which are handled at the system level. It also requires an extensive knowledge of the behaviour of the ISA, which may not be easily available.

5.2.4 Code displacement

One of the most widespread methods for addressing the problems tied to the preservation of the control flow when patching a program is *code displacement*, also called *code relocation*. It consists in branching the control flow to an added area of code. Some instructions in the original code are modified to be replaced with a branch instruction addressing this new area, which contains the replaced code and any modification requested by the patching operation. The flow is then branched back to the original code. An implementation of this method as the `qp` and `qpt` instrumenters for the MIPS and SPARC processors is described in [69].

A drawback of this method is that the inserted branch to the displaced code may be larger than the code to be displaced. This can occur in architectures with instructions of variable size or if the branch operation requires more instructions than those composing the displaced code.

An alternative is the use of system interrupts instead of branches to jump to the displaced code. This presents the advantage that the interrupt operation takes

care of saving the context and transferring the control to the area of relocated code. In the case of Intel x86, it presents the additional interest of being accomplished with a 1-byte instruction, the shortest length, thus ensuring minimal alteration of the original code to insert it, since other branches are always longer. The main drawback of this alternative is the important overhead brought by the interrupt operation for saving the context and transferring the control.

Code displacement can be used either in binary rewriting or dynamic patching.

5.2.5 Patching tools

We will present here some existing patching or instrumenting tools and how they address the challenges of binary rewriting.

We will also present tools using binary rewriting for code optimisation or binary translation instead of instrumentation. While those tools have to address similar issues to those met by instrumenting tools, they do not necessarily have to obey the same constraints with regard to preservation of the original code flow, as they aim at actively modifying it.

5.2.5.1 DynInst

DynInst [41, 40] is a static and dynamic binary instrumenting tool allowing to patch ELF or WindowsPE files for the IA-32/Intel 64 and Power ISA architectures. It allows to insert function calls at different points of the executable CFG.

Instrumentation is done through code displacement using a double trampoline. The original code is modified to replace one instruction with a branch to an inserted area of memory containing the replaced instruction, and a branch to a mini-trampoline. The mini-trampoline is another inserted area of code containing instructions for saving and restoring the context, passing arguments to the inserted function and invoking it. This behaviour is used for static as well as dynamic patching.

A drawback of this double trampoline approach is the overhead it causes due to the consecutive branch instructions.

5.2.5.2 PEBIL

PEBIL [79] is a binary rewriting tool allowing to patch ELF files for the Intel 64 architecture. PEBIL allows to insert function calls and assembly snippets in binaries for instrumentation purposes. PEBIL addresses the problems of patching by performing code relocation at a function level. New code and data segments are created to contain the added and modified code. When the assembly code in a function needs to be patched, the whole function is moved to the new code segment and its code is altered as needed by the patching operation.

PEBIL also adds code around insertions for saving the registers and moving the stack to save the context. Functions that could not be properly disassembled are flagged as being ineligible for instrumentation, as well as functions too small to contain a branch instruction. This last case can however happen in Intel 64.

5.2.5.3 PIN

PIN [67] is a tool developed by Intel performing runtime instrumentation through thread supervision, allowing to trace an executable during its execution and monitor various parameters. It is available for Linux, Windows and Mac OS executables on

the IA-32 and Intel 64 architectures. Support for the Xeon Phi coprocessor architecture was recently added for Linux as well as for the IA-32/Intel 64 architectures under Android.

The tool also allows to insert calls to functions during the execution of the code, as well as modify the contents of memory during execution. The thread supervision mode induces slowdown for the instrumented files. It is also possible to insert probes in the code before execution, which redirect the execution flow to another function in a process similar to code displacement. A file so instrumented with PIN can be run under GDB. This mode does not work on multi-threaded applications and does not check if the destinations of branch instructions is valid, which can lead the patched file to crash.

5.2.5.4 Valgrind

Valgrind [82, 83] is an instrumentation platform for Linux executables under IA-32, Intel 64, ARM, Power ISA and MIPS architectures. It is a framework for building tools using its instrumenting abilities.

The core of Valgrind functions as a JIT compiler. It begins by disassembling the executable code and converting it into an intermediate RISC-like assembly language. Instrumentation is done on this intermediate representation, which is then recompiled into the assembly language of the target architecture and executed. The contents of the registers used by the architecture are stored in memory.

System calls are handled by setting up the system state as if the original code was still executing. Threads are handled by Valgrind, which executes them on a single thread that periodically switches between the simulated threads.

A drawback of the simulation method is its significant overhead, a program simulated with Valgrind without executing any additional instrumentation being 20 times slower than the original.

5.2.5.5 SecondWrite

SecondWrite is a binary rewriting tool [38] based on the LLVM framework [71]. It is intended to perform specific optimisation and security improvements [37] to binaries. The binary code is disassembled into the LLVM internal representation, which is then used to perform additional analysis on the code in order to deduce information not present in binary code, such as local variables and stack frames. The internal representation is then used to perform additional optimisations. The LLVM back end is then used to write the modified binary as for a standard compilation.

This method ensures an optimal flexibility in the handling of the disassembled code and ensures the correctness of the rewritten code. However, its efficiency to deal with instrumentation is more questionable, as the modifications required may cause the structure of the instrumented code to differ from the original, leading to problems similar to those arising when instrumenting the source code.

SecondWrite is a licensed project that seems to have only recently evolved beyond the prototype stage.

5.2.5.6 Diablo

The DIABLO framework [42, 51] allows to perform binary rewriting during the linking stage of compilation. It is available for the ARM, i386 and Power ISA

architectures and operates only on statically compiled files.

Since dynamically compiled files are widespread in most Unix and Linux based operating systems, to the point of being the default mode for the **gcc** and Intel **icc** compilers, this restriction is an important drawback. The insertion during compilation also suffers from the same limitation as the methods relying on the compiler as they require it and the compilation process to be accessible.

The project does not seem to have known significant activity for the last 4 years.

5.2.5.7 REINS

REINS [103] is an IDAPro [16] based binary rewriter aiming at increasing security of untrusted executables. The rewriter diverts branches to a verifier which checks their safety before authorising them. Due to mechanisms ensuring the continuity of the flow in the rewritten executables, their size can be doubled.

REINS is currently available for IA-32 executables for Windows and seems to be still at a prototype stage.

5.2.5.8 Binary Translation tools

These tools aim at converting an executable from an architecture to another, allowing to emulate a different instruction set than the one on which the file is being executed. Because the translation may occur between very different instruction sets, this specific type of patching usually causes important changes between the original and its modified versions. However, the code may be closer when translating between instructions sets defined for the same family architecture, such as IA-32 and Intel 64, or different subsets of the Intel 64 or ARM architecture.

StarDBT [102] is a binary translation tool from IA-32 to Intel 64, performing the translation at runtime. Since both instruction sets are close, with only some instructions in IA-32 being absent in Intel 64 or under a different form, the translator can preserve part of the original code, switching to the translated version when such an instruction is encountered or when an optimisation is possible. The translator maintains a lookup table referencing the addresses at which the flow must branch to the translated code, which can cause a significant overhead.

[46] presents a dynamic instrumentation method based on the PIN [67] tool, intended to be used for securing x86 executables by running them in a protected area. The method is based on transactional memory to handle binary translation over multi-threaded code and prevent race condition on metadata accesses.

5.2.5.9 Other tools

The following tools have undergone little to no activity in the recent years and may be abandoned.

The **Elfsh** utility and libraries, later merged with **ERESI** [53] (Reverse Engineering Software Interface), allow to disassemble, perform dynamic and static analysis, and patch ELF executables. It is available for IA-32, Intel 64 and SPARC architectures. Binary rewriting is done by inserting relocatable files into executables, and redirecting function calls to point to the inserted functions. The project seems to have received few contributions in the last years.

Etch [89] is a binary rewriting tool for Windows IA-32 executables. It allows to add, modify or remove parts of the code of an executable in order to perform

instrumentation or optimisation. Etch was released in 1997 and has undergone few evolutions since.

PLTO [92] performs binary rewriting, and include optimisation as well as instrumentation options. It operates during link time and therefore suffers from the same drawbacks as the tools tied to the compiler as they are dependent on its accessibility. It is used for IA-32 and does not seem to have been updated since 2001.

5.3 Binary rewriting using code displacement

Our solution to address the challenges tied to binary rewriting relies on code displacement, as described in 5.2.4. It is used for all modifications that would change the size of the code, such as insertion or deletion of instructions, and modification of instructions into versions with a different encoding size for architectures with variable instruction length.

The main constraint of the code displacement method is being able to correctly detect and update all branches or memory references to and from the displaced code. Since those are not always easily identifiable, our solution involves reducing the size of the displaced code whenever possible. This could be achieved by moving only a single instruction, and returning the flow to the instruction immediately following it; however in architectures with variable instruction length, a branch instruction could be larger than the instruction to be patched, making this simple replacement not possible. In some cases, performing a branch may also require the insertion of more than one instruction. The solution was therefore to allow varying scopes of displaced codes depending on the patched location.

We will first detail here our implementation of code displacement, then the various workarounds allowing to address the remaining problems, and the current limitations. We will also address the issues tied to the saving of context.

5.3.1 Conventions

In most cases, a single patching modification will target a single address: it will either consist in the insertion of a function call or code snippet, or the modification or deletion of the instruction at this address. The only exception would be the deletion or replacement of a series of consecutive instructions, but this can also be considered as a modification targeting the address of the first instruction in the series. In the remainder of this chapter, we will use the term *patched address* to identify such an address. Since the patcher always targets the code, we will use interchangeably the term *patched instruction* to identify the instruction located at the patched address.

5.3.2 Code displacement

When displacing code larger than the branch instruction to insert, padding needs to be added to ensure that the instructions following the displaced area remain at their original address and that the overall size of the code does not change. This padding is composed of `nop` instructions. Although the original location of the code should not be accessed when the patched file is executed, this ensures that no undefined operations will be performed if it is still somehow accessed, which could happen if an indirect branch pointing to this part of the code was not detected. The original

code is not left in place, as at least one instruction has to be replaced in order to add the branch instruction, which could cause an instruction to be split in architectures defining instructions of different sizes.

All displaced codes are appended to one another and grouped into a new section, which is added to the patched file. Any relative references to another part of the executable code or data are updated in the displaced codes to reflect the change of addresses. The virtual address affected to the new section must be outside of the original range of virtual addresses of the file. The displaced code being in a new area, any modifications can then be performed on it without any impact on the remaining of the code. The detailed implementation of code displacement is described in algorithm 11 and illustrated in figure 5.2.

Algorithm 11 Code displacement.

Require: `insns`: List of instructions to patch
Require: `displaced_insns`: Contiguous block of instructions to displace
Require: `patch`: Patching operations to perform

```
1:  branch ← NewBranchInsn()
2:  new_insns ← NewInsnList()
3:  AppendToList(new_insns, branch)
4:  while new_insns.size < displaced_insns.size do
5:    padding_insn ← NewNopInsn()
6:    AppendToList(new_insns, padding_insn)
7:  end while
8:  next_insn ← NextInsn(displaced_insns)
9:  ReplaceCode(original_code, displaced_insns, new_insns)
10: PatchCode(displaced_insns, patch)
11: SetBranchTarget(new_branch, FirstInList(displaced_insns))
12: return_insn ← NewBranchInsn()
13: SetBranchTarget(return_insn, next_insn)
14: AppendToList(displaced_insns, return_insn)
```

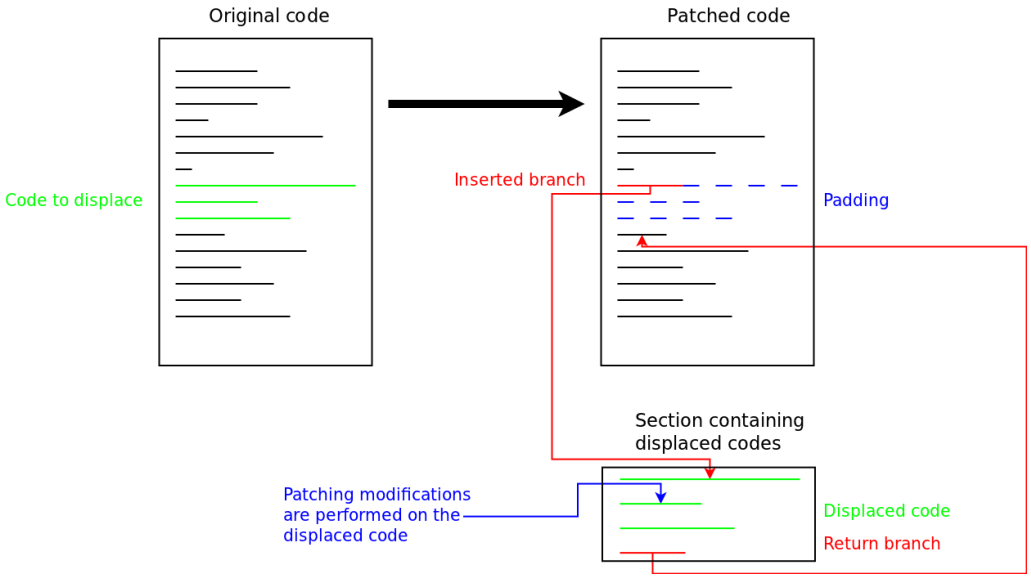

Figure 5.2: Patching a file using code displacement

The crucial step is the identification of the displaced code, which should be the narrowest possible to avoid moving the destination or origin of a branch instruction that was not detected, while being large enough to contain a branch instruction.

We devised three different scopes for identifying the code to displace, each of them leading to a potentially larger area of displaced code than the previous one. This allows to switch to the next scope if the current does not provide the necessary size. We also devised an alternative using trampolines if the scopes must not be changed or if none of them allow to achieve the required length.

5.3.2.1 Displacing single instructions

This scope offers optimal safety in the preservation of the control flow, as moving only the patched instruction ensures that all branches addressing it, even those that could not be detected, will address the inserted branch instruction instead. However it is not always possible if the patched instruction is smaller than the branch instruction to insert.

A workaround for this case consists in attempting to add instructions immediately following or preceding the patched instruction to the displaced code until either the length of a branch instruction is reached or a branch instruction or destination of another branch instruction is met. This fall-back allows to preserve a minimal number of displaced instructions without having to switch to a larger scope.

5.3.2.2 Displacing basic blocks

In this mode, the basic block containing the patched instruction is displaced. For the purpose of this operation, a basic block is identified as follows:

- It begins either:
  - at the first instruction preceding the patched instruction and addressed by a branch
  - after the first branch instruction preceding the patched instruction
- It ends either:
  - before the first instruction following the patched instruction and addressed by a branch
  - at the first branch instruction following the patched instruction
  - at the patched instruction if it is a branch instruction itself
- If the patched instruction is a branch, the block also includes all `nop` instructions following it

This scope offers a good compromise between limiting the size of displaced code and ensuring that it is large enough to contain a branch instruction. Another advantage of this scope is that it reduces the number of added branches if multiple patched instruction belong to same basic block, as it will be moved only once.

It is however possible to encounter basic blocks with the definition above that will be smaller than the shortest branch instruction. For instance, in the IA-32/Intel 64 architectures fnord, the instruction returning from a subroutine, `ret`, is coded on 1 byte, while the shortest branch instruction uses 2 bytes. Compilers can generate

code where a branch instruction points to a `ret` instruction immediately followed by non `nop` instructions, making it a 1 byte basic block.

The other drawback of this method is its reliance on the correct detection of branch instructions destinations, especially for indirect branches.

5.3.2.3 Displacing functions

The last scope moves whole functions. Functions are identified in a simplistic manner as the instructions contained between the first label immediately preceding the patched address up to but not including the label immediately following it.

The main advantage of this scope is that it ensures that the displaced code will have the required size in most cases, although it is still possible for a very simple function to be still smaller than a branch instruction.

It presents the drawback of increasing the size of displaced code, and thus the chances that some of the displaced instructions are targets of indirect branches which could not be identified and will not be updated, thus potentially leading to break the control flow.

5.3.2.4 Trampolines

Another method for handling displaced codes smaller than the standard branch instruction can be used if the architecture offers shortest branch instructions with a reduced range of accessible addresses. This is the case in Intel 64, where a 2-bytes branch instruction is defined (cf. 2.2.4.1), allowing to address instructions distant from at most 128 bytes.

If the displaced code is large enough to contain a shorter branch, a trampoline may be used. For this operation, the patcher attempts to find an area of code meeting the following criteria:

- It is eligible for being displaced given the current scope.
- It can be reached from the patched address with a shorter branch instruction.
- It is at least as large as two largest branch instructions.

This area can also be one to be displaced for another patching operation if it is large enough to contain an additional standard branch instruction.

If such an area is found, it is displaced as if it contained a patched instruction. Instead of padding, a large branch instruction, the *trampoline branch*, is inserted at its original location. The area containing the patched instruction is moved normally to the section containing displaced code and appended with a branch instruction addressing the instruction immediately following it as is the normal behaviour. The trampoline branch is set to address this displaced area. Finally, the area containing the patched instruction is replaced by a shorter branch instruction addressing the trampoline branch. The principle is illustrated in figure 5.3.

While it would be possible to link trampolines until an area suitable to contain a large branch instruction is found, this behaviour could lead to a "domino effect" causing large areas of code to be displaced and increasing the risk of displacing an indirect branch instruction causing a break in the control flow.

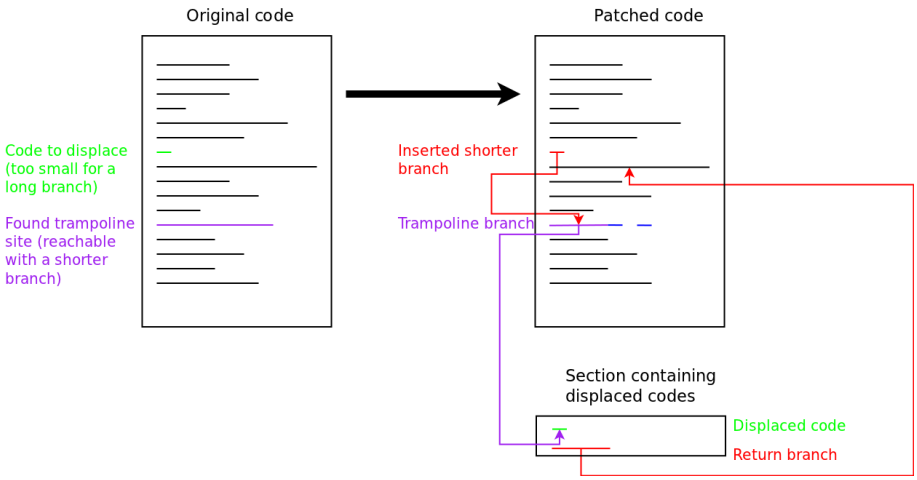

Figure 5.3: Illustration of the principle of using trampolines to displace code

5.3.3 Preserving the data environment

If the patching operation consists in the insertion of a function call, the inserted instructions performing it are surrounded by instructions for saving the data context before the invocation and restoring it afterwards. The instructions for saving the context perform the following operations:

1. Move the stack pointer to an alternate address and save its original value to the new stack. This address is chosen so as not to overlap with the existing stack. Alternate modes allow to use an area predefined when patching the file.
2. Save all registers, including those containing flags, to the new stack.
3. Align the new stack as specified by the ABI when beginning an executable. The address of the stack prior to its alignment is also saved to the aligned stack so as to allow its retrieval.

The instructions used to restore the context perform the reverse operations:

1. Reload the stack pointer with its value before alignment.
2. Restore all registers and flag registers from the stack.
3. Move the address of the original stack to the stack pointer.

This method ensures that the data environment will not be altered by the inserted function. Its main drawback is the overhead induced by all the load and store operations required to save the registers, which can be important if the architecture uses a large number of them, and have a significant impact on the overall performance of the patched file if the inserted function occurs in a loop with a high number of iterations. For these reasons, it is important to allow the number of saved registers to be tweaked in order to allow higher level analyses to identify those used in the patched function and restrict the saving operations to them.

5.4 The MADRAS patcher

We implemented the principles described in 5.3 into the MADRAS tool. The patcher functionality of MADRAS uses the instruction list returned by its disassembler func-

tionality as a base for its operations. It relies on an ELF editor to generate the patched file, which is the same editor used for parsing files during disassembly. The MADRAS patcher also uses the assembler whose architecture specific code is generated by MINJAG (cf. 3.4.3) to generate binary codings when needed.

Our main concern when implementing the patcher functionality was to offer a low level tool allowing an extended range of operations at the instruction level while keeping its use relatively simple. Another concern was reducing the amount of analysis required by the patching process so as not to increase its impact on the analysis chain, while its fine-grained approach allows to use the result of any higher level analyses to improve the safety of the operation performed with regard to the preservation of the control flow and data environment.

The MADRAS patcher is functional and currently supports the Intel 64 and Xeon Phi architectures, for the binary files using the ELF format. It is accessible through a low-level API described in Annex B.

5.4.1 Main features

The patcher offers to perform the following operations:

- Code modification
  - Insertion of function calls
  - Insertion of assembly code
  - Replacing a group of instructions by another
  - Deletion of instructions
- Data modification
  - Insertion of global variables
- Libraries modification
  - Renaming dynamic libraries
  - Adding new dynamic libraries
  - Adding new static libraries
- Label modification
  - Adding new labels

All code modifications cause the area of code where it must take place to be displaced as explained in 5.3.2.

5.4.1.1 Inserting assembly code

The inserted instructions are added into the displaced area at the patched address. If the inserted code has to be executed before the patched instruction, it is inserted before it. In that case, an extra step consists in updating all branch instructions addressing the patched instruction to point at the branch addressing the displaced code instead, so as to ensure that the inserted code is always executed first regardless of the path accessing it.

5.4.1.2 Inserting a function call

The patcher allows to insert a call to a function either already present in the executable or defined in a dynamic or static library. The assembly instructions for the function call are inserted into the displaced block at the appropriate location, surrounded by instructions needed to preserve the context (cf. 5.3.3).

If the function is defined in a dynamic library, a new block of code containing the architecture-specific instructions for the stub used to invoke an external function is added with the appropriate instructions in a separate list of inserted instructions. The inserted call is set to address this stub as is the normal behaviour for external functions. If the function is defined in a static library, the code of the function is inserted into the patched file, with the inserted branch addressing the entry point of the function. In both cases, relocations are handled by the ELF editor.

A function call insertion is otherwise handled as an insertion of assembly code, with the instructions for invoking the function and preserving the context considered as the inserted code.

5.4.1.3 Replacing or deleting a group of instructions

Replacing a group of instructions can serve two purposes:

- Modifying an instruction, which amounts to replace it by another instruction.
- Disabling instructions by replacing them with `nop` instructions.

This modification is the only code modification that may not cause the area where it takes place to be displaced, as it does not change the size of code. In that case, the modification directly takes place in the original code. The exception is the replacement of an instruction with another coded on a larger length.

Instructions can also be deleted and are then removed from the displaced block.

5.4.1.4 Insertion of global variables

The contents of inserted global variables are copied to a new data section added to the patched file. Those variables can be referenced by inserted codes.

5.4.1.5 Conditional executions

The MADRAS patcher also offers to set conditions on the execution of an inserted function or assembly code. These conditions are compiled into assembly instructions that are added to the inserted code and ensure it will be executed only if the conditions are met. This allows for instance to execute an inserted code only for some iterations of a loop, or for more complex uses insert different codes to execute depending on the possible values of an operand used by an indirect branch.

5.4.2 Customisable behaviour

In order to ensure the low granularity of the MADRAS patcher, most of its behaviour can be controlled through toggles either for a whole patching session or for separate patching operations while offering default modes for the general cases.

Scope selection It is possible to choose which kind of scope to use when identifying areas of code to displace, and whether the patcher can switch to a broader scope if the current one could not find an appropriate area. The use of trampolines can also be disabled. By default, instructions are displaced at the block level and trampolines are used whenever needed.

Branches updating The MADRAS patcher allows to selectively update the branches to patched instructions when inserting code that must be executed before it. It is possible to request for only some branches to be updated while leaving the others addressing the original patched instruction. This can be useful for instance if the inserted code aims at profiling the entry point of a loop or function, and should not be executed when accessed from inside.

Saving the context The instructions for preserving the context when inserting a function call impact all registers of the architecture by default. It is possible to change the registers concerned or remove the whole preservation of the context, allowing the application using MADRAS to fine tune the registers to save based on the knowledge of the inserted functions, thus reducing the overhead induced by the patched code.

5.4.3 Inner workings

We will briefly describe here the internal mechanisms involved in the processing of a file by the MADRAS patcher.

5.4.3.1 Interface with architecture specific code

The patcher handles multiple architectures through the use of a set of architecture specific functions which must be defined for each supported architecture. The principal operations performed by those functions involve the generation of architecture specific instructions lists, including the following:

- Instructions allowing to invoke a function, including passing its parameters and retrieving its return value.
- Instructions allowing to enforce a set of conditions.
- Branch instructions, including shorter versions of branches if they exist for this architecture.
- `nop` instructions.
- Stubs for invoking a dynamic function.

The use of such functions allow the main code of the patcher to remain agnostic with regard to the architecture and reduce the implementation needed when porting the patcher to a new architecture, in keeping with the main concerns involved in the design of MADRAS.

5.4.3.2 Finalising a patching session

The patcher performs all of the requested modifications at once, then finalises the patching session and generates the patched file.

During finalisation, the patcher ensures that all modified and inserted instructions are properly encoded, which includes the update of branch instructions and of instructions referencing data sections in the file. Dependencies of inserted static libraries are checked in order to ensure that they have also been inserted. The patcher also checks the presence of unconditional branches addressing other such branches, which can happen if a branch and the instruction it addresses have both been displaced, inducing unnecessary rebounds. Those rebounds are replaced by a single branch instruction in order to avoid inducing an overhead by performing unnecessary branches.

5.4.4 Limitations

The current version of the Madras patcher assumes that the added section containing displaced code is reachable from the original code with a direct branch instruction. In Intel 64, such instructions encode their offsets on 32 bits, allowing for an absolute offset of 2 Gb. If the section containing displaced code was added at an address more than 2 Gb apart from the original code, this instruction will not be usable to branch to and from this section. Such an offset may occur for instance if the displaced code was to be added after the section containing uninitialised data, which can reach such a size for some executables. In such a case, the patcher would need to use an indirect branch or use a trampoline.

Another limitation is that the patcher does not update indirect branch instructions when they are moved. This is not a problem in the Intel 64 and Xeon Phi coprocessor architectures, as all indirect branches they define use absolute addressing, but needs to be addressed when porting Madras to architectures where these branches can contain relative addresses. This is for instance the case with some indirect branch instructions in the ARM architecture.

5.4.5 Use in Maqao

The Madras patcher is fully integrated into the Maqao framework [39], allowing Maqao to complement its static analysis with profiling and memory tracing. The language Mil [43] (Maqao Instrumentation Language) offers extensive instrumentation functionalities and uses Madras for performing the low-level operations. Figure 5.4 describes the integration of Madras into the Maqao framework and its relationship with Mil.

Mil offers a simplified interface for instrumenting files, inserting function calls or assembly code at various points in the CFG. It has been used to integrate the Maqao instrumenting features to the TAU performance analysis tool [94].

The instrumentation functionalities of Mil were tested on the NAS parallel benchmarks using OpenMP [55], allowing to establish a comparison on the overhead induced by the instrumentation process with the DynInst and PEBIL instrumentation tools. The results presented in figure 5.5 show that code instrumented with Mil using Madras offers the lowest overhead and the better coverage for the files considered in the experiment.

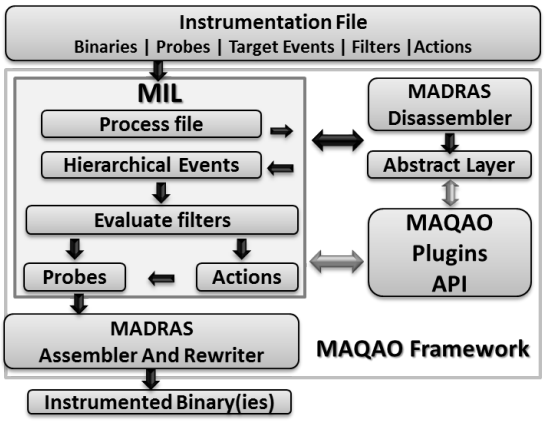

Figure 5.4: MIL: MAQAO Instrumentation Language and its integration in the MAQAO framework. Source: [43]

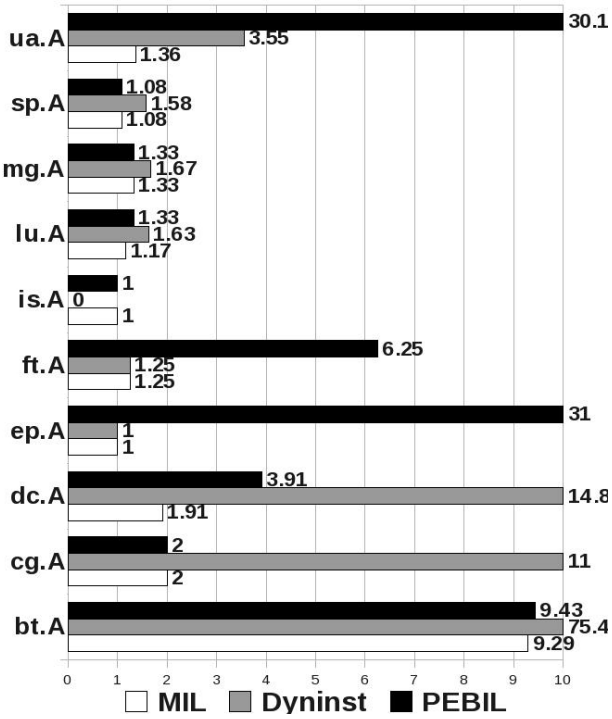

Figure 5.5: Comparing overhead time on NAS OMP benchmarks for MIL, Dyninst and PEBIL using TAU. X axis reports the overhead ratio compared to the original run. Lower is better. Overhead ratios greater than 10 are cut. A zero ratio means a crash at runtime. Source: [43]

5.4.6 Use by DECAN

DECAN [68] is a MAQAO module performing decremental analysis. This consists in removing selected instructions accessing memory from executables in order to identify their impact on the overall performance. DECAN relies on the MADRAS API to generate the modified executables, its fine granularity allowing it to operate at the instruction level and precisely pinpoint which instructions in a loop or function are responsible for the cycles spent. The transformations performed by DECAN include:

- Removing some or all load and/or stores instructions from a given loop.

- Replacing memory address operands with a fixed memory address.
- Replacing instructions store instructions with loads and reciprocally.

DECAN also inserts assembly code for timing the transformed loops. MADRAS allows to restrict these transformations to a single iteration of a given loop thanks to its handling of conditions.

5.5 Conclusion

In this chapter, we have presented an application of the list of instructions returned by the disassembler in the form of a binary rewriter. This patcher performs code displacement to address the challenges of binary rewriting and operates at low level with a fine granularity in order to offer an extensive range of operations to perform. It has been implemented as MADRAS for the Intel 64 and Xeon Phi coprocessor architectures for the ELF files. The MADRAS patcher is functional and used in the MAQAO framework for instrumentation with better performance and coverage than other notable instrumentation tools.

We will now present extensions to the work described so far in this dissertation.

Chapter 6

Extensions for Madras and Minjag

Separate complementary solutions were presented in the previous chapters for allowing a low-level access to binary code, while minimising the overhead brought by the handling of evolving and varied architectures. These solutions were validated by their implementation into the functional Madras tool and its associated code generator Minjag. This chapter will now describe extensions allowing to enhance the performance and abilities of this tool and further validate the principles involved in its conception.

As an essential component of a functional framework used for research and industrial projects, Madras is subject to constant updates either required by the new features implemented in Maqao or aiming at improving its performance, fine-tuning its interface with Maqao, and expanding its coverage. We will focus here on extensions affecting multiple components between the Madras disassembler, patcher, and Minjag.

We will first describe methods allowing to improve the performance of the disassembler, based on fine-tuning of the grammar representing the architecture and its associated parser, in Section 6.1. We will then present constraints and solutions brought by the implementation of new architectures, in particular ARM, in Section 6.2. We will finally offer leads and methods for improving the coverage of the patcher, in Section 6.3.

6.1 Optimising disassembly performance

The Madras disassembler is mainly used as the entry point of other tools for further processing, which may be time consuming in itself. It is therefore important for the disassembler to offer the best possible performance to avoid being a bottleneck in the processing as a whole, and to offer accurate information so as to not invalidate the end result of the whole analysis chain. One of the main goals of the extensions of the disassembler is therefore the continuing increase of its speed and accuracy. Special care must also be brought in avoiding for the implementation of the other evolutions to be made at the expense of its performance, especially in speed.

6.1.1 Optimising disassembly speed

Profiling performed on the disassembly of large files showed that parsing represents more than 30% of the total disassembly time. Improving the parser performance is therefore liable to have a significant impact on the speed of the whole disassembly process.

We believe however that the current state of the generated parser is close to the maximal optimisation available while keeping an agnostic approach with regard to

the architecture and that further improvements would offer a low return on investment. The achievement of significant improvements would then require allowing to tweak the generation process or the resulting FSA in order to optimise the resulting disassembly process with regard to the specificities of the architecture. A last resort would consist in performing architecture-specific optimisation based on the generic generated parser in order to handle cases where obtaining the optimal disassembly speed is an issue.

Other leads that we will not cover here consist in improving the implementation of the parallel mode described in 4.4.2.

6.1.1.1 Impact of the grammar format

Since the main motivation behind our work is to ease up the addition of new architectures and the update of those already implemented, the main constraint is to use a representation of instructions as close as possible to the format used in the architecture documentation. It is for instance not acceptable if the addition of a single mnemonic to the instruction set causes the edition of a dozen different lines in the grammar, especially if this implies complex transformation. While this constraint weighs on the choices made during the conception of the grammar representing an architecture, it is however possible to devise different grammar structures fulfilling these needs.

The formalism chosen for the grammar directly impacts the structure of the FSA and therefore the performance of the resulting parser. A proportionally high number of nonterminal symbols in a grammar will result in a higher number of reduction states in the corresponding FSA. The parsing of a word following such an FSA will involve multiple invocations of the functions handling reductions and of the associated semantic actions, possibly leading to a slowdown compared to other representations using less nonterminals.

For instance, let us consider the following possible encodings, where `A`, `B`, `C` and `D` represent expressions represented by nonterminals:

```
A B 1000 C
A B 1010 D
B 1001 C
B 1011 D
```

A possible grammar for representing these coding rules could be a straightforward transcription of them as productions of the start symbol:

```
Start: A B 1000 C
| A B 1010 D
| B 1001 C
| B 1011 D;
```

Using such a grammar, the parsing of a word represented as the first encoding of the list (`A B 1000 C`) would go through the following steps:

1. Attempting to match the input against productions of `A` and `B`
2. Reduction of `A`
3. Attempting to match against productions of `B`

4. Reduction of `B`
5. Attempting to match against `1000`
6. Attempting to match against productions of `C`
7. Reduction of `C`
8. Reduction of the start symbol

However, another possible grammar representing the same coding rules could distinguish codings beginning with the `A` symbol. A possible motivation for this could be the simplification of semantic actions if specific actions are associated to encodings beginning with the `A` symbol. The associated grammar would therefore have the following structure:

```
Start: A AfterA | NotAfterA;
AfterA: B 1000 C | B 1010 D;
NotAfterA: B 1001 C | B 1011 D;
```

The steps performed for parsing the first encoding in the list according to such a grammar would be identical to those for the simpler version of the grammar, with an additional step between steps 7 and 8 consisting in the reduction of the `AfterA` symbol. This alternate grammar therefore imposes an additional step during the execution of the parser. While this description may allow an easier implementation of the semantic actions, possibly avoiding additional tests, it is likely to increase parsing time.

Conversely, a grammar using a restricted number of nonterminals, possibly reduced to the start symbol only, will result in a higher number of transitions for the first state of the resulting FSA. This can impact the performance of the resulting parser, especially if the transitions overlap because of the presence of unfixed bits, as this case imposes multiple checks for matching transitions.

6.1.1.2 Grammar optimisation hints

A promising lead for optimising disassembly performance therefore involves the possible reformatting of the grammar to reduce the number of reduction steps or of matching transitions. While those general guidelines can be helpful by themselves, their implementation into the grammar format can be made easier using metrics deduced from grammar parsing and FSA generation. These metrics can be returned by Minjag to provide hints for improving the grammar. We present here such metrics, along with their methods of retrieval and their uses for improving a grammar.

Estimating the number of reductions By recursively counting the number of nonterminals involved in a production, it is possible to know the maximum and average numbers of reductions necessary to reduce a given symbol, excluding the cases of recursive symbols by considering they can appear only once for the purpose of establishing this metric. This allows to provide the average and maximal numbers of reductions required for parsing a word for the given grammar, along with the productions and symbols for which the maximum is reached. This metric is useful for identifying potentially excessive number of reductions which could slow down the parsing process. The knowledge of the specific grammar lines or symbols

responsible for this can help pinpointing the choices in grammar formalisation causing this potential loss of performance. This can be correlated with the number of productions per symbol to allow identifying nonterminal present in multiple productions while having only a single production; such symbols might be replaced by this production in all the productions into which they appear to reduce the total number of reductions during parsing. Conversely, this metric can be used to check that the grammar has been correctly optimised with regard to this problematic.

Number of states and transitions Another metric available after generation of the FSA consists in the number of transitions per shift states. A simplified example of the results brought by this metric can be found in table 3.1, characterising the grammars tested in Minjag. This allows to survey the repartition of transitions per states and to pinpoint states containing a single transition. The identification of those states and the associated grammar productions can help to detect possibly unnecessary decompositions of encodings resulting in extraneous shift steps. Another related metric consists in identifying the states containing transitions with multiple matching values.

FSA profiling In order to weigh the importance of the hints returned by the previous metrics, a profiling of the FSA can be performed. This is done by activating a special mode during disassembly where the parser logs the number of hits over each state and each transition inside them. This knowledge is especially useful for identifying the states most accessed during the disassembly of standard executables, and therefore deduce which of them are liable to offer the best performance improvement if optimised. This can also be used to optimise the disassembler with regard to the type of files targeted by the analysis tool using it.

6.1.1.3 Tweaking the FSA generation

Another option for optimising the disassembly consists in improving the resulting FSA. While the reformatting of the grammar is best performed manually, this optimisation can be performed during the generation process. A possible way for doing this is through the addition of optimisation flags to Minjag, allowing to enforce the addition of specific behaviours or assumptions to the generated parser. These optimisations can act upon the metrics described in 6.1.1.2, possibly including the results of profiling sessions for restricting their targets, and perform transformations during parser generation equivalent to the corresponding grammar updates. Since such transformations may slow down the parser generation process and may not result in increased disassembly performance, they are left optional. Some of those possible transformations are presented below.

Using extended states The parser generation can detect states with a single transition and automatically extend the transitions of the states leading to them to append this single transition. This is however not straightforward, as the reason this was not done during standard parser generation is that the single transition uses a reduced symbol while those leading to it use binary values, or reciprocally. It is therefore necessary to use special states able to perform both matchings at once, which is mainly applicable if the transition on binary values is the lone transition. Another possible optimisation concerns the reductions of symbols appearing at the

end of productions, which can be replaced by an extended reduction in order to avoid multiple consecutive reductions. Such transformation involves the use of extended states allowing to perform such multiple reductions. A combination of those transformations can also be envisioned through the use of extended states allowing to perform matchings and reductions simultaneously.

Reorganising states Another optimisation aims at implicitly rewriting the grammar in order to reduce the number of states. This can be achieved by attempting to automatically identify cases such as the one in example presented in 6.1.1.1. One possible lead is the detection of identical groups of productions appearing in independent symbols. Another method involves the identification of similar transitions between states reachable from identical prefixes. This optimisation may however prove counterproductive if it prevents other simplifications to be applied, for instance in the handling of semantic actions and needs to be validated. An hybrid solution allowing user input consists in presenting the generated FSA as a manually editable graph.

6.1.1.4 Interacting with the parser during disassembly

Another extension consists in the implementation of a very low-level API allowing to directly access the inner mechanisms of the parser in order to tweak its behaviour. The semantic actions could therefore use this API to take control of the parsing process while it is executing. This would in effect allow Minjag to generate a procedural parser.

This allows to add a measure of architecture specific code to the disassembler from the base granted by the grammar representation of the architecture. For instance, it would be possible to detect the first bits of special elements of the coding and bypass the normal parsing process to either match a sequence of bytes against its possible values or only check the relevant bits and directly generate the corresponding representation of instructions. Other possibilities involve the advance detection of cases that should not occur and directly skip them.

An example of this technique with the Intel 64 architecture would involve detecting the ModR/M byte in advance and directly match its possible values in a table directly triggering the creation of the corresponding structures. Another example involves the detection of opcodes not using this byte and bypassing the transitions to retrieve the values of the following fields by directly storing the binary stream into the structures representing instructions.

An extreme application of this extension would be equivalent to using Minjag for generating a base representation of the encoding format and use this to build an architecture specific disassembler.

6.1.2 Optimising disassembly accuracy

Detecting interleaved foreign bytes can be performed through the heuristics and methods described in 4.3.1, which can be extended and further combined to handle more cases.

Other methods involve the identification of jump tables inside the code through a combination of pattern matching and detection of accesses inside the executable code, correlated with a validation of the addressed instructions.

6.2 New architectures

Since Madras was designed as a multi architecture tool, implementation of new architectures is a crucial point in the validation of the theories involved in its conception. One branch of extensions therefore aims at validating the theories implemented in Minjag and expanding its functionalities and coverage by testing it on different architectures of interest to the HPC community, such as POWER ISA and Intel Itanium, and completing the implementation of ARM.

The most challenging aspect of the ARM architecture is presented by its *interworking* instruction sets, which allow for instructions belonging to two different sets to be present in the code section of a binary file. We should point out here that this is different from the technique consisting in packing multiple executables compiled for different architectures inside a single file, such as is allowed by the Mach-O multi-architecture binary format, as in this case each packed file contains information allowing to retrieve its instruction set, while in ARM files using interworking the information available from the format of the binary file contains no indication about the different architectures involved.

6.2.1 Handling multiple instruction sets in a file

The principal challenge when dealing with the ARM architecture is the possibility of a binary file to contain instructions belonging to the ARM and Thumb instruction sets. Since the binary encodings of those sets are overlapping, it is necessary to identify when the binary code switches from one to another in order to avoid disassembling erroneous instructions. Another impact of this possibility is that it requires the disassembler output to include for each instruction the reference to the architecture for which it is defined instead of storing this information at a higher level, which may impact the memory used for representing disassembled instructions handled by analysis tools.

6.2.1.1 Representing a list of instructions in the grammar

One possibility for this would be the redefinition of the grammar as describing a whole binary stream instead of a single instruction. The start symbol of such a grammar would then be reduced into a list of instructions, with the corresponding parser considering a binary stream as a word. In this case, separate symbols would be used to represent instructions from the different instruction sets involved, while the instructions specifying the switch between sets would be represented as distinct symbols.

Below is a simplified example of such a grammar:

```
Arch1_InsLst: Arch1_Ins Arch1_InsLst | Arch1_Ins ;
Arch2_InsLst: Arch2_Ins Arch2_InsLst | Arch2_Ins ;
Arch1_First: Arch1_InsLst SwitchToArch2 Arch2_First | Arch1_InsLst ;
Arch2_First: Arch2_InsLst SwitchToArch1 Arch1_First | Arch2_InsLst ;
Start: Arch1_First | Arch2_First ;
```

The parser and especially the disassembler would have to be adapted to take into account the fact that it returns a list of instructions. This especially impacts the handling of post-parsing actions, which would now be triggered after disassembling a symbol representing an instruction, identified through the keywords indicating

such a case in the associated semantic actions. The impact on encoder generation is minimal, affecting only its input which change from a single instruction to a list, while its output remains a binary stream. The implicit automaton used for assembling will be able to use this representation to detect the relevant branch instructions in the input list and adapt its parsing rules accordingly.

However, the main drawback of this approach is that it makes any particular processing performed at the instruction level more complicated, especially if it involves information not available from the executable code. This is the case for any information retrieved from other sections in the binary file, such as labels or virtual addresses. This will also impact any attempts at increasing the disassembler accuracy through recursive traversal techniques, as this would require the parser to be able to follow the control flow inside the stream it is decoding.

For these reasons, this method will not be used for addressing this specific case.

6.2.1.2 Bundling different architectures in a grammar

Another method for handling multiple architectures in a single file that is closer to the current implementation of the disassembler consists in switching to the appropriate parser during disassembly. In this model, the two instruction sets are represented as distinct grammars, each with a corresponding parser. The difficulty resides in the detection of the instruction triggering the switch and the identification of the parser to invoke when it is discovered, while keeping the agnostic approach of the disassembler with regard to architecture.

Currently the disassembler deduces the parser to use for a file from the information retrieved from its binary format. ARM files using interworking reference only one instruction set, making the detection of the presence of the other instruction set more difficult, especially in the context of avoiding the use of architecture specific information at the higher levels of the disassembly process.

Our solution consists in allowing multiple grammar files to be bundled together. This is different from the previous solution as both architectures will be described in separate grammars that will be then merged inside a single file with a specific header. The header of a grammar bundle contains the names of the architectures representing the different grammars it contains, and, for each of them, the production and associated semantic action allowing to switch to another grammar.

Below is a schematic example of such a bundle:

```
%bundle arch1, arch2
%switch arch1 arch2
<list of lines from grammar arch1 performing the switch>
...
%switch arch2 arch1
<list of lines from grammar arch2 performing the switch>
...
%grammar
<full grammar for arch1>
%grammar
<full grammar for arch2>
```

When parsing a bundle file, MINJAG identifies each grammar and processes them separately, building the corresponding FSA as if operating on a single file. It also

singles out for each grammar the productions referenced in the header, and adds a special flag to their corresponding semantic actions in the resulting FSA, referencing the architecture to switch to. This flag can then be picked up by the disassembler during execution, allowing it to perform the switch to the specified architecture as if it had detected it from a file header. Minjag also adds the information about the interworking architectures to the sources it generates containing additional information retrieved from the grammar for each architecture, possibly including references to the instructions performing the switch.

One advantage of this method is that it allows to update the bundled architectures separately, in keeping with the general principles involved in the conception of Minjag and Madras. Another is that it provides the necessary information to the disassembler to allow it to handle the change without needing to have additional knowledge of the architectures it is handling, thus preserving its agnostic approach.

6.2.1.3 Patching interworking files

As the patcher relies on the list of instructions returned by the disassembler, it can identify the architecture for which each instruction is defined. This enables the invocation of the relevant associated assembler when propagating patching modifications to the binary code of instructions. Similarly, this allows to identify the architecture for which must be defined the instructions added to perform code displacement or context preservation.

Since the information about the instructions performing switching between architectures retrieved from the grammar is also made available to the assembler, the patcher is able to detect those instructions and adapt the used encoder accordingly. This is mainly relevant when performing patch operations for inserting assembly code from its language representation, for instance in the case where patching aims at inserting code defined for a different architecture than the one used for instructions present at the site of the insertion.

6.2.1.4 Disassembling interworking ARM files

The handling of ARM files imposes additional processing, as the instructions used to switch from an instruction set to another are branches. It is therefore necessary to correctly identify their destination to allow the disassembler to apply the proper parser, which can present problems if the branch is indirect. Another possible issue is one of these branches pointing to an area of code that was already disassembled with a possibly wrong architecture.

The ARM ABI defines specific label names identifying the areas of code belonging to each different architectures, which can be used to complement the detection of switching instructions during disassembly. Since the format used for the grammar describing architectures allows the definition of ABI-specific information, the nomenclature for these labels can be added to the bundle header or grammar files, allowing it to be available in the generated code. The disassembler can then use this knowledge to correlate information retrieved from the binary file with the possible interworking architectures referenced in the parser to switch the FSA used for disassembly appropriately.

An alternate method involves the use of recursive traversal disassembly techniques, possibly restricted to branches flagged as performing an instruction set switch. A drawback of this approach is the possible excessive cost induced by

the analysis required to identify the targets of indirect branches and the possible repeated disassembly of blocks of code incorrectly identified as belonging to the wrong instruction set. It is however useful for handling cases of files lacking information for otherwise identifying the switches between instruction sets, which could happen for specific compilers settings or implementations.

6.2.2 Other architectures

The support of Intel Itanium is another interesting challenge in order to validate the implementation of the extended LR algorithm described in Section 3.3.4, although the dwindling representation of this architecture in the HPC community makes it a less likely target for analysis tools. Current testing has shown that a grammar representing the Intel Itanium encoding rules could be processed without errors by Minjag.

Another constraint brought by Itanium with regard to the handling of instructions in Madras and Maqao is tied to its use of bundles, which regroup 3 instructions. This impacts the coding and hence the expression of instructions in the grammar. It is therefore necessary for the disassembler to take into account the fact that a parsed word is not a single instruction. The drawbacks of using the parser to return a list of instruction that were described in 6.2.1.1 do not apply in this case, as the structure of the architecture ensures that no extensive operations such as associating labels will have to be performed at the instruction level.

6.3 Patcher extensions

As a consumer of the list of instructions returned by the disassembler and user of the code from the grammar, the patcher will benefit from extensions to either of these components. In particular, the support of more architectures by the Madras disassembler and assembler will allow the patcher to process files for those architectures with reduced implementation effort, mainly focusing on devising assembly instructions to use for performing context displacement or context saving. While the behaviour of the patcher has been designed to not be dependent on the architecture, the support of new architectures may also require the addition of further mechanisms to ensure an optimal coverage of the available codes, such as handling interworking code as described in 6.2.1.3.

We will focus here the extensions tied to improvements of the disassembly process, the assembly process, and finally those offering new patching options.

6.3.1 Consequences of improved disassembly accuracy

The main extension tied to the evolutions of the disassembler include the detection of jump tables. Extensions described in 6.1.2 allow to retrieve possible locations of those tables when interleaved with code, which can be confirmed using pattern matching correlated with analysis of the addresses present in the file. Being able to identify more precisely these tables used by indirect branches allows to update them when the addresses they contain correspond to instructions moved by the patch operations and therefore increase the options available to the patcher for performing the requested updates, resulting in better coverage.

6.3.2 Increasing performance

Since the Madras patcher is designed to be part of a tool chain including disassembler and analysis modules, and is intended to be used on complex or large files where disassembling or analysis could be time consuming, obtaining the best performance for the patching operations would prevent it to become a bottleneck. Extensions aiming at improving the patching speed are therefore equally important. One such extension concerns the assembler built from the grammar. Since the encoder follows an algorithm similar to packrat parsing, as it must give priority to the shortest possible encodings for a given instruction, it can use memoization for better performance, by storing the results of past matches to speed up further assembly operations. This is especially useful when assembling list of inserted instructions or updating portions of code modified by the patching operation.

6.3.3 Decorrelation of the patching process

Another branch of works aims at allowing to perform differed patching, by converting the requested operations into a patch to be applied on the executable, separately from the other operations performed in the analysis chain. A further extension consists in the generation of patcher programs allowing to record and execute a certain set of modifications to perform on a file. This is especially useful for patching similar files without additional analyses.

For instance, analysis tools may need to instrument specific functions present in multiple files, such as generic functions inserted by the compiler to support OpenMP or MPI processes. This extension would allow Madras to generate a simplified patcher targeting the specified instrumentation sites defined in these functions if present in the file, therefore offering the possibility of patching all concerned files without the overhead induced by the corresponding analyses for each of them.

6.4 Conclusion

In this chapter, we presented the principal extensions to the Minjag and Madras tools allowing to increase the performance and coverage of the disassembler and patcher. The performance of the disassembler can be improved by allowing to tweak the generated parser, possibly including architecture specific code if speed is an important issue. The structure of Madras and its reliance on code generated by Minjag ensures that implementation of new architectures very different from Intel can be done with minimal additional effort. Finally, while benefiting from the extensions to the disassembler and encoder generation, the patcher can be extended to improve its coverage and performance.

Chapter 7

Conclusion

We will now conclude this dissertation. We will first list the contributions of this thesis then present further applications of its results and some of the research perspectives they offer.

7.1 Contributions

The thesis presents methods allowing analysis tools to operate at the binary level on executables, which offers the most accurate representation of what will be executed by the processor, but requires the ability to disassemble and patch files from multiple evolving architectures. Tools operating on binary files usually maintain a hard-coded representation of the binary architecture, leading to higher maintenance cost for implementing new architectures and updating existing ones. They may also not allow to customise the actions to perform when disassembling instructions, nor have a fine-grained enough approach to patching.

Our first contribution is a solution for describing an ISA under a unified format inspired from context-free grammars. We have shown that the use of LR-parsing for disassembling binary code is possible with some tweaks from the standard theory in order to take into account its specificities, namely the fact that instructions are not separated from one another and can be of different length. This was addressed by considering bit fields, instead of single bits, as terminals in the grammar sense. The case where bit fields are a generalisation of one another was addressed by ordering transitions in the generated automaton based on their level of precision. This allowed us to build a disassembler generator, Minjag, using a grammar whose structure is kept as close as possible to that of the ISA specification to allow for easier updates. The grammar format also allows to easily build an assembler that constitutes a basis for a patcher.

Our second contribution is an application of the code generated by Minjag to implement a disassembler relying on LR-parsing, whose output is a unified representation of instructions that can be expanded depending on the needs of the tools using it, and allows them to remain agnostic with regard to the architecture used. This disassembler is functional for the Intel 64 and Xeon Phi coprocessor architectures and under testing for the ARM architecture, and is easily updated to follow the architectures frequent evolutions. It is also able to return additional information about disassembled instructions in a unified format. The performance of the resulting disassembler is close to those of tools using hard-coded definitions of the architecture coding rules in terms of speed, accuracy, or both, while allowing for easier updates to keep up with the evolution of architectures and implement new ones, and being available as a single executable or library. This allows to build analysis tools able to handle multiple architectures, which is essential for the analysis of heterogeneous systems.

Our last contribution makes use of the representation of instructions in a disassembled file to offer a patching functionality. Patching presents specific challenges, such as the preservation of the control flow and data environment. This was addressed by performing code displacement on patched areas. The implementation of the patcher focused on offering the finer-grained approach possible, in order to allow the higher level tools using it to choose the better options for the given code.

The disassembler and patcher were integrated into a functional tool, MADRAS, which supports ELF files for the Intel 64 and Xeon Phi coprocessor architectures. The MADRAS disassembler aims at offering a good compromise between speed and precision, while providing helpful hints to the analysis tools using its output. The MADRAS patcher offers a low-level interface allowing a fine grained control of its actions, while being able to handle most cases in its default behaviour.

MADRAS is part of the MAQAO analysis framework and is an essential part of its life cycle. The disassembler is the entry point of all static analyses, while the patcher is used by the MIL and DECAN modules to perform all instrumentations or other transformations. The multi architectural nature of MADRAS, provided by MINJAG, ensures that MAQAO can be easily adapted to support new architectures with minimal implementation effort. Figure 7.1 summarises the tools relationships and modes of operation.

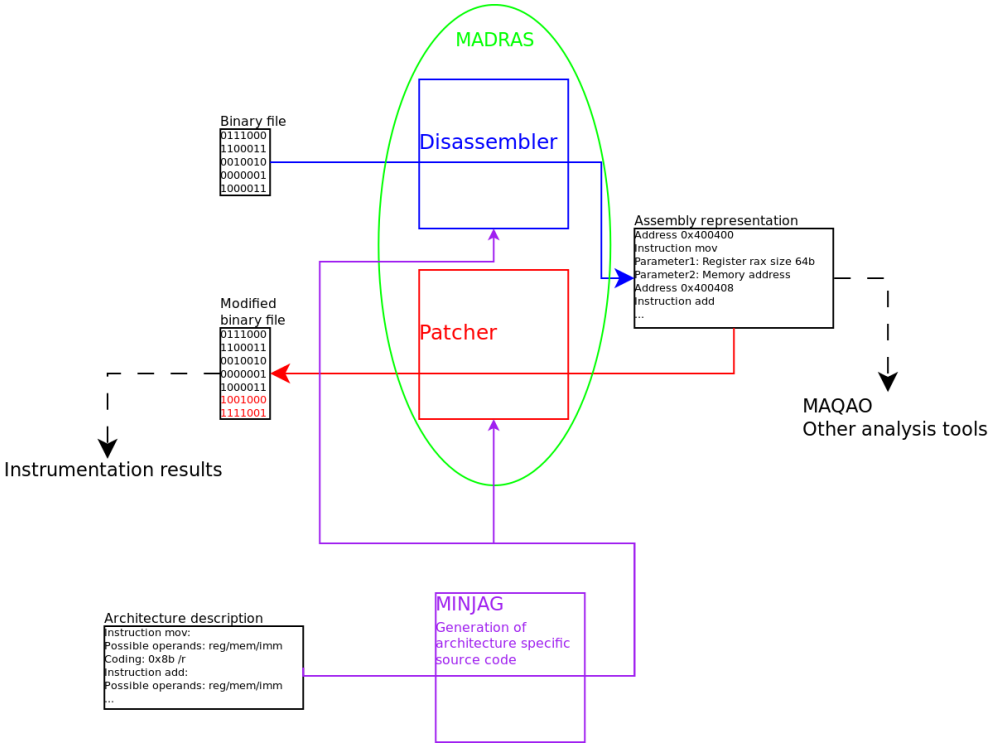

Figure 7.1: Global description of the MADRAS and MINJAG interconnection and of MADRAS operating mode.

7.2 Future works

We will present here the perspectives offered by the various contributions of this thesis, either as extensions of tools functionalities or as subjects of further research.

7.2.1 Implementing extensions

One branch of work consists in the implementation of the extensions described in Chapter 6.

Further extensions involve facilitating the implementation of a disassembler from the files generated by Minjag, taking the form of a toolkit of low-level primitives that could be used when implementing the macros representing the semantic actions of the grammar.

Another subject involves extending the binary parser to other formats such as WindowsPE or Mach-O, allowing to disassemble and patch files using these formats for the supported architectures. This implies adapting the patcher to handle behaviours varying depending on the ABI, such as parameters passing or invocation of dynamic functions.

Other extensions involve the use of the Madras disassembler by other tools using its ability to return additional information about instructions. An example of this would be using this information useful to analysis tools focused on power consumption instead of performance analysis.

7.2.2 Future research

An extension involving original research aims at simplifying the generation of the grammar representing the ISA, which is currently built from the list of instructions for the architecture through a simplified script. Since this list of instructions does not obey a specific format, this script has to be adapted for every architecture. A useful evolution would then be the definition of a unified format for this instruction list, allowing to use the same script to be used for all architectures, while remaining as close as possible to the format used by the architectures developer manuals, turning in effect the grammar into an intermediate representation of architectures with the instruction list becoming the entry point. The full specification of a format able to describe all types of architectures and of the associated algorithms for translating it into grammar format presents interesting challenges.

Since the Madras disassembler aims at providing the most accurate representation of the binary file, another interesting lead would be the retrieval of information from the data sections, especially in the identification of individual variables. The bytes in the data sections of the files are not separated depending on the variable which they represent and, while pattern matching could be used to identify specific variable types such as arrays or strings, there is no unified method to identify all variables from their values. It is however possible to extrapolate their location in the data sections from some of the information present in the file, such as labels or memory operands referencing addresses in this section.

Another point of interest is tied to the increase of the disassembler accuracy. Some of those works were covered in 6.1.2. Further algorithms and heuristics would allow the disassembler to handle obfuscated and even self-rewriting codes, which is challenging if it has to induce a minimal overhead. This is also interesting when using the disassembler outside the context of performance analysis, such as protection against potentially dangerous files. The knowledge gained from this can then be used to increase the security of executables by reducing their vulnerability to reverse engineering techniques.

Appendix A

Minjag developer documentation

Here is a detailed descriptions of the implementation of the encoder and decoder generation principles as Minjag.

A.1 Description

Minjag is a set of scripts and executables allowing to generate the source code for decoding and encoding the instruction set of a given architecture from a list of instructions and their coding rules.

We will focus here on the **minjag** executable, which allows to generate the code from a grammar. The scripts generating this grammar mainly perform the conversion from the human-readable instruction list to the slightly less user friendly grammar format. They also enforce grammar coherence, for instance ensuring that instructions with an identical opcode but a different mandatory prefix are added to the productions of different symbols.

The **minjag** executable (referred to from now on as **minjag**) performs the following functions:

- Parse the grammar file representing an architecture.
- Build the associated Finite State Machine (FSM).
- Generate the corresponding source files.

The source for **minjag** is entirely written using the C language and contains 11,000 lines of code. Its total implementation time is estimated to 12 months as of the writing of this dissertation.

A.2 Using minjag

To run **minjag**, use the command:

```
$ minjag [ -o <directory> OPTIONS] <grammarfile>
```

where <*grammarfile*> is the path to the grammar file to use and *OPTIONS* can take the following values:

- `-o`, `--output-directory` <*directory*>: Sets the directory where the files will be written. If not set, the directory containing <*grammarfile*> will be used.
- `-i`, `--info`: Activates the info mode (disabled by default). The FSM states will be copied to file <*grammarfile*>`.states` in the logs directory.
- `-t`, `--test`: Activates the test mode (disabled by default). The source files will not be generated, only the FSM states.

- `-L`, `--logs-directory` <*directory*>: Sets the directory where the log files will be written. If not set, the directory containing <*grammarfile*> will be used.
- `--insnlist`: Prints the list of instructions present in the grammar file into <*grammar*>`.inlist`.
- `-h`, `--help`: Displays the help.

The following options allow to generate some of the files only or to exclude files from generation. The `--only` flags are cumulative and prevent the generation of other files.

- `--arch-only`: Only generates the architecture definition file.
- `--sym-only`: Only generates the grammar symbols definition file.
- `--fct-only`: Only generates the files relative to semantic actions functions.
- `--final-only`: Only generates the files relative to the post-parsing functions.
- `--fsm-only`: Only generates the FSM definition files.
- `--revfct-only`: Only generates the files relative to the encoding functions.
- `--asm-only`: Only generates the files relative to the assembly functions.
- `--no-arch`: Does not generate the architecture definition file.
- `--no-sym`: Does not generate the grammar symbols definition file.
- `--no-fct`: Does not generate the files relative to semantic actions functions.
- `--no-final`: Does not generate the files relative to post-parsing functions.
- `--no-fsm`: Does not generate the FSM definition files. This also disables the generation of the FSM.
- `--no-revfct`: Does not generate the files relative to the encoding functions.
- `--no-asm`: Does not generate files relative to assembly functions.

A.3 Grammar format

We will describe here the format of the grammar accepted by MINJAG.

A.3.1 Outline

The grammar must contain the following sections (in this order):

- Begin code (optional).
- List of tokens, definition, and architecture specification.
- List of grammar symbols with their respective expansions.
- End code (optional).

The code sections are enclosed by the "`%{`" and "`%}`" strings. The list of variables must be enclosed by "`%%`" strings. The character '`#`' at the beginning of a line sets the whole line as a comment and prevents it from being parsed.

A.3.2 Begin and end code sections

Those two sections will be copied verbatim at the beginning and end of the generated file header describing the architecture (<*archname*>_*arch.h*). They can be left blank if not needed, otherwise they must contain standard valid C code. No check is made during the parsing of the grammar, so any error in those sections will be detected only when compiling the generated files.

A.3.3 Declarations section

This contains the declaration of tokens, definitions, and architecture specification.

A.3.3.1 Tokens

A token is a terminal symbol in the grammar with a fixed length but undefined value. A token declaration must specify its length and endianness. There are 3 different possible endiannesses for a token:

- Lower bit first: The leftmost bit is the lowest order bit. Example: `00101000 10101100` corresponds to the hexadecimal value of `35 12`.
- Lower byte first: The leftmost byte is the lowest order byte. Inside a byte, the leftmost bit is the highest order bit. Example: `00101000 10101100` corresponds to the hexadecimal value of `AC 28`.
- Bigger bit first: The leftmost bit is the higher order bit. Example: `00101000 10101100` corresponds to the hexadecimal value of `28 AC`.

A token is declared as follow:

```
%token <S,E> mytoken
```

Where `mytoken` is the token name, beginning with a letter or '`_`' and containing letters, numbers, '`_`' or '`-`', `S` its length in bits, and `E` its endianness, coded as '`l`' for lower bit first, '`L`' for lower byte first, and '`b`' or '`B`' for bigger bit first.

A.3.3.2 Definitions

A definition is a symbol that actually stands for one or more different symbols. Each line containing an occurrence of this definition in the grammar will be replaced during parsing by as many lines as the number of symbols the definition stands for, each containing a different symbol. It can be used to reduce the size of a grammar when for instance it contains multiple lines differing from each other by a symbol.

A definition must be declared as follow:

```
%define mydefine var1 | var2 | var3 ;
```

where `mydefine` is the definition name, and `var1`, `var2` and `var3` the names of the symbols it stands for. If a grammar using the definition above, the following line:

```
myvar:001 mydefine 11;
```

will be parsed as:

```
myvar:001 var1 11 | 001 var2 11 | 001 var3 11 ;
```

Definitions can contain either terminal symbols with a fixed value or nonterminal symbols, but not a mix of both. There are no limitations on the number of symbols that can be contained in a definition, but a greater number of symbols inside a definition can increase parsing time.

A.3.3.3 Architecture specifications

The grammar can contain a number of definitions for the description of the architecture. Each of these definitions is identified by a keyword at beginning of a new line and prefixed with the character '`%`'. Below is a description of the current accepted definitions:

- `archname:` This definition is mandatory. It must be followed by a string describing the architecture. This string will be used as a suffix for every generated file and as an identifier for the architecture in the code. Example: `%archname x86_64`

- `regtype:` This definition specifies a type of register in the architecture. It obeys the following format:

  ```
  %regtype typename regfamily regsize regnames;
  ```

  with:

  - `typename`: Identifier for this type of register. It must be a distinct string for each register types.
  - `regfamily`: The name of the family to which the register type belong. A family name can be shared between multiple register types. This is used to identify registers representing parts of other registers.
  - `regsize`: Keyword identifying the size of the register.
  - `regnames`: Comma-separated list of register names of the given type. It is advised to have those names appear in the same order as the index representing them in binary format, leaving a space if a given index has no associated register. A register name can be followed by flags beginning with a '/' character for specifying additional information. Currently, the only supported flags are `A` and `R`, respectively indicating that the register is used to pass parameters to routines or contain return parameters from routines by the ABI.

- `insn:` This definition specifies additional information about instructions in the architecture. It obeys the following format:

  ```
  %insn information value
  ```

  Currently the following keywords for `information` are recognised, any other keyword being ignored:

  - `maxlen`: Maximum length in bits of an instruction
  - `minlen`: Minimum length in bits of an instruction

A.3.4 Symbols definitions

A symbol name must begin with a letter or the '_' character, and can contain letters, numbers, '_' or '-'. A grammar symbol must be defined as follows:

```
Symbol: A B C #[ Macro1 ]#
| D E #[ Macro2 ]#
| ... ;
```

where `A`, `B`, `C`, `D` and `E` are either symbols defined elsewhere in the grammar, tokens (defined in the A.3.3.1 declaration section) or fields of bits (succession of 0 and 1 characters). `Macro1` and `Macro2` are macros representing the semantic action associated to the reduction of the symbol (see A.3.4.2).

For clarity, it is advised to separate the different possible expansions of a symbol with carriage returns, as in the example above, but this is not mandatory.

A.3.4.1 Restriction and constraints

The symbol names must be unique. A production can not contain multiple occurrences of the same symbol. Every symbol used in a production must be declared in the grammar, either as token or another nonterminal symbol. An error will be raised if a symbol is used undeclared. The start symbol, from the expansion of which every other symbol may be eventually reached, must be named "`template`".

Due to some internal operations made by the parser when building the finite state machine, no symbol must have the name "`Axiom`". The grammar supports recursive declaration, where a symbol appears in one of its productions. The current version of the parser does not handle well the grammars where some expansions of different symbols can be identical to one another until a symbol is reached.

A.3.4.2 Semantic actions

Each production of a symbol must have an associated semantic action enclosed with "`#[`" and "`]#`". Semantic actions are represented by macro-like statements: a keyword, followed by any number of arguments enclosed in parentheses.

Semantic actions macros must obey the following rules:

- Only one macro must appear for a given semantic action.
- Macros must have either of the following prefixes: `BIN_`, `INSN_` or `OPRN_`. Statements without one of these prefixes will be ignored
- Macro parameters can be either constant values (strings), references to a symbol in the associated production identified with the symbol '`$`' followed by the order of appearance of the symbol in the expansion starting at 1, or another macro the name of which is not required to have one of the mandatory prefixes.

Instruction macros The semantic action associated to the reduction of a symbol representing an assembly instruction must be a macro with the prefix `INSN_`. Such a macro must have the following prototype:

```
INSN\_xxxx(mnemonic, \textcolor{keyword}{set}, family,
      annotation, operand1, operand2, ...)
```

where:

- `mnemonic` is the mnemonic of the instruction, written in upper case. The current version does not support mnemonics containing a space character (this can happen if a mnemonic has a mandatory prefix). A possible workaround is to use an underscore ('_') to replace the space.
- `subset` is a string representing the instruction set to which the instruction belongs, if more than one are available for the given architecture. It is advised to prefix it with `ISET_`.
- `family` is the family (`mov`, `add`, ...) to which the instruction belongs. It is advised to prefix it with `F_`.
- `annotate` is the default annotation associated to this instruction, as defined in the base MAQAO definitions for representing instructions.
- `operand1`, `operand2`, etc are the operands of the instruction.
- Other parameters can be added after the list of operands.

Instruction operand macros The operand of an instruction must be represented by a macro with a `OPRN_` prefix. They have no restrictions on the number or type of arguments.

A.4 Source file generation

We will now describe the files generated by MINJAG, which include those allowing to execute the parser and encoder, as well as those containing the description of the architecture defined in the grammar.

A.4.1 Outline

All generated files are prefixed with the name of the architecture, as defined in the grammar. If no architecture was defined, the files are prefixed with the base name of the grammar file, and the architecture description files are not generated. The following files are created:

- ***archname*_arch.h** / ***archname*_arch.c:** Those files contain the definition of the structure representing the architecture, as well as macros for the identifiers of the names and types of the registers for this architecture. Their generation is described in A.4.2.
- ***archname*_assembler.h** / ***archname*_assembler.c:** Those files contain the declaration of the function allowing to encode an instruction. Their content is fixed and only depends on the architecture name.
- ***archname*_fct.h** / ***archname*_fct.c:** Those files contain the declaration of functions encapsulating the macros corresponding to the semantic actions of the grammar. Their generation is described in A.4.3.1.
- ***archname*_finalfct.h** / ***archname*_finalfct.c:** Those files contain the declaration of functions encapsulating the macros corresponding to the post-parsing actions. Their generation is described in A.4.3.2.

- ***archname*_fsm.h / *archname*_fsm.c:** Those files contain the declaration of all the variables used to run the FSM for this grammar. Their generation is described in A.4.4.
- ***archname*_macrofinal.def:** This file contains the list of all macros that can be run after a successful parsing (they are used in *archname*_finalfct.c). It is not intended to be a source file, but to allow the generation of one. Its generation is described in A.4.5.
- ***archname*_macros.def:** This file contains the list of all macros corresponding to a semantic action (they are used in *archname*_fct.c). It is not intended to be a source file, but to allow the generation of one. Its generation is described in A.4.6.
- ***archname*_revfct.h / *archname*_revfct.c:** Those files contain the declaration of functions encapsulating the macros corresponding to reverse semantic actions. Their generation is described in A.4.3.3.
- ***archname*_revmacros.def:** This file contains the list of all macros used for performing the encoding (they are used in *archname*_revfct.c). It is not intended to be a source file, but to allow the generation of one. Its generation is described in A.4.7.
- ***archname*_sym.h:** This header contains the definition of identifiers for the symbols of the grammar. Their generation is described in A.4.9.

A.4.2 Generation of the architecture definition

These files contain the declaration of the structure describing the given architecture. The following notable fields of the structure are set:

- `name`: Architecture name, retrieved from the `archname` field in the grammar.
- `mnemonic`: Array of mnemonics names, retrieved from the `INSN_OPCODE` macros in the grammar.
- `families`: Array of family names, retrieved from the `INSN_OPCODE` macros in the grammar.
- `size_mnemonics`: Number of mnemonics.
- `reg_names`: Bi-dimensional array of register names by types, retrieved from the `regtype` fields in the grammar.
- `regs`: Bi-dimensional array of register instances, built from the names and types retrieved from the `regtype` fields in the grammar.
- `reg_sizes`: Array of different register sizes, retrieved from the `regtype` fields in the grammar.
- `reg_families`: Array of different register families, retrieved from the `regtype` fields in the grammar.
- `noprnd_min`: Array of minimal number of operands per mnemonic, computed amongst all the occurrences of `INSN_OPCODE` with a given mnemonic parameter in the grammar.

- `noprnd_max:` Array of maximal number of operands per mnemonic, computed amongst all the occurrences of `INSN_OPCODE` with a given mnemonic parameter in the grammar.
- `dflt_anno:` Array of default annotate flags per mnemonic, calculated amongst all the occurrences of `INSN_OPCODE` with a given mnemonic parameter in the grammar.
- `nb_type_registers:` Number of different register types, deduced from the `regtype` fields.
- `nb_names_registers:` Maximum number of register names per types, deduced from the `regtype` fields.
- `return_regs:` Array containing all registers which can be used to return a value at the end of a function.
- `nb_return_regs:` Size of `return_regs`.

The header also contains the following macros:

- Each mnemonic is defined as a macro, beginning with `I_` and followed by the name of the mnemonic. This macro maps to a unique number starting at 0. This number is the index of the mnemonic name in the `mnemonic` array in the structure representing the architecture.
- Each instruction subset is defined as a macro, mapping to a unique number starting at 1. This number is used in the `insn_set` field of the structures representing disassembled instructions.
- Each register type is defined as a macro, mapping to a unique number starting at 0. This number is the index of the line containing the registers of this register type in the `reg_names` and `regs` arrays in the structure representing the architecture.
- Each register index is defined as a macro, mapping to a unique number starting at 0. This number is the index of the column containing the registers with this index in the `reg_names` and `regs` arrays in the structure representing the architecture. This index is determined by the order into which registers appear for a given type in the `regtype` fields in the grammar.
- Each register family is defined as a macro, mapping to a unique number starting at 0. This number is used in the `reg_family` array from the structure representing the architecture.

A.4.3 Handling files with macro definitions

Files defining functions encapsulating macro invocations share the following characteristics:

- They contain functions whose body is the invocation of the macro
- For clarity and to ease up comparison for non-regression tests, those functions are written in the order of increasing identifier. This identifier increases in the lexicographical order of the macros used by the function.

- To avoid compilation warnings, a function is declared only if the macro it contains is defined. Thus, all function declarations using a given macro are enclosed within a `#ifdef` statement testing the definition of the macro.

A.4.3.1 Generation of the semantic action functions

Those files contain a series of functions whose prefix is defined by the architecture name followed by the value of the `MACROFUNCNAME` macro, and suffixed by a unique identifier starting at 0.

Each of those function takes as parameter an array of pointers. Its body consists in the invocation of a macro, passing as parameters the relevant cells in the array of variables, which are identified by the identifier of the associated grammar symbol, as defined in the symbol list (see A.4.9).

The return type of those functions is `void` if the associated macro corresponds to a semantic action; in that case, the first parameter of the macro is the cell in the array where the result of the semantic action will be written. Otherwise, if the macro is invoked by another macro, the function returns a pointer, which is also retrieved from the first parameter of the macro.

A.4.3.2 Generation of the post-parsing functions

The name of functions for end-parsing actions are prefixed by the architecture name followed by the value of the `FINALFUNCNAME` macro, and suffixed by a unique identifier starting at 0.

Each of those functions takes as parameter a pointer to the object generated after the successful parsing of a word, and another pointer used for passing additional information if needed. Their body consists in the invocation of a macro, passing the parameters of the function to it.

A.4.3.3 Generation of the matching functions for the encoder

The matching functions for the encoder are those that will be responsible of checking whether a given input matches with the data returned by a semantic action.

They are printed by encapsulating a semantic action macro, followed by all macros it invokes, which we will call sub macros in this context. The naming conventions for the printed macros are identical to those described in A.4.6. An additional parameter, `FOUND`, is added to the printed macros. It is intended to be updated by the macro and set to `TRUE` if the entry is found matching. When the macro is invoked in the function, its `FOUND` parameter will be fed with a local variable to the function, `found`, which is initialised to `TRUE` and is also the return value of the function.

Matching functions are printed to take as parameters an array of pointers and another of 64-bits integers, which will be used respectively for the variable and token parameters of the macros. The parameters will be stored at the index corresponding to the identifier of the symbol.

The functions are printed so that the invocation of the sub macros is conditioned to the local `found` variable being still `TRUE`.

A.4.4 Generation of the FSM structures

The FSM is described as structures representing the states and transitions, defined in the header file. The associated source file only contains inclusion of an architecture independent file defining the function returning the array of state.

A.4.4.1 Transitions

There is a distinction between structures representing transitions over bit fields and those over variables.

Transitions over bit fields Transitions values of a given shift state are broken down into sub values, as described in Section 3.3.4.3.

Transitions on infinite length (corresponding to a transition over a null value) are not included when breaking transitions into sub values. If a shift state contains such a transition, its next state will be added in the definition of the state.

A sub value is identified by its value and its mask, which specifies which bits in the value are fixed (they are set to 1 in the mask). Sub values of identical length, index, and preceding sub values are grouped into lists and tables.

A list contains all the sub values which could take a given value because of the masks, in the same order the corresponding transitions were ordered in the state.

A table of sub values is an array of lists whose size is the total number of values possible with the length of the sub values it contains (2^{length}). Each cell of a table contains the list of sub values which could take as value the index in the table. It is important to note that sub values containing unfixed bits (mask is smaller than $2^{length} - 1$) will appear in multiple cells of the array.

If a table should contain a single sub value, it is a 1 cell array containing the value, and will be identified with the `SUBTBL_SINGLEVALUE` flag. If this single sub value has a mask set to 0 (no useful bits), the table will be identified with the `SUBTBL_ALWAYSOK` flag instead. In this case, the corresponding sub value may be larger than 8 bits if the following sub values also have a mask set to 0. This is the only case where a sub value will be larger than 1 byte, and it will never be used for comparison by the FSM. This exception is used for optimisation when dealing with transitions larger than 1 byte and containing only unfixed bits.

Finally, a sub value also contains a pointer to the table of the sub values that could follow this sub value. If this sub value was the last in a transition, this table will be set to `NULL` and contain the transition of the next state instead.

Transitions over variables Variable names (when reducing symbols) are defined as numbers in order to reduce size and comparison times. For a given state, transitions over reduced symbols are stored in an array whose size is the number of variable symbols from the grammar. The array contains either the identifier of the next state or the `STATE_NONE` identifier when the state does not contain a transition for this symbol.

A.4.4.2 States

The states containing only one item and no transition are *reduction* states. States containing at least one item whose step is at the end of production and at least one transition are *shift/reduce* states. All other states are *transition* states. An array

containing all the states is also printed, ordered by their identifier, with the first state in the FSM being at index 0.

A.4.5 Generation of the list of post-parsing macros

A post-parsing macro is generated for each semantic action corresponding to the reduction of an instruction, identified by containing a macro prefixed with `INSN_` (cf. A.3.4.2). Its name is built from the instruction mnemonic, concatenated with its operands type and size, as represented in the following example for an instruction with two operands `Op1` and `Op2`:

```
IN\_<Mnemonic>\_OP\_<Op1 type>\_<Op1size>\_OP\_<Op2 type>\_<Op2size>
```

A.4.6 Generation of the list of semantic action macros

All macros used in semantic actions are printed in a list, as they should appear in the header defining them.

The name of a macro is built from its name as it appears in the grammar, suffixed by a unique identifier of 4 characters. These characters are hexadecimal numbers (0 to 9 and A to F) and represent the number of parameters of the macro with a given type. Those types are respectively:

- Another macro.
- A constant (another macro with no parameters).
- A grammar nonterminal symbol (actually the value returned by the semantic action triggered by the reduction of this variable).
- A grammar token (the value of the token as encoded in binary).

The macro parameters are, in the order by which they appear:

- A parameter named `OUT`.
- The macro parameters, named `MCRx`, x starting at 0.
- The constant parameters, named `CSTx`, x starting at 0.
- The variable parameters, named `VARx`, x starting at 0.
- The token parameters, named `TOKx`, x starting at 0.

The macro is followed by comments (formatted as C comments), detailing the possible values for the macro parameters, as well as the grammar symbols for which this macro is a semantic action.

A.4.7 Generation of the list of encoding macros

The list of encoding macros is built identically to the list of semantic action macros described in A.4.6, with the exception that an additional parameter, `FOUND`, is added first to the macro parameters.

A.4.8 Generation of the encoding structures

The encoder is built from structures representing reverse semantic actions and symbol encoding rules. A reverse semantic action contains the pointer to the corresponding matching function, printed in A.4.3.3, and a description of the elements of its binary expression, including the encoding rules for the nonterminal symbols it contains. A symbol encoding rule is a list of reverse semantic actions corresponding to all the possible expansions of the symbol. We use the same structures and functions to print the upward actions and encoding rules, while they represent the completion of a partially encoded input that was not encoded into the start symbol.

The generation of reverse action structures is done through the following steps:

1. Build a list of all symbols that have at least one production whose semantic action either is an instruction or recursively contain a symbol that does.

2. This list is built so that the path of recursion between the top-most symbol and a symbol having a production whose semantic action is an instruction is the shortest possible.

3. For each symbol, the list contains the semantic action macro of the production the symbol belongs to.

The generation of a reverse encoding rules structure for a symbol is done by printing an array containing pointers to the reverse semantic action structures corresponding to each possible expansion of the symbol. The generation of an upward reverse encoding rules structure for a symbol is done by printing an array containing pointers to the reverse semantic action structures for all symbols between the symbol and the top most symbol. In both cases, the reverse actions are printed in order of increasing size of the associated binary expression.

Finally, structures representing the encoding rules for instructions are printed. They are handled as the encoding rules of a symbol and contain pointers to the structures representing the reverse of all semantic actions of type `INSN_` containing the given instruction as mnemonic name.

A.4.9 Generation of the symbols list

The symbols list is written as an enumeration, containing the names of all grammar symbols. Symbols representing nonterminal are written first. The symbols are printed in lexicographical order, with the following exceptions:

- The first element of the enumeration has no associated symbol.
- The second element of the enumeration is the actual start symbol `Axiom`.
- The third element of the enumeration is `template` the grammar start symbol.

A.5 Implementing a new architecture

Implementing a new architecture with MINJAG is done through the following steps:

1. Create the grammar from the instruction list.

2. Building headers from the .def files:

- Writing disassembly macros for semantic actions.
- Writing assembly macros for reverse semantic actions.
- (Optional) Writing final macros

The grammar header must obey the grammar syntax described in Section A.3. It must contain the variables relative to the architecture description as specified in A.3.3.3. In addition, all tokens, defines, and non terminal symbols used in the instruction list must be declared there if needed.

A.5.1 Building the headers from the .def files

Headers have to be built from the corresponding $*.def$ files. The main modification is to add the `#define` keyword before the macro names. It is advised to keep the header file synchronised with the corresponding $*.def$ file so that the comments are updated, making implementation easier. To avoid compilation warnings, it is advised to comment or remove an unused macro from the header instead of leaving it defined with no value, as functions using them will be excluded from compilation by a preprocessor directive if the macro is not defined.

A.5.1.1 Writing macros for semantic actions (disassembly)

The semantic actions macros accept 4 types of parameters : macro, constants, variables and tokens. A comment after each macro lists the possible values for each parameter. All macros have at least one parameter, `OUT`. The parameters are described in more detail in A.4.6. When the parser is used for disassembling, those macros are expected to allocate the representation of an instruction. When writing the code for disassembly macros, the following points have to be taken into account:

- Each macro represents the body of a different C function, except for macros invoked by another macro.
- The `OUT` parameter of the macro represents its return value. It is expected to be a C pointer (type `void*`).
- Parameters of type macro (`MCRx`) are actually the direct invocation of the corresponding macro. The code should handle such a parameter as if it was the invoked macro itself.
- Parameters of type constant (`CSTx`) will be used as such by the macro. They have to be defined somewhere, or may be concatenated to another string to create the name of a different identifier.
- Parameters of type variable (`VARx`) are C pointers (type `void*`). They are the trickiest to handle, as they will contain the "return" value (`OUT` parameter) of the macro that was invoked as semantic action of the reduction of the corresponding grammar nonterminal symbol (variable). As such, it is imperative to ensure that the same type of data will be returned by all semantic actions possibly invoked when reducing the grammar variables possibly used as parameter of a given macro.
- Parameters of type token (`TOKx`) are `paramcoding_t` structures and can be accessed with the appropriate functions from the FSA sources.

A.5.1.2 Writing macros for reverse semantic actions (assembly)

The reverse semantic action macros have the same type of parameters as the semantic actions macros, with the `OUT` parameter being replaced by `INPUT` and `FOUND` (see A.4.7). All parameters of the macros except the `INPUT` and constant parameters are return parameters and must be updated by the code of the macro. When used for assembling, those macros are expected to check if a given input matches a pattern (which is what the associated semantic action returned), and build the coding for the symbols present in the associated grammar production. The following points have to be taken into account when writing the code for assembly macros:

- Each macro will be invoked in the body of a different C function, along with the macros it invokes.
- The `FOUND` parameter must be set by the macro to `TRUE` if the match was successful or `FALSE` otherwise.
- The `INPUT` parameter of the macro represents the input data to match. It is expected to be a C pointer (type `void*`).
- Parameters of type macro (`MCRx`) will be used as `INPUT` parameters in the corresponding macro invoked by the current macro. The constraints about the type of data they contain are the same as those described in A.5.1.1.
- Parameters of type constant (`CSTx`) will be used as such by the macro, and are handled like the constant parameters of semantic actions macros as described in A.5.1.1.
- Parameters of type variable (`VARx`) will be used as input by encoding rules of the corresponding grammar symbol and thus will be used as `INPUT` parameters by all possible reverse semantic action macros for this symbol. The constraints on the data type are the same as those described in A.5.1.1.
- Parameters of type token (`TOKx`) are 64 bits integers to set at the correct value.

A.5.1.3 Writing final macros

Those macros will form the body of functions invoked after a successful parsing. What they do is entirely left up to the needs of the application using the disassembler; they are not required by the parser and can be all safely commented out.

Each of those macros accept two C pointer parameters (`void*`). The first is the structure returned by a successful parsing. The second parameter is user defined and can be used to pass additional information.

Appendix B

Madras API

Operating the Madras disassembler and patcher and possible through an API allowing to access their functionalities. We will describe here this API.

The **libmadras** API allows to pilot the disassembly and patching of a binary file, and some examination of its contents. Patching operations cover inserting function calls or assembly instruction, delete or modify existing assembly instructions, and modify the list of libraries needed by a file.

libmadras is available on architectures supported by the patcher, disassembler, and binary parser. In the current version, only the Intel 64 and Xeon Phi coprocessor architectures under the ELF binary format are supported.

The source for the whole Madras disassembler, patcher and API is entirely written using the C language and contains 30,000 lines of code for its architecture independent core, plus 8,000 lines for each implemented architecture. Its total implementation time is estimated to 24 months as of the writing of this dissertation. As a part of the Maqao framework, Madras is intended to be released as Open Source along with it.

B.1 libmadras structures

The main structure used by **libmadras** is `elfdis_t`. It is used to store a disassembled file, the modification requests and logging settings. An `elfdis_t` structure is returned upon disassembling a new file. It must be freed using `madras_terminate`.

All subsequent **libmadras** functions need a pointer to the current `elfdis_t` structure. It is not advised to access directly the members of an `elfdis_t` structure.

B.2 Disassembling functions

The main function for disassembly is `madras_disass_file`. It needs a valid file name and will return a pointer to a new `elfdis_t` structure containing the disassembled file. None of the structural analysis functions from the core components of Maqao are performed on the disassembled file. Only the information from the disassembly are available.

The API allows to retrieve information on the disassembled instructions. This part of the API is subject to change or become obsolete in the following releases of Madras and will not be detailed here.

B.3 Patching functions

The functions allows to patch a file and save the results to a binary file. This operation is defined as a patching session by the Madras API. A patching session contains the following steps:

1. Patcher initialisation: this operation sets the variables needed for the session.
2. Patch requests: registering a series of patching requests. No patching is actually performed at this stage.
3. Patch commit: this step must always be the last one in a session. It performs all the patch requests and writes the result to another file.

In the current version, only one patching session can be performed on a disassembled file. It is also not possible to produce multiple patched files from a single disassembled file. It is advised to terminate the `elfdis_t` structure after a patching session has been completed. If other patching operations are needed, a new structure must be created by disassembling the file once again.

Because patching operations are all performed during the commit, all potential patching errors will be reported at this point only.

B.3.1 Patcher initialisation

Patcher initialisation must occur before applying any patch requests. It is performed through the function `madras_modifs_init`.

This function also allows to choose the method used for saving the stack when performing code insertions (cf. 5.3.3). **libmadras** offers three different ways of dealing with the stack:

- **Keep:** The stack pointer is not modified in any way. This mode is advised for insertions into codes where the stack is not used, or for patching sessions which will not involve code insertions. It can lead to crashes if used while performing insertions into codes where the stack is used.
- **Move:** The stack is moved to an area of memory that has been added to the file. This ensures that the stack used by the inserted code does not overlap with existing memory areas. However, it is not supported in multi-threaded mode, as the same area will be used for all threads. It can also lead to an overflow if the inserted functions make a heavy use of the stack, as the size reserved for the moved stack is limited (currently to 1 Mb).
- **Shift:** The stack is shift upward by a number of user-defined bytes, allowing to skip the area used by the current stack while remaining thread-safe. As there is currently no way to know the size of the current stack, this method may still lead to errors in the patched file if the shift is not important enough. It has been observed that a shift of 512 bytes is enough for all tested codes. This mode is advised for most patching sessions involving function call insertions.

B.3.2 Data modification

The MADRAS patcher allows to insert a new global variable to the file, using the `madras_globalvar_new` function. The inserted global variable can be accessed from any place of the executable; as its address is fixed, it is not protected against multi-thread access. An inserted global variable will be filled with zeroes by default. It is possible to specify a value to initialise it. This is the standard behaviour when inserting strings. It is possible to use either the address or the value of global variables as parameters to inserted functions (see B.3.4.1) or operands to inserted instructions (see B.3.4.2).

B.3.3 Libraries modification

It is possible to modify the list of dynamic libraries needed by an executable. The name of a needed library can be changed using the `madras_extlib_rename` function. This can be useful for instance when a needed dynamic library has been itself patched and saved under a different name.

The dynamic library where is defined a function whose call is inserted (see B.3.4.1) is automatically added to the list of needed libraries. The function `madras_extlib_add` also allows to do that but it is not needed for dynamic libraries. This function is intended to be used to insert static libraries to a file.

B.3.4 Code modification

The MADRAS patcher allows to insert calls to functions present in a file or defined in a dynamic library, insert assembly code, and delete or modify instructions.

All modifications are performed in the order of ascending addresses. In case of multiple modifications at the same address, the modifications will be performed in the order into which their respective requests were made. It is also possible to choose whether an insertion must be performed before or after the instruction present at the given address.

The behaviour of the MADRAS patcher is undefined if an instruction is requested to be deleted as well as modified. It is also not possible to modify an instruction that is being added by the patcher.

All functions allowing to modify the code return a pointer to a `modif_t` structure which represents the modification request and allows to add further options to it.

All modifications performing code insertions can be set at a null address. Those modifications are intended to be used either when linked to another insertion (see B.3.4.6) or as an "else" statement to an insertion with conditions (see B.3.4.7). An error will be raised when committing modifications requests if some modifications set at a null address are not found linked to another modification.

B.3.4.1 Inserting a function call

It is possible to insert a call to a function either defined in a dynamic or static library file or already present in the executable, using the `madras_fctcall_new` function. If no library name is provided, the MADRAS patcher will assume the function is present in the executable, and will display an error if not found there. Otherwise, the MADRAS patcher will use the library name to identify its type (static or dynamic) and choose the type of insertion to perform. If the library where the function is defined is static, it is necessary to also add all libraries defining the symbols used in the library using `madras_extlib_add`. The patcher will return an error if some symbols remain undefined.

When inserting a function call, the MADRAS patcher will automatically surround the inserted call with the appropriate assembly instructions for saving and restoring all registers, as well as aligning the stack pointer and saving the current stack depending on the chosen method (as described in B.3.1). The `madras_fctcall_new_nowrap` function allows to insert only the function call without any such surrounding instructions. This function can be useful to reduce the overhead of saving and restoring the context provided the user takes care of it (using for instance `madras_insnlist_add` to save and restore only a subset of registers) or

if the inserted function can not change the execution context or on the contrary if it is the expected behaviour. [1]

Both functions return a pointer to a `modif_t` structure, which contains all the details on an insertion request. The pointer can be used to add parameters and return value to the function call by invoking the `madras_fctcall_add*` functions.

An inserted function call can accept up to 6 parameters (additional parameters will be ignored). Parameters are added to an existing function insertion via different API functions depending on the parameter type. Those functions are:

- `madras_fctcall_addparam_fromstr`, for adding a valid assembly operand from its string representation,
- `madras_fctcall_addparam_frominsn`, for adding an operand from another instruction,
- `madras_fctcall_addparam_imm`, for adding an immediate (integer) operand,
- `madras_fctcall_addparam_fromglobvar`, for adding a global variable added by the MADRAS patcher.

It is not possible to use an existing global variable as parameter.

In the current version, all parameters are passed as 64 bits integers, so an inserted function can use either 64 bits integers or pointers as parameters.

An inserted function call can be set to return a value. This is done with function `madras_fctcall_addreturnval`. The return value must be a global variable added by the MADRAS patcher. Like the parameters, the return value is a 64 bits integer, so it can be either treated as an integer or a pointer.

B.3.4.2 Inserting assembly instructions

It is possible to add assembly instructions either in string format using the function `madras_insnlist_add` or as a queue of the structures used by MAQAO to represent an assembly instruction (`insn_t`) using the function `madras_add_insns`. The instructions will be added at the given address without any check nor addition of wrapping instructions.

Instructions present in inserted lists can reference a global variable added by the MADRAS patcher. An instruction referencing such a variable must use a memory operand using the instruction pointer as base and a null displacement (`0(%RIP)`). An array of pointers to the `globvar_t` structures describing the referenced global variables must be passed as parameter to the function, with the pointers appearing in the same order in the array as they appear in the instruction list. It is possible for a pointer to appear more than once if the corresponding global variable is to be referenced multiple times.

Instructions added using `madras_insnlist_add` must be passed as strings, all written in upper case, and separated by carriage return "\n" characters. Such lists can use labels, identified as followed by a colon character ":" and beginning a line, to reference branch destinations. It is not possible to reference a label from the patched file in an inserted list.

[1] `madras_fctcall_new_nowrap` is now deprecated, as the same result can be obtained using `madras_fctcall_new` then `madras_modif_addopt` with the `PATCHOPT_FCTCALL_NOWRAP` flag.

B.3.4.3 Inserting a branch instruction

It is possible to insert an unconditional branch instruction using the function `madras_branch_insert`. The branches added using this function can point either to an existing instruction in the file, referenced by its address, or another modification request (only insertions modification requests are currently supported). In the latter case, the branch will point to the first instruction inserted by the modification.

It is also possible to choose whether the branch must be updated if a code insertion is performed before its destination. The standard behaviour would be to update the branch to point to the beginning of the inserted code, but an option allows to override this behaviour to ensure the branch points to the original instruction.

B.3.4.4 Modifying an instruction

It is possible to modify an instruction at a given address using the function `madras_modify_insn`. This function allows to change the mnemonic and/or some or all operands of the instruction; it is also possible to remove or add operands. Mnemonics and operands must be provided as strings identical to their assembly representation, written in capital case.

A flag allows to choose the behaviour of the patcher when the modified instruction has a smaller coding than the original. In such a case, the MADRAS patcher can pad the remaining space with `nop` instructions, which allows to avoid displacing the area containing the modified instruction as is the standard behaviour. Otherwise, the modified instruction will be displaced as it would if larger than the original.

There is no test on the validity of the requested modifications when performing the request. If the modified instruction is invalid, an assembly error will be returned when committing the changes.

B.3.4.5 Deleting instructions

It is possible to delete one or more instructions in the file using the function `madras_delete_insns`. The instructions will be removed from the patched file. All direct branch targets pointing to a deleted instruction will be updated to point to the first remaining instruction immediately following the deleted ones.

The function `madras_replace_insns` allows to delete one or more instructions and replace them with `nop` instructions to preserve the size of the program, thus eliminating the need to perform code displacement. Direct branch instructions pointing to any replaced instruction are updated to point to the first instruction of the replacement block instead.

B.3.4.6 Linking modification requests

It is possible to link modification requests from one to another. This is only supported for modifications representing a insertion request (for code, function call, branches, ...). Linking a modification M' to modification M will ensure that the code of M' is executed after the one of M.

- If the linked modification M' has a non null address, an unconditional branch will be added after the last instruction inserted by M, branching to the first instruction inserted by M'.

- If the linked modification M' has a null address, the code that it should insert will be directly appended to the code inserted by M.

B.3.4.7 Adding conditions on modification requests

It is possible to add conditions for the execution of the code generated by the modification request when running the patched file. This is currently supported only for modifications representing a insertion request.

A condition object must first be created using `madras_cond_new`. A condition can be formed with either:

- A numerical value, an assembly operand represented as a MAQAO `oprnd_t` structure, and a comparison operator ($<$, $=$, $\leq$, ...)
- Two other conditions and a logical operator (*AND* or *OR*).

It is thus possible to build a complex condition using multiple comparison of operands and values. The current version does not support a comparison between two assembly operands.

A condition object can be attached to an existing modification using the function `madras_modif_addcond`. If another condition was already attached to this modification, the final condition attached to the modification will be formed by both conditions and the specified logical operand (*AND* by default).

It is also possible to add a condition to a modification from its string representation, using `madras_modif_setcond_fromstr`. The string representation of a condition must obey a syntax close to the one used in C:

- All conditions are enclosed by brackets ("(" and ")")
- A condition is formed by either of
  - two conditions separated by a logical operator ("&&" or "||")
  - an assembly operand and a value separated by a comparison operator ("==", "!=", "<", ">", "<=", ">=")
- Assembly operands used in conditions must be enclosed by quotes (")

A modification to which a condition was added can also be affected an `else` statement, using `madras_modif_addelse`. An else statement is an insertion modification set at a null address (an error will be raised if it is not). The code inserted by this modification will be executed if the condition is not met.

B.3.5 Patcher options

The `madras_modifs_addopt` function allows to tweak the behaviour of the patcher for a whole patching session. The `madras_modif_addopt` offers the same functionality, but limited to a single modification request.

B.3.5.1 Options altering code displacement

It is possible to alter how the MADRAS patcher handles code displacement when performing modifications that implies a change of size.

By default, the MADRAS patcher will not perform an insertion if there is not enough space to insert a branch to the displaced code. The option `PATCHOPT_FORCEINS` allows to force insertions even when there is not enough space for the branch. For specific cases this can cause insertions to actually succeed because adjacent blocks are also moved, thus leaving enough space for the insertions.

By default, the MADRAS patcher moves basic blocks when inserting code, which may lead to insertion failing when a basic block is too small for the insertion. The `PATCHOPT_MOVEFCTS` option allows the MADRAS patcher to attempt moving whole functions when such a case is encountered. This allows most patching operations to succeed, but can lead to errors in the patched file if a function was incorrectly detected or if indirect branches from other functions point inside the moved function, which is often the case in OpenMP codes. Conversely, the `PATCHOPT_MOV1INSN` option allows the MADRAS patcher to attempt moving only one instruction.

Those options may allow more patching operations to succeed by increasing the risk of causing the patched executable to crash. The behaviour of the MADRAS patcher is also not defined if those options are used on some but not all of the modification requests in a same basic block.

B.3.5.2 Options altering branch updates

It is possible to alter how the MADRAS patcher handles the updates of branches to instructions before which some code is added.

The default behaviour of the patcher is to update all branches to an instruction to point to the first instruction of code inserted before it. The `PATCHOPT_NO_UPD_EXTERNAL_BRANCHES` and `PATCHOPT_NO_UPD_INTERNAL_BRANCHES` options allow to restrict those updates respectively to the branch instructions belonging to the same function as the patched instruction and to a different function than the patched instruction. The `PATCHOPT_BRANCHINS_NO_UPD_DST` concerns inserted branch instructions, and prevents the branch to be updated if instructions are inserted before its destination.

B.3.6 Changing the padding instruction

By default, the instruction used to pad sections of code moved because of code displacement is the shortest `nop` instruction available for the given architecture. It is possible to set it to another instruction for a given modification using the `madras_modif_setpaddinginsn` function or for the whole patching session using the `madras_modifs_setpaddinginsn` function. The given instruction must have the same length as the default padding instruction or an error will be raised.

B.3.7 Committing changes

A patching session is ended by committing the changes with the `madras_modifs_commit` function, which performs all staged modifications and generates a patched file with the requested name. It is advised not to use the same name as the original. The operations are performed in the following order:

- Applying requests for renaming of dynamic libraries.
- Applying requests for addition of dynamic and static libraries.
- Applying requests for addition of global variables.

- Applying all requests for code modification in the order of their addresses.
- Saving the patched file under the given name.
- Freeing the list of pending requests.

At this point the generated patched file contains all requested changes, except for those that caused an error during patching. It is not advised to keep using the MADRAS `elfdis_t` structure for other operations afterwards.

B.4 Logging

The **libmadras** API allows to record all requests performed on a disassembled file and write them to a log file. By default, this feature is turned off and can be activated using the `madras_traceon` function. It is also possible to choose the name of the log file, by default **madras_trace.log**.

All invoked API functions will be recorded in the log, along with their parameters. A MADRAS script offers to turn such a file into a C source file, allowing to compile it and rerun the requests.

B.5 Example of use of the MADRAS API

The code below presents an example of the use of the MADRAS API. It defines the function `insert`, which allows to patch a file by inserting a function call at a given address and save the patched file under a different name. The required parameters are the name of the file to patch, the static or dynamic library defining the function to insert (an empty string specifies that the function is defined in the patched file), the name of the function to insert, the address at which it must be inserted, and the name of the patched file. The `main` function displays an example of invocation of this function.

```
#include <libmadras.h>
void insert(char* file, char* lib, char* fct,
                  int64_t addr, char* out) {
  //Disassembles the file and inits the modifications
  elfdis_t* madras = madras_disass_file(file);
  madras_modifs_init(madras, STACK_SHIFT, 512);
  //Adds a function call at the given address
  insert_t* ifct = madras_fctcall_new(madras, fct, lib, addr, 0);
  //Adds the given address as an immediate parameter
  madras_fctcall_addparam_imm(madras, ifct, addr, 0);
  //Commits changes and creates patched file
  madras_modifs_commit(madras,out);
  //Terminates the madras structure
  madras_terminate(madras);
}
void main(int argc, char* argv[]) {
  insert("myfile", "libfoo.so", "myfunction",
                0x400042, "myfile-patched");
}
```

B.6 The madras executable

The **madras** executable is a standalone program allowing to use most of the features of the API. In the case of patching, those are simplified and mainly intended for test or examples. **madras** was used to perform the performance tests of the MADRAS disassembler whose results are presented in Section 4.5. It is invoked through the following command line:

```
$ madras options filename
```

where `filename` is the path of a valid ELF file and `options` a set of flags controlling **madras** behaviour. **madras** accepts the following options:

- `-d`, `--disassemble`: Prints the disassembly of all code sections in the file
- `-t`, `--disassemble-text`: Prints the disassembly of the .text section of the file
- `--shell-code`: Prints the disassembly of all sections, with the hexadecimal encodings of instructions in string format.
- `--label=`*`name`*: Prints instruction from the given label to the next one.
- `--color-mem`: Adds colors during printing: colors instructions performing memory accesses in red and floating point instructions in blue.
- `--color-jmp`: Adds colors during printing: colors unsolved indirect branches in red, solved indirect branches in green and other branches in blue.
- `--no-coding`: Does not print instruction codings.
- `--get-external-fct`: Gets external functions using ELF data.
- `--get-dynamic-lib`: Gets dynamic libraries using ELF data.
- `--with-family`: Adds instruction family during printing.
- `--with-annotate`: Adds instruction annotations during printing.
- `--with-roles`: Adds operand roles during printing.
- `--with-debug`: Prints debug information from the file (if available and retrieved)
- `--no-debug`: Does not attempt to retrieve debug information from the file.
- `--count-insns`: Prints the number of instructions in the file.
- `--raw-disass` *`arch`*: Raw disassembly: disassembles the whole content of the file without parsing the ELF using architecture *arch*. The following filters can be used to control the part of the binary file to disassemble:
  - `--raw-start` *`offset`*: Starts disassembly after *offset* bytes (0 if not set).
  - `--raw-len` *`len`*: Disassembles *len* bytes (whole file if not set or set to 0). Ignored if `raw-stop` is used first.
  - `--raw-stop` *`offset`*: Stops disassembly at *offset* bytes (whole file if not set or set to 0). Ignored if `raw-len` is used first.
  - `--raw-first` *`addr`*: Assigns address *addr* to the first disassembled instruction (0 if not set).
- `-e`, `--printelf`: Prints all data retrieved from the ELF file. The following filters can be used to print only a part of the ELF data:
  - `--elfhdr`: Prints ELF header.
  - `--elfscn`: Prints ELF section headers.
  - `--elfseg`: Prints ELF program headers.
  - `--elfrel`: Prints ELF relocation tables.

  - `--elfdyn`: Prints ELF dynamic tables.
  - `--elfsym`: Prints ELF symbol tables.
  - `--elfver`: Prints ELF version tables.
  - `--elf-code-areas`: Prints the start, length and stop of consecutive sections containing executable code in the file.

- `--function=`*`format`*: Inserts a function call. The function does not have any parameters. *format* is a string containing parameters used to insert the function. It has the following structure:

  `fct``;[@`*`address`*`[@`*`address...`*`]][;`*`library`*`][;`*`after/before`*`][;`*`wrap/no-wrap`*`]`

  - *fct* is the name of the function to insert.
  - *address* is the address at which the function call must be inserted. If not specified, the function is inserted but not called.
  - *library* is a dynamic library containing the function. If not specified, it is assumed that *fct name* is an internal function.
  - *after/before* can be used to choose if the function call must be inserted before or after the instruction at *address*. *before* is the default choice.
  - *wrap/no-wrap* can be used to choose if the context must be saved before the function call and restored afterwards. *wrap* is the default choice.

- `--delete=`*`format`*: Deletes one or several instructions. The *format* parameter has the following structure: *`@address`*`[@`*`address...`*`][;`*`number`*`]`
  - *address* is the address of the first instruction to be deleted.
  - *number* is the number of instructions to delete. If not specified, the default value is 1. *number* must be a positive value.

- `--stack-keep`: Sets the method for safeguarding the stack to `STACK_KEEP`.
- `--stack-move`: Sets the method for safeguarding the stack to `STACK_MOVE`.
- `--stack-shift=`*`value`*: Sets the method for safeguarding the stack to `STACK_SHIFT`. This is the default option, with 512 for *value*.
- `--set-machine=`*`value`*: For ELF binaries, changes to machine type for which it is compiled in the ELF header to *value*.
- `--check-file`: Check that the file is a valid ELF executable, linkable or library file.
- `-o`, `--output` *`output`*: Saves the file to *output*. If no patching command has been issued, the new file will be identical. If omitted while a patching command has been issued, the result file will be *`filename`*`_mdrs`.
- `-m`, `--mute`: Disassembles but does not print anything.
- `-h`, `--help`: Prints the detailed options of `madras`.
- `-v`, `--version`: Displays the `madras` executable version.

Below are some examples of use of the `madras` executable:

- Disassembling binary file `foo` and printing debug data:

```
madras -d foo --debug-print
```

- Patching file `foo` to insert function `bar` from `libfoo.so` at address 0x400000, and saving the result as `foo-patch`:

```
madras foo --function=bar;@0x400000;libfoo.so -o foo-patch
```

Bibliography

[1] AMD64 Architecture Programmer's Manual Volume 1: Application Programming. http://support.amd.com/us/Processor_TechDocs/. 8

[2] AMD64 Architecture Programmer's Manual Volume 2: System Programming. http://support.amd.com/us/Processor_TechDocs/. 8

[3] AMD64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions. http://support.amd.com/us/Processor_TechDocs/. 8

[4] AMD64 Architecture Programmer's Manual Volume 4: 128-bit and 256 bit media instructions. http://support.amd.com/us/Processor_TechDocs/. 8

[5] AMD64 Architecture Programmer's Manual Volume 5: 64-Bit Media and x87 Floating-Point Instructions. http://support.amd.com/us/Processor_TechDocs/. 8

[6] a.out – assembler and link editor output. cm.bell-labs.com/cm/cs/who/dmr/man51.ps. 15

[7] ARM Architecture Reference Manual. https://silver.arm.com/download/ARM_and_AMBA_Architecture/. 11

[8] ARM Software Development Toolkit. http://infocenter.arm.com/help/topic/com.arm.doc.dui0041c/DUI0041C.pdf. 17

[9] ARM®Architecture Reference Manual ARMv7-A and ARMv7-R edition. https://silver.arm.com/download/ARM_and_AMBA_Architecture/. 11

[10] ARM®v7-M Architecture Reference Manual. https://silver.arm.com/download/ARM_and_AMBA_Architecture/. 11

[11] ARMv6-M Architecture Reference Manual. https://silver.arm.com/download/ARM_and_AMBA_Architecture/. 11

[12] ARMv8 Instruction Set Overview. https://silver.arm.com/download/ARM_and_AMBA_Architecture/. 11

[13] Common Object File Format. http://www.ti.com/lit/an/spraao8/spraao8.pdf. 16

[14] Executable and Linkable Format (ELF). http://www.skyfree.org/linux/references/ELF_Format.pdf. ix, 14

[15] GNU gprof. In *GNU Binary Utilities*. Free Software Foundation, Inc. 76

[16] IDAPro Disassembler. Hex-Rays. https://www.hex-rays.com. 54, 80

[17] Intel®64 and IA-32 Architectures Software Developer's Manual Combined Volumes 2A, 2B, and 2C: Instruction Set Reference, A-Z. http://download.intel.com/products/processor/manual/. 8

[18] Intel®64 and IA-32 Architectures Software Developer's Manual Combined Volumes 3A, 3B, and 3C: System Programming Guide, Parts 1 and 2. http://download.intel.com/products/processor/manual/. 8

[19] Intel®64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture. http://download.intel.com/products/processor/manual/. 8

[20] Intel®Itanium®Architecture Software Developer's Manual. http://www.intel.com/content/dam/doc/manual/itanium-architecture-vol-1-2-3-4-reference-set-manual.pdf. 10

[21] Intel®Xeon Phi™Coprocessor Instruction Set Architecture Reference Manual. http://download-software.intel.com/sites/default/files/forum/278102/327364001en.pdf. 9

[22] Intel®Xeon Phi™Coprocessor System Software Developers Guide. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-software-developers-guide.pdf. 9

[23] MAQAO: Modular Assembly Quality Analyser and Optimizer. http://www.maqao.org. 61

[24] Microsoft Portable Executable and Common Object File Format Specification. www.skyfree.org/linux/references/coff.pdf. 16

[25] Mont-Blanc: European approach toward energy efficient high performance. http://www.montblanc-project.eu/home. 11

[26] Ndisasm. In *NASM Documentation*. The NASM team. http://www.nasm.us/xdoc/2.10.07/html/nasmdoca.html. 52

[27] Objdump. In *GNU Binary Utilities*. Free Software Foundation, Inc. http://sourceware.org/binutils/docs/binutils/objdump.htm. 51

[28] OS X ABI Mach-O File Format Reference. 17

[29] Power ISA™Version 2.06 Revision B. https://www.power.org/documentation/. 11

[30] Standard performance evaluation corporation. http://www.spec.org/. 61

[31] The DWARF Debugging Standard. dwarfstd.org/. 15

[32] XCOFF Object File Format. http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.aix.files/doc/aixfiles/XCOFF.htm. 16

[33] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. pages 93–112, 1986. 17

[34] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: Tools for performance analysis of optimized parallel programs, 2008. 5

[35] V. Agel, L. Eichinger, H.-W. Eroms, P. Hellwig, H.-J. Heringer, and H. Lobin. Parsing with dependency grammars. 21

[36] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison Wesley, 2007. 27

[37] K. An, A. Kotha, M. Smithson, R. Barua, and A. D. Retrofitting security in cots software with binary rewriting. 79

[38] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 295–308, New York, NY, USA, 2013. ACM. 79

[39] D. Barthou, A. Charif Rubial, W. Jalby, S. Koliaï, and C. Valensi. Performance Tuning of x86 OpenMP Codes with MAQAO. In M. S. Muller, M. M. Resch, A. Schulz, and W. E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 95–113. Springer Berlin Heidelberg, 2010. 58, 89

[40] A. R. Bernat and B. P. Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, PASTE '11, pages 9–16, New York, NY, USA, 2011. ACM. 53, 78

[41] Bryan Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications*, 14:317–329, 2000. 5, 53, 78

[42] D. Chanet and L. V. Put. Compacting Arm Binaries with the Diablo Framework, 2003. 79

[43] A. S. Charif-Rubial, D. Barthou, C. Valensi, S. S. Shende, A. D. Malony, and i.-p. William Jalby. MIL: A language to build program analysis tools through static binary instrumentation. In *20th Annual International Conference on High Performance Computing (HiPC'13)*, Hyderabad, India, Dec. 2013. xi, 89, 90

[44] M. Charney. XED. Intel Corporation. `http://software.intel.com/sites/landingpage/pintool/docs/58423/Xed/html/`. 52

[45] V. Chipounov and G. Candea. Enabling sophisticated analyses of x86 binaries with revgen. 54

[46] J. Chung, M. Dalton, H. Kannan, and C. Kozyrakis. Thread-safe dynamic binary translation using transactional memory. 80

[47] C. Cifuentes and M. V. Emmerik. Recovery of Jump Table Case Statements from Binary Code. In *Science of Computer Programming*, pages 2–3, 1999. 47

[48] C. Cifuentes, M. V. Emmerik, N. Ramsey, and B. Lewis. Experience in the Design, Implementation and Use of a Retargetable Static Binary Translation Framework. Technical report, 2002. 53

[49] G. Dabah. Distorm. `http://code.google.com/p/distorm/`. 52

[50] N. K. Dahra. Disassembly and parsing support for retargetable tools using sim-nml, 2007. 22

[51] B. De Bus, B. De Sutter, L. Van Put, D. Chanet, and K. De Bosschere. Link-time optimization of ARM binaries. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '04, pages 211–220, New York, NY, USA, 2004. ACM. 79

[52] B. D. de Dinechin. A machine description system. 22

[53] A. Desnos, S. Roy, and J. Vanegue. ERESI : une plate-forme d'analyse binaire au niveau noyau, 2008. 80

[54] T. G. developers. GDB: The GNU Project Debugger. Free Software Foundation, Inc. `http://sourceware.org/gdb/`. 52

[55] N. A. S. Division. Nas parallel benchmarks. `http://www.nas.nasa.gov/publications/npb.html`. 89

[56] A. Fauth, J. V. Praet, and M. Freericks. Describing instruction set processors using nml. In *in Proceedings of the Conference on Design, Automation and Test in Europe*, pages 503–507, 1995. 22

[57] A. Fog. Objconv. `http://www.agner.org/optimize/objconv-instructions.pdf`. 54

[58] A. Fog. Calling conventions for different C++ compilers and operating systems, 2008. 13

[59] B. Ford. Packrat Parsing: Simple, Powerful, Lazy, Linear Time. 20, 40

[60] B. Ford. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In *Symposium on Principles of Programming Languages*, pages 111–122. ACM Press, 2004. 20

[61] M. Freericks. The nml machine description formalism. 22

[62] M. Gerndt and S. Strohhäcker. Distribution of Periscope Analysis Agents on ALTIX 4700. 77

[63] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a Call Graph Execution Profiler, 1982. 76

[64] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An Instruction Set Description Language for Retargetability, 1997. 22

[65] D. Kästner. TDL - A Hardware and Assembly Description Language, 2000. 54

[66] M. Kerrisk. PTRACE(2), 2013. `man7.org/linux/man-pages/man2/ptrace.2.html`. 76

[67] C. keung Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and R. K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *In PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200. ACM Press, 2005. 5, 78, 80

[68] S. Koliaï, Z. Bendifallah, M. Tribalat, C. Valensi, J.-T. Acquaviva, and W. Jalby. Quantifying performance bottleneck cost through differential analysis. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ICS '13, pages 263–272, New York, NY, USA, 2013. ACM. 90

[69] J. R. Larus and T. Ball. Rewriting Executable Files to Measure Program Behavior. *SOFTWARE PRACTICE & EXPERIENCE*, 24:197–218, 1994. 77

[70] C. Lattner and V. Adve. Architecture for a Next-Generation GCC. In *Proc. First Annual GCC Developers' Summit*, Ottawa, Canada, May 2003. 22

[71] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation, 2004. 22, 52, 54, 79

[72] J. Levine, T. Mason, and D. Brown. *Lex & Yacc*. Computer Science Series. O'Reilly & Associates, 1992. `http://books.google.fr/books?id=YrzpxNYegEkC`. 22

[73] S. Li. A Survey on Tools for Binary Code Analysis, 2004. 49

[74] C. Linn and S. Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *IN ACM CONFERENCE ON COMPUTER AND COMMUNICATIONS SECURITY (CCS*, pages 290–299. ACM Press, 2003. 46, 47

[75] C. Linn, S. Debray, G. Andrews, and B. Schwarz. Stack Analysis of x86 Executables. 74

[76] H. Lu. ELF: From The Programmer's Perspective. *NYNEX Science & Technology Inc*, page 95, 1995. 14

[77] D. D. McCracken and E. D. Reilly. Backus-Naur form (BNF). In *Encyclopedia of Computer Science*, pages 129–131. John Wiley and Sons Ltd., Chichester, UK. `http://dl.acm.org/citation.cfm?id=1074100.1074155`. 38

[78] J. Mellor-crummey, R. Fowler, and G. Marin. HPCView: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23:2002, 2002. 5

[79] Michael A. Laurenzano and Mustafa M. Tikir and Laura Carrington and Allan Snavely. PEBIL: Efficient Static Binary Instrumentation for Linux. 53, 78

[80] B. P. Miller and K. A. Roundy. Binary-Code Obfuscations in Prevalent Packer Tools. *ACM Computing Surveys*, June 2012. 48

[81] B. P. M. Nathan E. Rosenblum, X. (Jerry) Zhu and K. Hunt. Learning to Analyze Binary Computer Code, 2008. 46, 53

[82] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *In Third Workshop on Runtime Verification (RV'03*, 2003. 79

[83] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *In Proceedings of the 2007 Programming Language Design and Implementation Conference*, 2007. 79

[84] M. Pietrek. An In-Depth Look into the Win32 Portable Executable File Format. `http://msdn.microsoft.com/en-us/magazine/cc301805.aspx`. 16

[85] V. Rajesh and R. Moona. Processor modeling for hardware software codesign. In *in Int. Conf. on VLSI Design*, pages 132–137, 2000. 22

[86] N. Ramsey and M. F. Fernández. Specifying Representations of Machine Instructions. *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS*, 19:492–524, 1997. 21

[87] T. Reps, G. Balakrishnan, J. Lim, and T. Teitelbaum. A Next-Generation Platform for Analyzing Executables. In *In APLAS*, pages 212–229, 2005. 54

[88] E. Rohou, F. Bodin, A. Seznec, G. L. Fol, F. Charot, and F. Raimbault. Salto: System for assembly-language transformation and optimization, 1996. 22

[89] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *In Proceedings of the USENIX Windows NT Workshop*, pages 1–7, 1997. 80

[90] L. Ryzhyk. The ARM Architecture, 2006. 11

[91] B. Schwarz, S. Debray, and G. Andrews. Disassembly of Executable Code Revisited. In *In Proc. IEEE 2002 Working Conference on Reverse Engineering (WCRE*, pages 45–54. IEEE Computer Society, 2002. 49

[92] B. Schwarz, S. Debray, G. Andrews, and M. Legendre. PLTO: A Link-Time Optimizer for the Intel IA-32 Architecture. In *In Proc. 2001 Workshop on Binary Translation (WBT-2001)*, 2001. 53, 81

[93] A. Sepp, J. Kranz, and A. Simon. GDSL: A Generic Decoder Specification Language for Interpreting Machine Language. *Electron. Notes Theor. Comput. Sci.*, 289:53–64, Dec. 2012. 21

[94] S. S. Shende and A. D. Malony. The Tau Parallel Performance System. *The International Journal of High Performance Computing Applications*, 20:287–331, 2006. 5, 89

[95] T. G. team. GCC, the GNU Compiler Collection. Free Software Foundation, Inc. `http://http://gcc.gnu.org/`. 22

[96] V. Thampi. Udis86. `http://udis86.sourceforge.net/`. 52

[97] H. Theiling, U. D. Saarlandes, and A. A. I. Gmbh. Extracting Safe and Precise Control Flow from Binaries. In *In Proc. 7th Conference on Real-Time Computing Systems and Applications*, 2000. 54

[98] W. Underwood. Grammar-Based Specification and Parsing of Binary File Formats. *The International Journal of Digital Curation*, pages 95–106, 2012. 20

[99] D. A. Varley. Practical Experience Of The Limitations Of Gprof, 1993. 76

[100] L. Ďurfina, J. Křoustek, P. Zemek, D. Kolář, T. Hruška, K. Masařík, and A. Meduna. Design of an Automatically Generated Retargetable Decompiler. In *2nd European Conference of COMPUTER SCIENCE (ECCS'11)*, pages 199–204. North Atlantic University Union, 2011. 54

[101] D. W. Wall. Systems for Late Code Modification. In *WRL Research Report 91/5*, pages 275–293. Springer-Verlag, 1991. 76

[102] C. Wang, S. Hu, H.-s. Kim, S. R. Nair, M. Breternitz, Z. Ying, and Y. Wu. Stardbt: An efficient multi-platform dynamic binary translation system. In *Proceedings of the 12th Asia-Pacific Conference on Advances in Computer Systems Architecture*, ACSAC'07, pages 4–15, Berlin, Heidelberg, 2007. Springer-Verlag. 80

[103] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *In Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC*, 2012. 80

[104] W. A. Woods. Transition network grammars for natural language analysis. 21