Universidade Federal de Pernambuco
Graduação em Ciências da Computação

Undergraduate Project

Developing an Intermediate Representation for the Analysis of Binary Code

Student: Julio Auto de Medeiros
Supervisor: Prof. André Santos

Recife, August 21, 2007

Signatures

This work is the result of the efforts of the student Julio Auto de Medeiros, under the supervision of Prof. André Santos, under the title of “Developing an Intermediate Representation for the Analysis of Binary Code” and conducted at the Center of Informatics of the Federal University of Pernambuco (CIn-UFPE). The people listed below acknowledge the contents of this document and the results of this Undergraduate Project.

Julio Auto de Medeiros

André Luís de Medeiros Santos

Acknowledgments

This work could not be accomplished without the participation of many other people. Although their involvement with this may vary on different levels of (in)direction, I fail to recognize a clear classification of their importance and, therefore, list them in no particular order. I would like to thank:

- God, Jesus Christ and all the others that influence my religious convictions, for they constitute such a big part of the person I am today;
- My family (Humberto, Evelina, Felipe and Humberta), for giving me a healthy and happy environment that I could grow up in and call ‘home’, and for being an eternal treasure to be proud of;
- Regina, for being absolutely everything she is, and for letting me spend some time enjoying the pleasure of her company;
- My supervisor, André Santos, for having created many learning experiences during the many classes of his I have attended throughout my undergraduate course and for accepting me and my work to go under his supervision;
- My dear friend, Julien Vanegue, for being available practically whenever I feel like having the long technical chats that have honestly influenced every line in this work;
- All of my friends (that have stayed or gone), for letting me laugh, cry, think, learn and try to be a better person every time I meet them. And, of course, for grabbing a beer with me every now and then;
- And finally, but not lastly, to all of those that I forgot to mention here. I am deeply sorry if you deserved a more proper acknowledgment, but I honestly hope to be forgiven.

Resumo

O campo de Análise de Programas é vasto e complexo. Apesar de ele ter muitas décadas de estudos e avanços, alguns dos maiores e mais alvejados problemas ainda se encontram em aberto. Em particular, uma rápida busca na literatura sobre a intersecção entre as disciplinas de análise estática de programas binários e detecção automática de bugs mostra que existe uma grande oportunidade para cientistas dispostos a se engajar nesse excitante campo de pesquisa.

Esse trabalho tenta fazer um estudo sobre o estado-da-arte dos assuntos que tangem as questões relacionadas à análise estática de programas, análise de binários e detecção automática de bugs. Uma vez propriamente contextualizado, esse documento apresentará o framework ERESI, um projeto open-source sobre o qual toda a implementação deste trabalho foi baseada. Por fim, o leitor encontrará um relatório detalhado do trabalho feito para transformar código de máquina Intel IA-32 na LIR (Low-level Intermediate Representation) do ERESI, um passo importante para estender as funcionalidades de análise do framework em questão.

Abstract

The field of Program Analysis is vast and complex. Even though it has many decades of study and advances now, some of the biggest and most pursued problems remain open for resolution. In particular, a quick search through the literature on the intersection between the disciplines of static analysis of binary programs and automated bug-finding reveals that there is a big window of opportunity open to scientists willing to engage in this exciting research field.

This work attempts to perform a survey on the state-of-the-art of the subjects touching the questions on static program analysis, binary analysis and automated bug-finding. Once properly contextualized, this document will introduce the ERESI framework, an open-source project on top of which all of this work's implementation is based. Finally, the reader will find a detailed report of the work done to transform Intel IA-32 machine code into the ERESI LIR (Low-level Intermediate Representation), an important step to extend the analysis features of the framework in question.

Table of Contents

Index of Tables

Index of Figures

1. Introduction

For many years now, a considerable amount of effort has been put into making the automated analysis of software a powerful and practical tool in the workbench of professionals and researchers that deal with software. During this time, many formal approaches to software verification have been created and further research still goes on in an attempt to make these techniques capable to answer the most relevant questions in program analysis, some of which are: Does this software have bugs? How can these bugs be triggered? How critical are they?

Furthermore, a branch of this research field (and a quite recent one) is of special importance: the analysis of binary code. Being capable of analyzing binary code enhances the possibility to answer all those intriguing questions without the need of the source code. All that is needed is the software in its most natural form: the bits and bytes that are ready to be fed into a program loader.

The reasons behind the concerns of the software industry and academic researching around the mitigation of bugs vary in perspectives. The market undeniably suffers with the existence of bugs by having to make considerable investments in processes and tools that aid in bug mitigation during the development cycle. To this date, no process or tool does the job perfectly and, after the software release and deployment, remaining bugs may cause the company to lose customers and/or invest more money in fixes and patching initiatives. Furthermore, depending on how vital third-party software is to a given businesses, bugs in this third-party software may bring losses to the organization making use of it.

From a security standpoint, the situation is no better. Surveys reveal that, during the year of 2006, over 7,000 security vulnerabilities were publicly disclosed [ISS07]. Experts of the field make even more alarming evaluations of the scenario, taking guesses that, during the same year, the number of undisclosed vulnerabilities could cross the mark of 130,000 [Oll07], giving a grand total of almost 140 thousand security breaches, most of them caused by bugs in software products.

1.1. Context and Objectives

Many of the problems regarding program analysis remain unsolved, although the subject receives a fair amount of attention and research contributions from the scientific community. Furthermore, in contrast with the exciting achievements in source-code analysis and the release of academic and commercial software tools that can perform rather effective and useful analysis of source (e.g. finding occurrences of a given class of software bugs), the literary references to binary analysis techniques are considerably less expressive and more scarce. That is, at least in part, because of the added complexity of binary analysis in comparison to its source-code counterpart. For example, binary analyzers have to deal with extra issues such as the absence of types, provided no debugging information is available.

This work intends to contribute to the advances in binary analysis by participating in the development of the ERESI project. ERESI (ELF Reverse Engineering Software Interface) is a framework dedicated to the analysis and instrumentation of binary programs. It is compatible with multiple architectures and runs on a variety of UNIX-based operating systems, by working on ELF (Executable and Linking Format) objects. The framework is formed by a set of seven libraries and three applications that run on top of them, providing innovative features for analysis, instrumentation, tracing and even debugging of binaries, among other possibilities [VGA+07].

The remainder of this text includes, in order: a study of the state-of-the-art in program analysis, a presentation of the ERESI framework and its components most closely related to this work and, finally, a report of the work done while dealing with the transformation of Intel IA-32 machine code into the ERESI LIR (Low-level Intermediate Representation). Brief words of conclusion can be found at the end of this document, just above the bibliographic references and appendices.

2. Program Analysis

This section discusses some of the many techniques and approaches that may be taken by program analyzers. For the sake of comprehensiveness, this set of subjects was restricted only to those that matter the most to this work as a whole.

It is important to notice, as well, that the term ‘program analysis’ here refers to the ways of making a computational system reason automatically (or at least with little human assistance) about the behavior of a program and draw conclusions that are somehow useful. Conceptually, this process can be divided into two distinct steps: one to gather substantial information about the program in question, i.e. the analysis itself; and another to extract the actual conclusive data, i.e. the verification. The results of the verification step are said to be useful typically in the sense that, otherwise, they would be harder to analyze, i.e. by manually inspecting the code.

Program analyzers commonly fall into one of these two categories: static analyzers or dynamic analyzers. Then again, this text refers to ‘program analysis’ to describe the techniques used by static analyzers, unless when explicitly noted. The main difference between these two branches is that the dynamic analyzers need to really execute (or otherwise emulate the execution of) the program in order to perform their analyses, analyzing as the program goes. On the other hand, static analyzers need not to execute the program, and work just by transforming and reasoning about a given representation of the program in question, such as its source-code. Static analysis is used, for example, in many of the modern compilers due to its importance to optimization and error-checking. Because of that, program analysis is frequently related to compiler research and literature. Due to this proximity between the compiler and analysis disciplines, this text may freely employ the term ‘program analyzer’ to refer to a compiler or compiler suite as far as it concerns its program analysis functionalities.

In the following sub-sections we shall discuss some topics of major interest to program analysis. There are certainly many resources and techniques that program analyzers can make use of and it is not the purpose of this document to cover all of them, neither in breadth nor in depth. Instead, this section aims to contain comprehensive explanation about the subjects that are most relevant to modern program analyzers. The reader is expected to find more extensive information in the references.

2.1. Intermediate Representations

It is safe to say that the IR (Intermediate Representation) is the central piece to every modern program analyzer. In simple terms, the IR is merely an alternative representation to the one of the input program at the analyzer front-end, which can accept many forms of representation such as multiple programming languages or machine code for various computer architectures.

Besides multiplexing the possibly many input representations of a program analyzer into a unified form, the IR typically has many more important features.

As the program analyzer executes its phases, the IR is decorated with information that is critical to the process of verification or to more elaborated analysis routines. It is also often required that the IR must not contain information that is present in the input representation but irrelevant to the posterior phases of the analyzer. It can be said, then, that in some sense the IR is the result of the analysis itself.

In practice, an IR may add complexity to the representation of the program by incorporating information about data-flow, control-flow, types of operands and so forth. On the other hand, to keep things simple, the IR may choose to represent the program by using a set of operations with cardinality less than that of the original representation.

2.2. Flow Graphs

Graphs are data structures that are able to express undirected or directed relations between many-to-many entities of some sort. In graph terminology, an object pertaining to a graph is called a *node*, or *vertex*. A link between a pair of vertices is called an *edge*. Graphs are often represented under graphical forms similar to the one below:

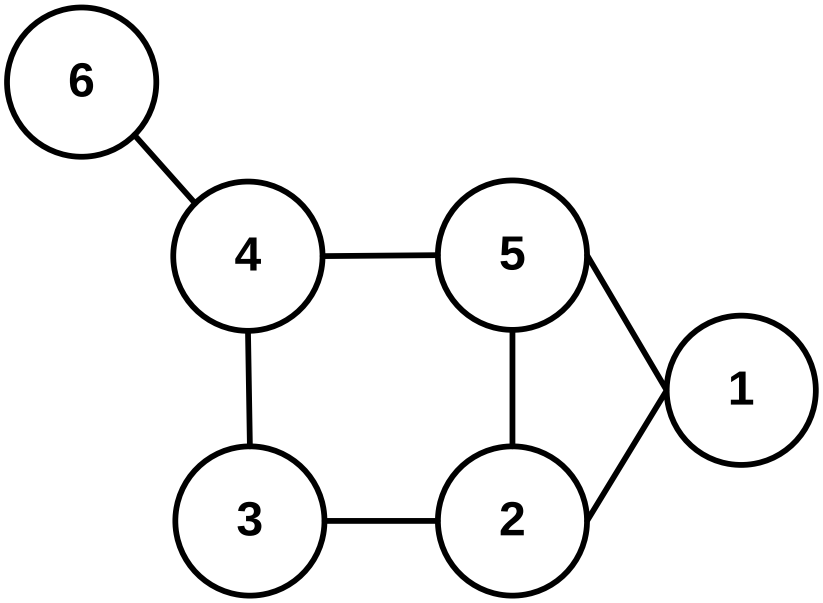

Figure 1. Undirected graph with six vertices and seven edges

In program analysis, graphs are especially useful to represent two types of relations: control-flow links, and data-flow dependencies. This kind of information is largely used by optimization and analysis techniques and often a program analyzer chooses to embed them into its IR.

Flow graphs are directed graphs that have a start node and an end node, thus inducing the notion of flow. In graphical representations, the start node is typically the topmost node, while the end node is the bottom-most one.

Some properties of flow graphs are important to program analysis, such as *reachability*. Reachability is a property that states whether there is a path between two nodes (preserving the direction of the edges, i.e. the flow of the graph). If there is a path from, say node X, to node Y we say that node Y is *reachable* from node X. Node Y is *unreachable* from node X otherwise.

Dominance is another property of flow graphs. We say that node X *dominates* node Y if every path from the start to node Y must pass through node X. Conversely, we say that node Y *postdominates* node X if all paths from node X to the end must go through node Y. The same concepts can be applied to edges in an analogous manner.

2.3. Control-Flow Information

Modern software does not work purely linearly, i.e. it frequently requires that the control of execution is transferred on demand to another operation it wishes no matter where the target operation lies in the memory address space. Therefore, the modern computers provide operations that the programmer can use to perform these transfers of control.

Changes in the control-flow are conceptually present in every conditional block, loop or procedure call and the compiler typically makes large use of them. Control-flow analysis may indicate, for example, that a part of the program is unreachable (i.e. it will never be executed) and that its code can be removed for the sake of size optimizations.

About the division of a program in parts, as far as it concerns control-flow it can be performed by splitting the program into either of tow kinds of unit: subroutines or basic blocks. Subroutines are the usual procedures, functions or methods that are present on the majority of the modern programming languages. Consider the following program in C code:

```c
int subtract (int a, int b) {
  int c = a - b;
  return c;
}

int main () {
  int a, b, c;
  a = 2;
  b = a;
  c = subtract(a, b);

  if (c == 0)
    return 0;
  else
    return -1;

  return -2;
}
```

Code 1. A simple C program

The control-flow graph that represents the links between subroutines is called a call graph. Figure 2 shows the call graph generated from Code 1.

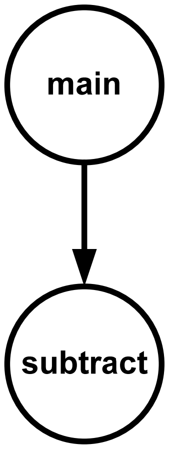

Figure 2. Call Graph for Code 1

Basic blocks, on the other hand, are units of a finer level of granularity. They are blocks of code that can be executed from the start to the end without transferring control in between. As control-flow graphs link its vertices by terms of control transfers, the last operation is usually a control-flow statement and the first operation, the target of another control-flow statement. The obvious exceptions to this rule are the block containing the starting point of the program (or that are otherwise invoked by an ‘invisible’ party, such as the operating system program loader) and the blocks either halt execution or that do not transfer control to other blocks other than by executing return statements.

The flow graphs generated from control-flow information of basic blocks are simply called control-flow graphs (CFGs, for short) or block-level CFGs. In Figure 3, b_main_1 refers to the block of code containing the lines 7 to 10 of Code 1. The block b_main_2 corresponds to the test performed by line 12 and b_main_3 to the code in line 17, just after the if-then-else statement. The other blocks are hopefully intuitive enough.

It is important to notice that the block b_main_3 in fact could never be reached and, therefore, could be optimized away. It becomes clear, then, that the choices made during the generation of the control-flow information and, consequently, of the CFG influence on the quality and overall usability of the data for analysis purposes. In this specific case, the control-flow analyzer could easily opt not to insert out-edges into blocks ending with a return statement, such as b_main_if and b_main_else, thus generating a CFG with accurate information about the reachability of b_main_3. Many other algorithmic choices could be made, e.g. creating an edge out of b_subtract and into b_main_2, which could alter the quality of the result of the analysis, be it in complexity, soundness or any other criterion that ultimately affects the excellence of the analyzer.

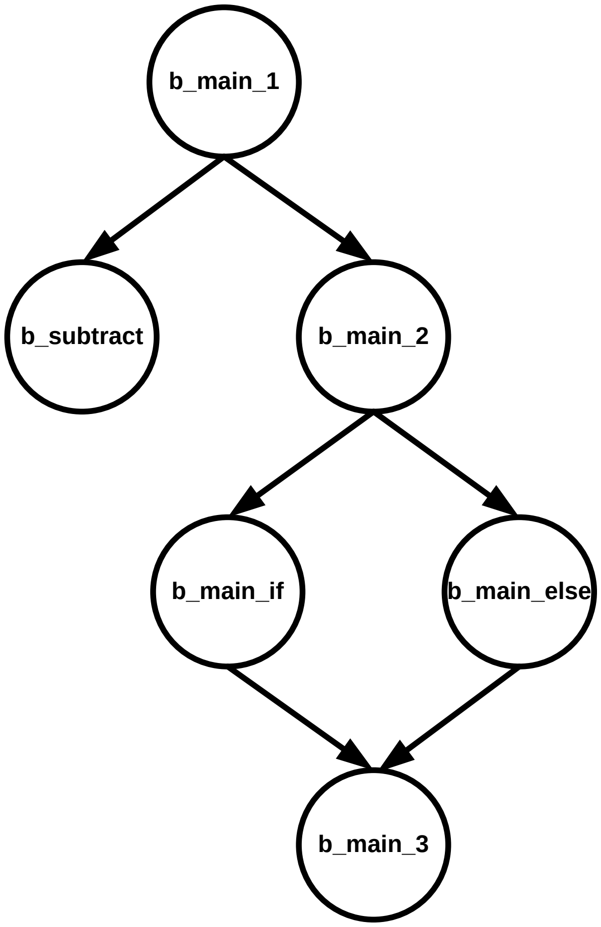

Figure 3. Control-Flow Graph for Code 1

2.4. The Program Structure Tree

The program structure tree [JPP94], or PST, is another representation of the control-flow graph of a program. Its nodes, however, are not basic blocks or subroutines. Instead, they represent Single Entry Single Exit (SESE) regions. Intuitively, SESE regions are segments of a given graph that are entered via only one edge and exited by only one different edge. More formally, SESE regions are bounded by a pair of edges (X,Y) such that X dominates Y, Y postdominates X, and every cycle in the graph containing X also contains Y and vice versa. Figure 4 highlights the SESE regions of the CFG from Figure 3.

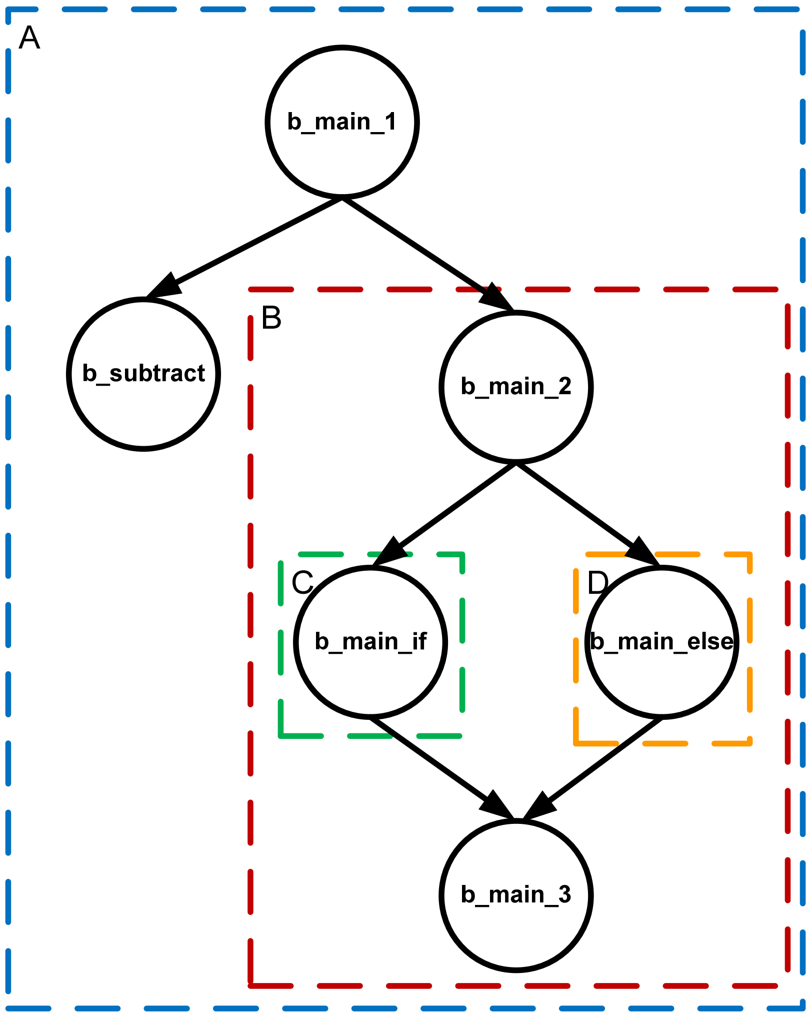

Figure 4. CFG with SESE regions highlighted

A PST is, then, the representation of the SESE regions of a CFG based on their nesting relationships. That is, in a PST a node Y is a child of node X if the SESE region represented by node Y is contained in the SESE region represented by node X. Figure 5 depicts the PST for the SESE regions highlighted in Figure 4.

PSTs can be used in divide-and-conquer analysis strategies where the complexity of the algorithm is high and the cost of combining the partial results is low. Suppose we have an algorithm of quadratic complexity in proportion to N, where N is the number of units it can operate upon (e.g. basic blocks, operations…) in the target program. Now imagine we have a PST with k SESE regions of about the same size, N/k (i.e. the same number of units), and we can run the algorithm unmodified over the SESE regions. If the cost of combining the results of the algorithm for each SESE region is irrelevant, the algorithm will run, for each SESE region, in $O\left((N/k)^2\right)$ time and take an overall $O\left(N^2/k\right)$ time, thus speeding up the performance for the whole analysis. Furthermore, if the algorithm can enjoy from properties of sparsity of the program, i.e. if it can be applied only to a subset of the SESE regions in the PST, this result can clearly be improved even more.

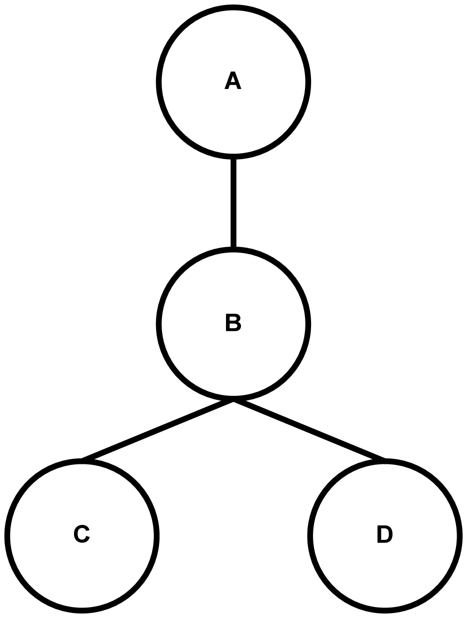

Figure 5. PST generated from Figure 4

2.5. Data-Flow Information

Data-flow analysis investigates how the data contained in memory cells (or registers, or variables for all that matters) is used and modified across the program's operations. In data-flow terminology, we say that a variable is *used* when it is referenced by an operation, and *defined* when its data is modified. The flow of definitions and uses of a variable through its lifetime is often called the def-use chain.

One of the most interesting properties revealed by data-flow analysis is the data dependencies existent between sequentially ordered operations. One notable application that can make use of this information is instruction ordering. These data dependencies can be visualized in the form of data-flow graphs (DFGs), seldom called data-flow diagrams. Figure 6 shows the DFG for Code 1. It is important to notice that it was chosen to represent the result of the comparison operation '==' as if it was stored in an implicit temporary variable, here denoted by '$'. Also, the 'subtract' function was decomposed into a simple subtraction operation.

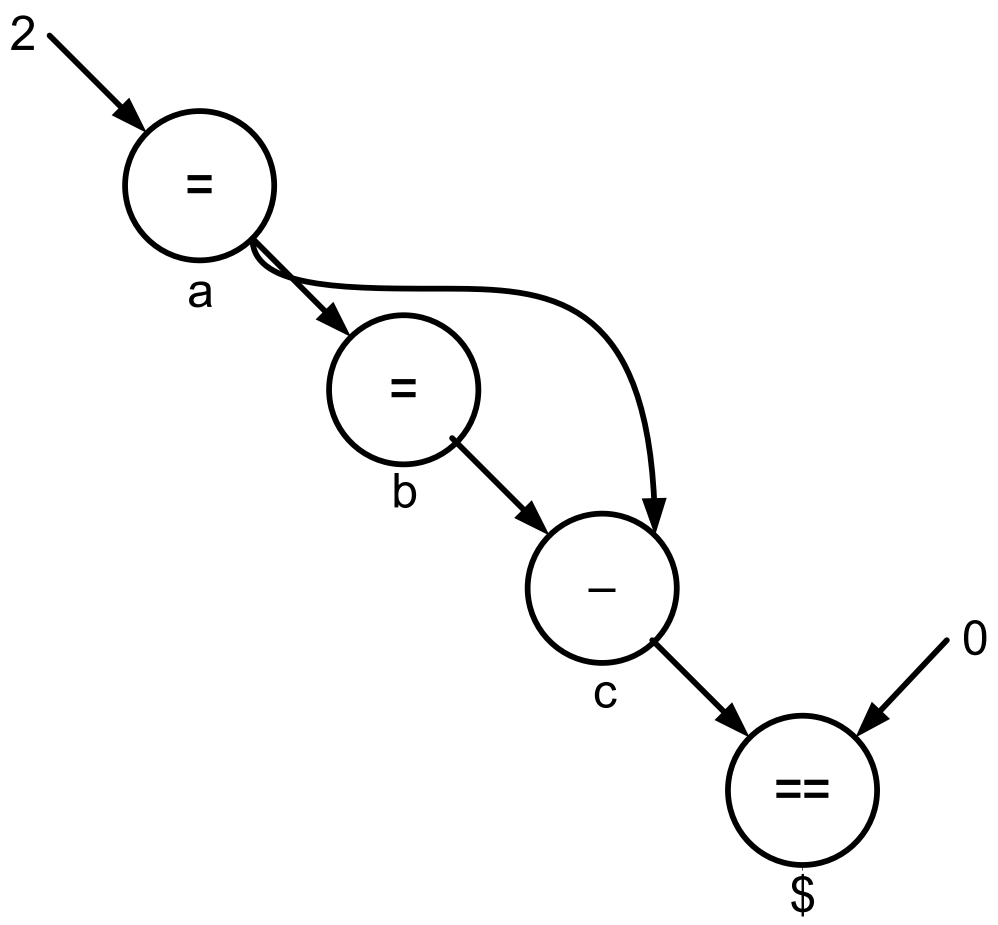

Figure 6. Data-Flow Graph for Code 1

2.6. SSA and SSI

Amongst the intermediate representations oriented towards data-flow analysis, possibly the most famous is the Static Single Assignment (SSA) form [CFR+91]. The main characteristic of the SSA form is that, for every redefinition of a virtual register, i.e. a variable, in the original program, a new variable is created (normally the new variable is named with an incremental number subscripted to the original variable name) in the SSA representation. Therefore, the transformed program will end up with multiple newly-created variables for each (re)definition of the original variable, and each of these new variables is assigned to exactly once.

When computing the SSA form of a program, an interesting question arises: what to do when control-flow reaches a join point and the merging control-flow branches have different 'versions' of a given variable (i.e. the variable has been independently modified by one or more of the different branches)? Which version of the variable should the CFG nodes subsequent to the join point use? To solve this issue, SSA introduces the ϕ-function. Semantically, the ϕ-function is a special form of assignment that 'decides' which version of the variable to use, representing the possible values this variable could assume in runtime.

Figure 7 illustrates the use of the SSA form and ϕ-functions given the simple excerpt of C-style code named Code 2.

```
int a = 3;
int test = 1;

if (test == 0)
  a = a + 1;
else
  a = a + 2;

int b = a;
```

Code 2. Simple piece of C-style code

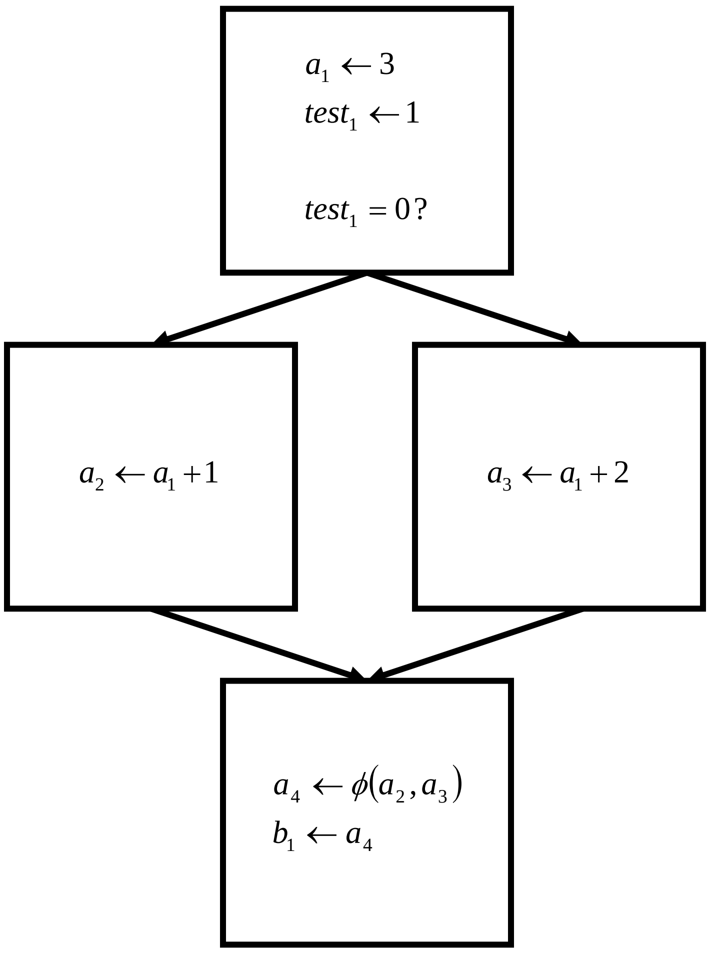

Figure 7. SSA representation of Code 2

Traditional methods of computing the placement of ϕ-functions [CFR+91] are based on finding the *dominance frontier* of each node in the CFG. For a variable in question (suppose it is the only variable in the program), a node Y is said to be in the dominance frontier of a node X if, given the nodes in the path $X \rightarrow Y$, Y is the first node *not* dominated by X and X is the last node to have defined the given variable. These traditional methods of transforming a program into the SSA form have $O(N^2)$ complexity (where N is the number of nodes in the CFG). However, experiments with the optimization techniques offered by the PST features afore mentioned [JPP94] successfully increase the performance of this task.

Out of the many existing SSA variants, one of the most interesting to this work is the Static Single Information (SSI) form [Ana97]. SSI uses the same scheme of virtual register renaming as SSA to guarantee that, in its representation, the program text contains only one assignment of any given variable. However, in addition to the rule of renaming virtual registers due to multiple assignments and to the use of the ϕ-function to manage multiple reaching definitions at merge points, SSI introduces the σ-function. Being placed at control-flow split points, instead of merge points, σ-functions behave essentially as a counterpart to the ϕ-function. A σ-function should be used whenever one or more branches of the control-flow split use a given variable. Semantically speaking, the σ-function takes one source operand, namely the virtual register used in the following branches, and assigns its value to N destination operands, where N is the number of branches starting at the split point. The successor branches then can make use of its respective exclusive newly-created virtual register. Figure 8 depicts the SSI form of program Code 2.

SSI allows for efficient predicated and backward data-flow analyses and is similar to SSA in terms of size and time of computation [Sin05]. Therefore, SSI proves itself to be a good choice, among the modern IRs, for the purpose of static data-flow analysis.

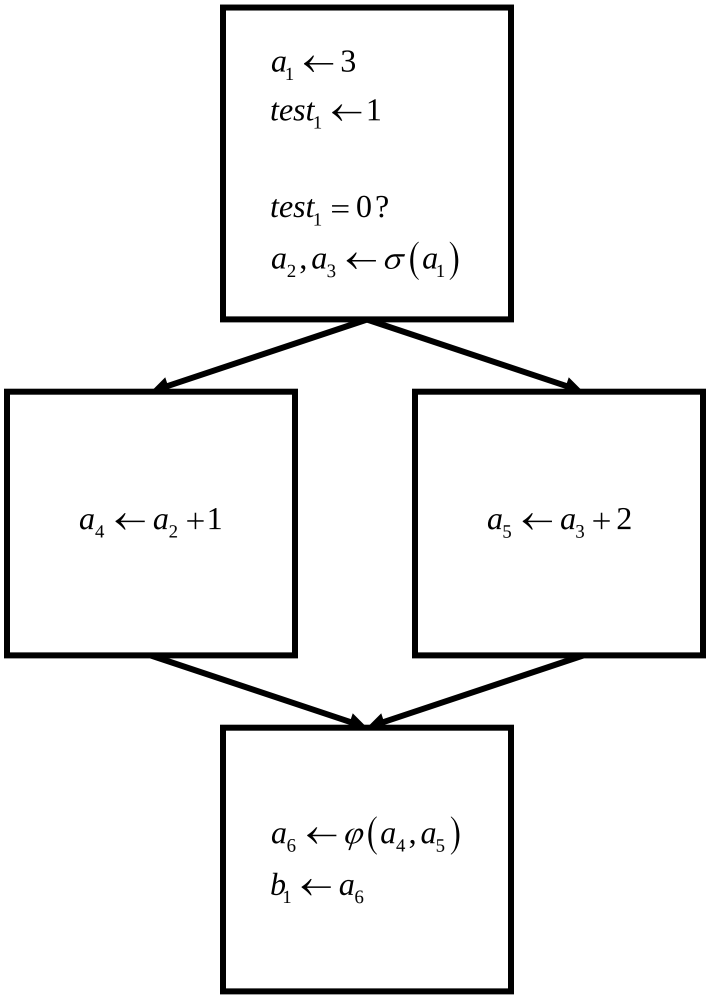

Figure 8. Code 2 in its SSI representation

2.7. Program Semantics

Programming language designers often find themselves facing the difficult task of describing the dynamic semantics of languages, i.e. what the commands of a given programming language effectively do. These semantic descriptions can be useful for a variety of purposes, including serving as reference for programmers or compiler writers. Furthermore, as the semantics of a language can be used to clarify the semantics of a program, one can be interested in studying these semantics in order to take advantage when designing program analyzers.

In contrast to the problem of describing language syntaxes, which is now fairly well-satisfied given the power of the current resources such as the (Extended) Backus-Naur Form ((E)BNF), no formal notation for describing semantics of programming languages is universally accepted and formalizing language semantics is still considered a hard job, with few practical and useful examples applied to complex modern languages. In this section we shall overview some of the methods for describing dynamic semantics of programs.

2.7.1. Axiomatic Semantics

Axiomatic semantics was first introduced by Hoare [Hoa69], after the work of Floyd [Flo67], and is closely related to Hoare's Logics. The idea behind axiomatic semantics is that program statements are enclosed by assertions of preconditions and postconditions. The preconditions establish constraints on the machine state before the execution of the given statement, while the postconditions obviously constrain the state of the machine after the execution of the statement.

Specification of axiomatic semantics for a given statement/program is done using the notation P{Q}R, where P is the precondition, Q is the program and R is the postcondition. This notation means that if P evaluates to true before Q executes, then R will evaluate to true after the execution, provided Q completes.

More than describing semantics, axiomatic semantics are an intuitively natural tool for proving correctness of programs. The type of correctness verifications that axiomatic semantics allow for is called *partial* correctness, i.e. correctness in the finite cases. Referring to the notation above, nothing can be proved if Q never finishes.

When doing correctness proofs, there must be an axiom (i.e. a condition which must be true) or an inference rule for each program statement. Inference rules will propagate the values of conditions across compound statements. From that point on, a theorem proving software must verify, taking the provided axioms into consideration, the conditions and inference rules associated with the program in order to find inconsistencies.

2.7.2. Denotational Semantics

The formalization of denotational semantics became popular in the 1970s [Sto77], following the work of Dana Scott and Christopher Strachey, and relies basically on constructing mathematical objects and creating functions that can map instances of an entity of a language (or program) into these mathematical objects. When expressing the semantics of programming languages with denotational semantics, there has to be a mapping to a mathematical object for each language construct.

The semantics is said to be denotational because these mathematical objects *denote* the meaning of their corresponding syntactic entities. In this case, the most important fact is that there are rigorous and efficient ways of manipulating mathematical objects, while the same does not hold true for programming language constructs.

2.7.3. Operational Semantics

With operational semantics, the meaning of the program is described in terms of the behavior of the machine it executes on, which can be real or simulated. Due to the fact that many programming languages can be executed on a variety of machine models, it is often used an abstract machine that conserves only a subset of the properties of real-world computers, such as the organization of memory cells, the existence of a stack in memory, and so forth.

The first significant employment of the operational semantics formalism was made by a group of scientists at the IBM research lab in Vienna [LW69]. They used operational semantics to describe the semantics of the PL/I language. The methodology they developed was named the Vienna Definition Language (VDL) and included the notation and the abstract machine used. Due to its high complexity, however, this description was of little practical use and VDL, not having made many adopters, was soon obsolete.

After IBM's VDL and to this day, one of the most notable contributions to the description of operational semantics was the introduction of Structural Operational Semantics (SOS), by Gordon Plotkin [Plo81]. SOS proposes the description of operational semantics by means of transition rules in a notation commonly used in logics. An example of how a rule would look in SOS is the following:

$$\frac{(rhs,\sigma)\xrightarrow{e}v}{(mk-Assn(lhs,rhs),\sigma)\xrightarrow{s}\sigma\dagger\{lhs\mapsto v\}}$$

Where $\xrightarrow{e}: Expr\times\Sigma\rightarrow Value$, $\xrightarrow{s}: Stmt\times\Sigma\rightarrow\Sigma$, and Σ denotes an abstraction for memory stores, e.g. the collection of mappings $Id\rightarrow Value$. This notation establishes deduction rules. The predicate in the conclusion (that is, the one below the horizontal line) holds if the premises (the statements above the horizontal line) are valid. The particular rule in the example above describes the operation of an assignment (*mk-Assn*) statement. Here, *rhs* means the expression in the right-hand side of the assignment, and *lhs* means the identifier in the left-hand side. The lower sigma is an instance of the capital sigma, i.e. a representation of the memory space of the program in question. The cross sign introduces the changes, enclosed in the curly brackets, to happen in the memory space.

The notation used in SOS maintains an acceptable level of complexity as it addresses issues such as the description of complex commands and constructs like functions and classes, as well as dealing with questions such as type-checking and error-checking. SOS revived the interest in operational semantics and its viability as way of describing the meaning of languages and programs.

3. The ERESI Project

The ERESI Reverse Engineering Software Interface is a unified multi-architecture binary analysis framework enhanced for UNIX operating systems based on the Executable & Linking Format (ELF) such as Linux, *BSD, Solaris, Cisco IOS, IRIX and BeOS. It can be qualified as *hybrid*, in the sense that it includes both static and dynamic runtime capabilities. ERESI has an evolving language conceived with reverse engineering in mind since its early designs, making it programmable and adaptable to the precise needs of its users.

The current ERESI package includes three applications that interface with the user via command-line, which then invokes the framework's language interpreter. These applications offer a variety of functionalities, categorized into one of the following: debugging, binary file instrumentation or procedure tracing. At the time of writing, two new applications are under development. One of them, already at an advanced stage, shall provide features for kernel debugging. The other, of special interest to this work, is intended to offer the most advanced binary analysis primitives of the project.

3.1. ERESI Internals Overview

Supporting the ERESI applications, seven original libraries spread across the lower layers of the framework's architecture. Figure 9 depicts these layers and their respective components. A brief description of each of them (in top-down order, compared to the architectural model) shall follow.

It is important to notice that the components are designed to feature the largest degree of disassociation among them as possible. Given this modularity, the framework can be tailored to the user's needs by removing unused components or plugging custom components on top of existing libraries. In fact, the lowest-level components, such as libasm, can be linked against other applications completely independently of the rest of the framework.

Applications:

- The ELF shell (elfsh), an interactive and scriptable component dedicated to instrumentation of ELF binary files.
- The embedded ELF debugger (e2dbg), an interactive and scriptable high-performance user-land debugger that works without standard debug API (namely without ptrace).
- The embedded ELF tracer (etrace), an interactive and scriptable user-land procedure tracer that works at full frequency of execution without generating traps.

Applications at earlier development stages:

- The kernel shell (kernsh), an interactive and scriptable user-land shell created to inspect and modify kernel structures by defining them as types of the ERESI language.
- Evarista, a static binary analyzer written almost entirely in ERESI language and devoted to automatic bug-finding. This

component will be in charge of making the needed program transformations, analyses and verifications in order to extract meaningful information from raw binary programs.

Libraries:

- Libe2dbg, library that contains the majority of the code that implements the functionalities of the embedded ELF debugger (e2dbg), being linked to the debuggee process at load-time.
- Librevm, the Reverse Engineering Vector Machine, which contains the meta-language interpreter and the standard ERESI library.
- Libmjollnir, the flow analyzer and code fingerprinting library.
- Libelfsh, the binary manipulation library on which ELFsh, E2dbg, and Etrace are based.
- Libasm, the disassembly engine that gives semantic attributes to instructions and operands.
- Libedfmt, the ERESI Debug Format library, which can convert debug information in the dwarf or stabs formats into the ERESI debug format by automatically generating new ERESI types.
- Libaspect, library responsible for the type system and the aspect-weaving features (reflection). It defines complex data types, making them available to be manipulated ad-hoc by ERESI programs.

It is important to elucidate better the *modus operandi* of the ‘embedded’ applications, e2dbg and etrace. In a nutshell, they work by injecting themselves into the target binary file, so that when it is executed, the application embedded within them, i.e. one of the two ERESI-based applications afore mentioned, will take control of the process. This is made possible by complex injection and relocation techniques provided by libelfsh, and also present in the form of commands of the ELF shell. Not only this speeds up the overall performance of the application, because it is running directly from inside the address space of the process, but it also enables the possibility of even using applications of this kind in scenarios that forbid the use of similar tools that rely on a standard debugging API (namely, ptrace).

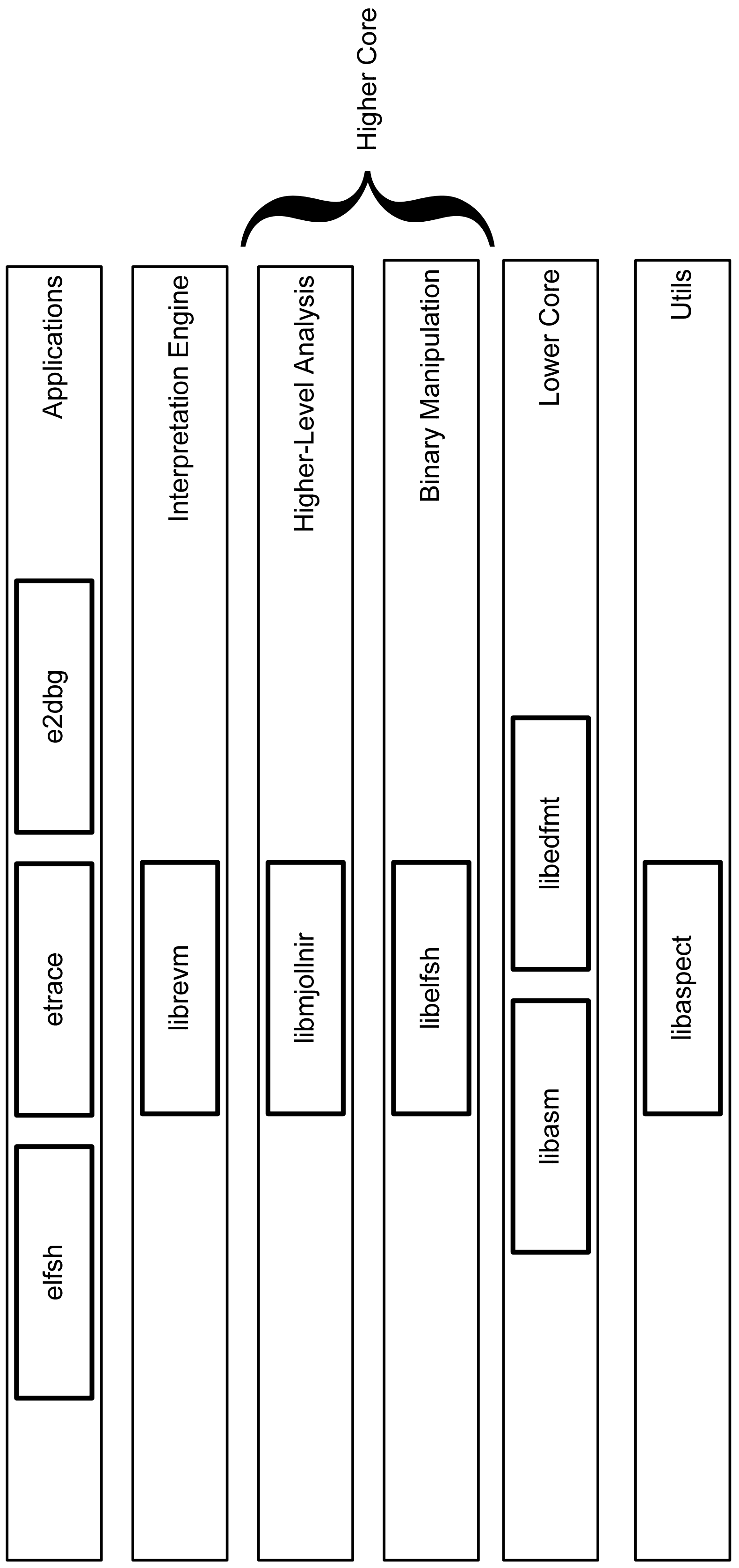

Figure 9. Layered representation of the architecture of the ERESI Project

3.2. The ERESI Language

The ERESI language, which may as well be simply called ERESI, is one of the most innovative features of the framework. It is a domain-specific meta-language dedicated to reverse engineering and analysis of binary programs and is aimed at providing its users with a way of quickly and easily developing applications such as analyzers or decompilers. It is said to be a *meta-language* because it manipulates programs written in another language, namely the machine code language.

Although the ERESI language is a very interesting concept that could be detailed and discussed for many pages, this discussion will be restricted to a brief explanation of the meta-language commands that will be necessary to understand the program transformations explained in later sections of this document. Table 1 summarizes these commands.

Command name	Command description
type	Declares new complex types
define	Defines constants and identifier aliases
load	Loads a binary to be manipulated
reflect	Builds the list of blocks of the loaded binary
set	Assigns a value to a variable
transform	Transforms the program according to the user's specification

Table 1. ERESI commands used in the program transformation code

Given the transformation code in the appendices of this document, which can be used as reference of examples of the use of the commands in Table 1, a formal description of these commands' syntax is unnecessary and beyond the scope of this work. It should be noted, however, that the body of a *transform* construct is formed by a set of *case* statements, which compose the actual specification of the transformation. Besides performing the transformation, the *case* statements also support the execution of an optional statement, which may denote side-effects, such as changes in the data-flow information of the program in question (ERESI also features commands for data-flow analysis, which are not covered here). A *transform* block is ended by an *endtrans* statement, which executes a final command (e.g. list iteration) before executing the transform command again, in a cyclic fashion.

Variables in ERESI are preceded by the dollar sign ($), similarly to many other languages. Furthermore, ERESI makes available a special variable, namely *$_*, which stores the return of the last command executed. This variable is especially useful to access the return of commands that do not take a destination parameter, e.g. the unary commands.

3.3. Libasm

Libasm is the disassembling library of the ERESI project. Besides performing the basic disassembly, libasm features some low-level analysis-related aspects and many other characteristics that turn this library into a very special disassembling library, *sine qua non* to a variety of features of the ERESI framework.

Due to its undeniable importance, allied to the fact that this is the component at which most of this work's efforts were targeted, it becomes essential that the features of libasm are better detailed.

3.3.1. Multi-Architectural Support

Libasm currently supports two machine models: SPARC v9 and Intel IA-32. Although these architectures are certainly two of the most widely used on modern personal computers, at the time of writing there are also ongoing efforts to make it support MIPS machines. Following this trend of extending ERESI features to embedded systems, future work holds the possibility of porting libasm to support ARM processors too.

The current SPARC v9 support is capable of disassembling any of the instructions in the architecture's instruction set and representing them as structures internal to the library. As the SPARC architecture is almost totally backwards compatible with its previous versions, the v9 support includes the capability of correctly disassembling nearly all of the constructs of v8 and v7 machine code.

Although the IA-32 port does not support the disassembling of some instructions (in particular, some of the instructions belonging to the newest SSE and MMX extensions), the support is extensive enough so that libasm can effectively disassemble a wide range of binaries present in many flavors of UNIX variants, such as the applications contained in the /bin and /usr/bin directories of default installations of the supported operating systems. Having that said, the IA-32 port of libasm is constantly evolving and more instructions become supported upon demand.

3.3.2. Semantic Annotation of Instructions and Operands

Libasm holds an internal representation of the disassembled instructions and operands. This representation, besides containing all the syntactic information needed to output an instruction correctly, carries semantic annotations (attributes), gathered at disassemble-time, about instructions and operands. These semantic attributes are summarized on Tables 2 and 3 for instructions and operands, respectively.

Attribute	Description
IMPBRANCH	Branching instruction which always branch (jump)
CONDBRANCH	Conditional branching instruction
CALLPROC	Sub Procedure calling instruction
RETPROC	Return instruction
ARITH	Arithmetic (or logic) instruction
LOAD	Instruction that reads from memory
STORE	Instruction that writes in memory
ARCH	Architecture dependent instruction

WRITEFLAG	Flag-modifier instruction
READFLAG	Flag-reader instruction
INT	Interrupt/call-gate instruction
ASSIGN	Assignment instruction
COMPARISON	Instruction that performs comparison or test
CONTROL	Instruction modifies control registers
NOP	Instruction that does nothing
IO	Instruction accesses I/O locations (e.g. ports)
TOUCHSP	Instruction modifies stack pointer
BITTEST	Instruction investigates values of bits in the operands
BITSET	Instruction modifies values of bits in the operands
INCDEC	Instruction does an increment or decrement
PROLOG	Instruction is part of a function prolog
EPILOG	Instruction is part of a function epilog
STOP	Instruction stops the program

Table 2. Semantic attributes used for instructions in libasm

Attribute	Description
REG	Register operand
IMM	Immediate value
MEM	Memory Access

Table 3. Semantic attributes used for operands in libasm

These semantic attributes allows code from upper layers of ERESI (including programs written in ERESI code) to refer to instructions collectively, in terms of their semantics annotations, instead of individually. As will be seen later in this document, this simplifies significantly the task of writing a back-end for program transformation of a specific machine code into the LIR used in the project. Similarly, many other applications performing instrumentation or analysis may benefit from this feature.

It is very important to notice that the semantic attributes listed here are not mutually-exclusive, i.e. libasm can annotate an instruction with more than one attribute. This obviously extend the possible classes of instructions to those beyond the simple attributes provided. While the meaning of most of these combinations should be fairly intuitive to the reader, some of the less intuitive ones should be highlighted in order to clarify the process of classifying instructions:

- READFLAG + BITTEST – This combination is used to annotate instructions that perform some kind of conditional test (that is, test the value of a flag) and is not a conditional branching instruction. This behavior should not be confused with COMPARISON, which describes instructions that explicitly perform some kind of test and *set* the necessary flags. Conditional moves, for example, are READFLAG + BITTEST.

- IO + LOAD/STORE – Specifies reads/writes from/to I/O locations, such as ports.
- INT + RETPROC – Returns from interrupts or traps.
- TOUCHSP + LOAD/STORE – Stack pops/pushes.

It should also be stressed that the same semantic attributes are available to all ports of libasm. This unified system allows for the development of code that analyzes programs directly on top of their disassembled representation (i.e. not on top of any IR) in an architecture-independent manner. This is, for instance, how some of the current code in libmjollnir works.

Finally, we end our discussion of the semantic attributes by adding that instructions annotated with READFLAG and/or WRITEFLAG have their representation enriched with information about which flags (specific to each machine model) are possibly read or modified.

3.3.3. Vectors of Handlers

In its source code organization, libasm splits the code that disassembles instructions or operands into different functions, one function for each instruction (known as opcode handlers or instruction handlers) and one function for each operand 'type'[1] (known as operand handlers). The function pointers to these instruction and operand handlers are then stored into vectors.

Vectors are multi-dimensional data structures provided by libaspect. Every dimension of a vector corresponds to a parameter to which the data contained in the vector is associated. A compositional association of values to these parameters serves as an index into the vector, and is used for storing and recovering data from the vector. For example, a vector containing instruction handlers could recover a specific handler if provided an identifier to a machine model/architecture and the opcode value of the instruction desired.

One of the most interesting characteristics of vectors is that they have their structure and contents accessible from ERESI language. This feature, inspired from the concept of reflection of aspect-oriented systems, ultimately means that function pointers to instruction and operand handlers can be retrieved and modified by the ERESI user, who can in turn easily write simple instruction and operand tracers by updating these records.

[1] Operand 'types' are not real types, as in type theory. Instead, they are syntactic attributes, which specify the format/encoding of operands and are machine-dependent. These 'types', not to be confused with the operand semantic attributes, will not be further discussed here.

4. Contributions of this Work

The author of this work has contributed to the ERESI project in many ways before the writing of this document. Attacking bugs and building new features everywhere, the contributions have ranged from the lowest-level and most central pieces of code of the disassembling library to the construction of control-flow graphs and their graphical representation to the end user.

For the purpose of this undergraduate project, however, it was chosen to detail the development process that led to the goal set specifically for this work and done during the literary research and conception of the present document: the transformation of Intel IA-32 machine code into the ERESI low-level intermediate representation (LIR).

4.1. Transforming IA-32 Machine Code into ERESI LIR

IA-32 is the name of the computer architecture created and maintained by the Intel ® Corporation. This architecture gave birth to many families of processors since the inception of the first chips, 8086/8088, in 1978. Across these decades of development, IA-32 processors became increasingly popular and today these chips undoubtedly represent a big fraction of the processors powering modern computers (particularly successful in the market of desktops and workstations).

Due to its popularity, supporting IA-32 code in the analyses to be performed by ERESI is an obvious good way to clarify their practical relevance. Therefore, it seemed equally natural that this goal should be kept in mind since the earliest developments of the analysis features and, consequently, it was decided that this work should focus on executing the first step of the binary analysis process: transforming raw IA-32 machine code into the LIR used in the ERESI framework.

Beyond reporting the work done, the following sections may serve as guidance for other developers and users willing to write back-ends for machine code transformation into ERESI LIR for other computer architectures or, to some extent, even writing other kinds of program transformations on top of ERESI.

4.1.1. The Instruction Set

The scope of this work restricts itself to transform the instructions contained in the instruction set of the original 8086/8088 processors. The only instructions not covered in this project are the ESC instruction (a gate to the instructions provided by the Floating-Point Unit) and the prefix instructions, LOCK and REP/REPxx. The full listing of the remaining instructions, i.e. the instructions that this work covers, can be found in the appendices.

In contrast to the afore mentioned exceptions, this work probably adds support to many other instructions that do not belong to the original set. In particular, there was a beneficial and intentional disregard around the exact operand formats supported in the basic instruction set. That is, regardless of whether a instruction such as MOV, for example, in the 8086/8088 specification, only existed in its form where it accepts two registers as input (just an hypothetical case, not necessarily true), the transformation back-end was written in a way such that it can handle other forms of the MOV instructions, such as the one that receives a register and a memory operand, or a register and an immediate operand and so on. It was added to the back-end every instruction form supported by the opcode and operand handlers code in libasm at the time of development. As these different forms of a same instruction, in the machine code, are represented by different opcodes, they can technically be considered different instructions in themselves. In this way, it can be said that the transformation back-end handles some instructions that are not part of the original 8086/8088 instruction set.

4.1.2. Semantically Annotating the Instructions

After narrowing down the instruction set, the immediate step was to annotate each instruction with the semantic attributes listed on Table 2, by modifying the code contained in the opcode/instruction handlers. Similarly, the operand handlers must contain the code to accurately categorize the operands involved. This task shall make use the operand attributes from Table 3. The knowledge required to execute this step came essentially from two sources: 1) the instruction set reference [Int2A05, Int2B05] or similar literature and 2) from previous knowledge on assembly programming, reverse engineering and related subjects.

The reference manuals are the basic resource to instruction classification. It is very important that the reference used, no matter what it should be, is as accurate and complete as possible. As far as the IA-32 architecture goes, the official manuals proved themselves a reliable resource. They contain both a textual description and a primitive description of 'operational semantics', which are actually listings of pseudo-code describing the microcode behavior. Due to the notable complexity of the IA-32 architecture (when compared to modern RISC machines), though, the reference manuals are a large piece of literature split into multiple volumes, which makes the seeking for topics a bit harder.

There are some things that the manuals do not make explicit, however, and it may take a little bit of previous expertise not to let it go unnoticed. This is the case, for example, of the use of the PROLOG and EPILOG semantic attributes. To identify the correct cases where instructions are performing functions prologs and epilogs, one must know in advance some assembly programming or, most importantly, how compilers typically write function prologs and epilogs. Although this may look like a somewhat imprecise methodology, a careful evaluation and implementation of this step may reduce the number of false-positives and false-negatives to very low rates, when analyzing compiler-generated code.

4.1.3. Writing the Transformation Back-End

Once the instructions are accordingly annotated, we can proceed by transforming them into the LIR constructs. The importance of having the annotations lies on the fact that, armed with them, we can address the instructions by their semantic attributes, instead of referencing them individually. This of course reduces significantly the amount of cases explicitly defined in the source code.

The code of the transformation was written in ERESI and can be found in the appendices. Additional (though necessary for the code execution) definitions, such as the LIR constructs and the internal types of libasm, can also be found in the appendices. The transformation code should be run by the Evarista analyzer, still under construction.

The targets of the transformation are the LIR constructs, which represent common operations such as assignments and arithmetic operations in a generic manner. Table 4 lists the LIR constructs.

LIR Construct	**Description**
Ins	Generic Instruction
IndBranchR::Ins	Indirect branch (register operand)
IndBranchM::Ins	Indirect branch (memory operand)
Branch::Ins	Direct branch
Call::Ins	Procedure call
IndCallR::Ins	Indirect procedure call (register operand)
IndCallM::Ins	Indirect procedure call (memory operand)
Return::Ins	Return from procedure
TernopR3::Ins	Ternary operation (arithmetic or logic)
TernopM3::Ins	Ternary operation (arithmetic or logic)
TernopRI::Ins	Ternary operation (arithmetic or logic)
TernopMI::Ins	Ternary operation (arithmetic or logic)
TernopMR::Ins	Ternary operation (arithmetic or logic)
TernopRM::Ins	Ternary operation (arithmetic or logic)
TernopRMI::Ins	Ternary operation (arithmetic or logic)
AssignRI::Ins	Assignment operation
AssignMI::Ins	Assignment operation
AssignRM::Ins	Assignment operation
AssignMR::Ins	Assignment operation
AssignMM::Ins	Assignment operation
AssignRR::Ins	Assignment operation
IoRR::Ins	I/O access operation
IoIR::Ins	I/O access operation
IoRI::Ins	I/O access operation
BitSet::Ins	Operation that sets a bit in the operand
CmpRI::Ins	Comparison or test operation
CmpRR::Ins	Comparison or test operation
CmpRM::Ins	Comparison or test operation
CmpMR::Ins	Comparison or test operation
CmpMI::Ins	Comparison or test operation

XchgRR::Ins	Value exchange operation
XchgMR::Ins	Value exchange operation
Prolog::Ins	Function prolog
Interrupt::Ins	Interrupt or trap
IReturn::Ins	Return from interrupt or trap
Epilog::Ins	Function epilog
Stop::Ins	Stop execution
Nop::Ins	No operation
FlagR::Ins	Flag-reading operation
FlagW::Ins	Flag-writing operation

Table 4. ERESI LIR Constructs

The R/M/I suffixes on some constructs, such as *Ternop* and *Assign*, denote the types of the destination and source operands they receive. *Ins* is a generic construct, from which all other constructs inherit. It contains an ‘addr’ attribute, which specifies the address of the instruction corresponding to the operation in question, and the ‘uflags’ attribute, which records information about any relevant reading or writing of flags performed by the instruction. The LIR definition in the appendices displays all of the operands (and their type) that each construct takes.

At first the LIR was not ready to support the transformation from IA-32 code, so it was necessary to add many new constructs to the LIR definition. The majority of these new constructs was created to represent the great diversity of combinations of operand types that IA-32 features. Although the addition of these constructs make the LIR look more complex, this added complexity ultimately turns into an aid for the simplicity of the data-flow analysis code, since the types of the operands can be known in advance from the name of the construct, instead of testing for overloaded constructs or another complex mean of making the LIR definition smaller.

Finally, when the LIR is all ready to represent the machine code, the transformation code can in fact be written. The heart of the transformation code is the *transform* command, which takes one source instruction and applies the first matching transformation found in the *case* statements. The following is an example of the use of the *transform* command (taken from the real IA-32-to-LIR transformation code):

```
transform $instr into
...
# IN (reg, reg)
case instr_t(type:io-rm, op1(type:reg)) ->
  IoRR(addr:$curaddr,
  dst(id:$instr.op2.baser),
  src(id:$instr.op1.baser), uflags:0)
...
endtrans
```

Code 3. Example of transformation code in ERESI

In the excerpt of code above it can be seen that the left-side of the arrow is the specification of the source of the transformation, while its right-side represents the instantiation of a construct in the LIR. It is important to notice that, when specifying the transformation, the most specialized cases must be placed on top, since the program will execute the first matching transformation. That is, if a *case* specifying “the first assignment operation” is placed before one specifying “the first assignment operation between registers”, the transformation specified by the second *case* will never be triggered, since the *transform* command searches for the first matching specification sequentially, in top-down order.

In the transformation code, the *transform* command is placed in the body of a loop, which executes the transformation for each instruction in the program. As previously stated, one of the most interesting characteristics of the transformation code is its ability of specifying a transformation for many source instructions in one single *case*. The following *case* is an example of this aspect:

```
# Jxx, LOOP, LOOPE, LOOPNE (imm)
# Jxx = JA, JAE, JB, JBE, JE, JECXZ, JG, JGE,
# JL, JLE, JNE, JNO, JNP, JNS, JO, JP, JS

case instr_t(type:cb) -> Branch(addr:$curaddr,
  dst(val:$instr.op1.imm), uflags:0)
```

Code 4. Many-to-One transformation in ERESI

In this *case* we can clearly see how the transformation code benefits from the semantic annotations made earlier. By referring to a group of instructions simply by their semantic attributes (in this case, the CONDBRANCH one), a transformation can be performed on a many-to-one basis. The complete code of the IA-32 transformation back-end, found in the appendices of this document, relies on this ability to specify many of the transformation *cases*.

4.2. *Future Work*

The analysis features of the ERESI project are rapidly evolving and still on unstable stage of development. As such, this work is by no means perfect and at definitive state. Consequently, the most obvious chances of future work are improvements on this particular piece of work itself.

A very natural improvement of this project is the extension of its range of source instructions to cover instructions added to the IA-32 instruction set upon release of processor chips successor to the 8086/8088. This is an immediate task that should start soon after this work.

Additionally, the LIR may go through many revisions in the future. It is important that the ERESI project has a LIR that satisfies the current and future needs of the program analysis features and, being the LIR a recent concept within the project, it is probable that it may have many possibilities of improvement. Similarly, the choices of the semantic annotations implemented inside libasm are extremely important for the soundness of the transformation, so they may be reviewed in the future as well.

Finally, but not lastly, this work can be further extended by writing back-ends for other machine architectures. As the development of the MIPS porting of libasm advances, the opportunity of writing a transformation back-end for this architecture becomes more and more desirable. And, of course, the development of the Evarista analyzer should not stop there. The near future holds opportunities for developing and transforming the code into higher-level IRs, implementing more advanced analyses, writing effective verification algorithms, etc. This work is definitely just the tip of the iceberg.

5. Conclusions

This work has attempted to shed some light into the still obscure field of binary analysis. The report included a brief discussion on the importance of this work as well as a fairly broad introduction to some of the most relevant topic on program analysis. Subjects such as intermediate representations, control-flow and data-flow analyses, and program semantics were discussed.

We have introduced the ERESI project, telling a little of its history, architecture, each of its components and its own domain-specific language. Special focus was given to the libasm component, the disassembling library of ERESI, as it represented an especially important component for the purposes of this work. We have detailed how libasm organizes its code in reflective vectors, support multiple architectures and, most importantly, how it gives semantic annotations to disassembled instructions and their operands.

This document also reported the development process of the primary goal of this project: transforming Intel® IA-32 machine code in the LIR used by the ERESI framework and ultimately consumed by the Evarista analyzer. We have dealt with the choosing of the source instruction set and the semantic annotations of the instructions in question, highlighting the most important steps and precautions in executing these tasks. Naturally, we have also covered the actual transformation of the machine code into LIR, describing the format of the LIR used in the project and how to write the transformation code in ERESI language. At this point, it was appropriate to stress the link between this task and other subjects discussed in this report. Particularly, we reinforced how the transformation code benefits from libasm's feature of semantic annotations.

Finally, it is expected that, more than an undergraduate project, this report and the work associated with it represent real advances for the ERESI project and for the field of binary analysis in general. It has been a great opportunity for learning and hopefully it will also be a pioneering contribution to the pursuit of an objective that, although of obvious relevance, so few have adventured on: extracting meaningful and pertinent information from programs in their binary form.

References

[Ana97] C. Scott Ananian. The Static Single Information Form. Ph.D. Thesis, Massachusetts Institue of Technology, September 1999.

[BJ66] Corrado BÖhm and Guiseppe Jacopini. Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules. Communications of the ACM 9(5):366-371, May 1966.

[CFR+91] R. Cytron , J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems, 13(4):451-490, October 1991.

[Flo67] Robert W. Floyd. Assigning Meanings to Programs. Proceedings of Symposium in Applied Mathematics, 19:19-32, 1967.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. Communications of the ACM 12(10):576-580, October 1969.

[Int2A05] Intel® Corporation. IA-32 Intel® Architecture Software Developer’s Manual, Volume 2A: Instruction Set Reference, A-M. Intel® Corporation, June 2005.

[Int2B05] Intel® Corporation. IA-32 Intel® Architecture Software Developer’s Manual, Volume 2B: Instruction Set Reference, N-Z. Intel® Corporation, June 2005.

[ISS07] IBM® ISS X-Force. IBM® Internet Security Systems X-Force 2006 Trend Statistics. Whitepaper, January 2007.

[Jon03] Cliff B. Jones. Operational Semantics: Concepts and their Expression. Information Processing Letters 88(1-2):27-32, October 2003.

[JPP94] R. Johson, D. Pearson, and K. Pingali. The Program Structure Tree: Computing Control Regions in Linear Time. In Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation, pages 171-185, Orlando, U.S., June 1994.

[LW69] P. Lucas and K. Walk. On The Formal Description of PL/I, volume 6 of Annual Review in Automatic Programming Part 3. Pergamon Press, 1969.

[Luc81] P. Lucas. Formal Semantics of Programming Languages: VDL. IBM Journal of Research and Development 25(5):549-561, September 1981.

[MLB72] M. Marcotty, H. Ledgard, and G. V. Bochman. A Sampler of Formal Definitions. ACM Computing Surveys 8(2):191-276, June 1976.

[Oll07] Gunter Ollman. Counting Vulnerabilities. http://blogs.iss.net/archive/CountingVulns.html

[Plo81] Gordon Plotkin. A Structural Approach to Operational Semantics. Technical Report, Aarhus University, 1981.

[Plo04] Gordon Plotkin. The Origins of Structural Operational Semantics. Journal of Logic and Algebraic Programming 60-61:3-15, July-December 2004.

[Seb02] Robert W. Sebesta. Concepts of Programming Languages, 5th Edition. Peason Education, 2002.

[Sin05] Jeremy Singer. Static Program Analysis based on Virtual Register Renaming. Ph.D. Thesis, University of Cambridge, March 2005.

[Sto77] Joseph E. Stoy. Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics. MIT Press, 1977.

[VGA+07] J. Vanegue, T. Garnier, J. Auto, S. Roy, and R. Lesniak. Next-Generation Debuggers for Reverse Engineering. Black Hat Europe, Amsterdam, Netherlands, March 2007.

Appendix A. Intel 8086/8088 Instruction Set

Instruction	Description
AAA	ASCII adjust AL after addition
AAD	ASCII adjust AX before division
AAM	ASCII adjust AX after multiplication
AAS	ASCII adjust AL after subtraction
ADC	Add with carry
ADD	Add
AND	Logical AND
CALL	Call procedure
CBW	Convert byte to word
CLC	Clear carry flag
CLD	Clear direction flag
CLI	Clear interrupt flag
CMC	Complement carry flag
CMP	Compare operands
CMPSB	Compare bytes in memory
CMPSW	Compare words
CWD	Convert word to doubleword
DAA	Decimal adjust AL after addition
DAS	Decimal adjust AL after subtraction
DEC	Decrement by 1
DIV	Unsigned divide
HLT	Enter halt state
IDIV	Signed divide
IMUL	Signed multiply
IN	Input from port
INC	Increment by 1
INT	Call to interrupt
INTO	Call to interrupt if overflow
IRET	Return from interrupt
Jxx	Jump if condition
JMP	Jump
LAHF	Load flags into AH register
LDS	Load pointer using DS
LEA	Load Effective Address
LES	Load ES with pointer
LODSB	Load byte
LODSW	Load word
LOOP/LOOPxx	Loop control
MOV	Move
MOVSB	Move byte from string to string
MOVSW	Move word from string to string

MUL	Unsigned multiply
NEG	Two's complement negation
NOP	No operation
NOT	Negate the operand, logical NOT
OR	Logical OR
OUT	Output to port
POP	Pop data from stack
POPF	Pop data into flags register
PUSH	Push data onto stack
PUSHF	Push flags onto stack
RCL	Rotate left (with carry)
RCR	Rotate right (with carry)
RET	Return from procedure
RETN	Return from near procedure
RETF	Return from far procedure
ROL	Rotate left
ROR	Rotate right
SAHF	Store AH into flags
SAL	Shift Arithmetically left (multiply)
SAR	Shift Arithmetically right (signed divide)
SBB	Subtraction with borrow
SCASB	Compare byte string
SCASW	Compare word string
SHL	Shift left (multiply)
SHR	Shift right (unsigned divide)
STC	Set carry flag
STD	Set direction flag
STI	Set interrupt flag
STOSB	Store byte in string
STOSW	Store word in string
SUB	Subtraction
TEST	Logical compare (AND)
WAIT	Wait until not busy
XCHG	Exchange data
XLAT	Table look-up translation
XOR	Exclusive OR

Table 5. Restricted Intel 8086/8088 instruction set

Appendix B. Configuration of Common ERESI Types

```
# This file can be appended to your ~/.eresirc

type operand_t    = len:int ptr:*byte type:int name:string size:int
                     content:int regset:int prefix:int imm:int
                     baser:int indexr:int sbaser:string sindex:string
                     address_space:int scale:int
type instr_t      = proc%4 instr:int type:int prefix:int spdiff:int
                     wflags:int rflags:int ptr_prefix:*byte
                     ptr_instr:*byte annul:int prediction:int nb_op:int
                     op1:operand_t op2:operand_t op3:operand_t len:int
type listent_t    = key:string data:long next:*listent_t
type list_t       = head:*listent_t elmnbr:int type:int linearity:byte
type hash_t       = ent:*listent_t size:int elmnbr:int type:int
                     linearity:byte
type container_t  = id:int data:long type:int nbrinlinks:int
                     nbroutlinks:int inlinks:*list_t outlinks:*list_t
type registry_t   = registry:*hash_t
type mjrblock_t   = vaddr:long size:int symoff:int
type mjrfunc_t    = vaddr:long size:int name:byte[64] first:*mjrblock_t
                     md5:byte[34]
```

Appendix C. Script of LIR Definitions

```
#!evarista/evarista32
#lir-definition.esh

# Attributes for ASM instructions
define b   ASM_TYPE_IMPBRANCH
define cb  ASM_TYPE_CONDBRANCH
define c   ASM_TYPE_CALLPROC
define i   ASM_TYPE_INT
define r   ASM_TYPE_RETPROC
define p   ASM_TYPE_PROLOG
define cmp ASM_TYPE_COMPARISON
define bs  ASM_TYPE_BITSET
define bt  ASM_TYPE_BITTEST
define a   ASM_TYPE_ASSIGN
define wm  ASM_TYPE_STORE
define rm  ASM_TYPE_LOAD
define e   ASM_TYPE_EPILOG
define s   ASM_TYPE_STOP
define n   ASM_TYPE_NOP
define ar  ASM_TYPE_ARITH
define wf  ASM_TYPE_WRITEFLAG
define rf  ASM_TYPE_READFLAG
define id  ASM_TYPE_INCDEC
define io  ASM_TYPE_IO
define sp  ASM_TYPE_TOUCHSP

define ar-wf    ar  wf
define ar-id-wf ar  id wf
define cmp-wf   cmp wf
define io-rm    io  rm
define io-wm    io  wm
define i-rf-bt  i   rf bt
define i-r      i   r
define a-rm-wm  a   rm wm
define sp-rm    sp  rm
define sp-rm-wf sp  rm wf
define sp-wm-rf sp  wm rf
define wm-rm    wm  rm
define a-rm     a   rm


# Attributes for ASM operands
define reg  ASM_OPTYPE_REG
define imm  ASM_OPTYPE_IMM
define mem  ASM_OPTYPE_MEM

# Types of LIR operands
type Reg            = id:int
type Immed          = val:long
type Mem            = base:Reg index:Reg scale:Immed off:Immed
name:string

# Types of LIR instructions
```

```
type Ins             = uflags:Immed addr:Immed

type IndBranchR::Ins = dst:Reg
type IndBranchM::Ins = dst:Mem
type Branch::Ins     = dst:Immed
type Call::Ins       = dst:Immed
type IndCallR::Ins   = dst:Reg
type IndCallM::Ins   = dst:Mem
type Return::Ins     = dst:Immed
type TernopR3::Ins   = dst:Reg rsrc1:Reg rsrc2:Reg
type TernopM3::Ins   = dst:Mem msrc1:Mem msrc2:Mem
type TernopRI::Ins   = dst:Reg rsrc:Reg isrc:Immed
type TernopMI::Ins   = dst:Mem msrc:Mem isrc:Immed
type TernopMR::Ins   = dst:Mem msrc:Mem rsrc:Reg
type TernopRM::Ins   = dst:Reg rsrc:Reg msrc:Mem
type TernopRMI::Ins  = dst:Reg rsrc:Reg msrc:Mem isrc:Imm
type AssignRI::Ins   = dst:Reg src:Immed
type AssignMI::Ins   = dst:Mem src:Immed
type AssignRM::Ins   = dst:Reg src:Mem
type AssignMR::Ins   = dst:Mem src:Reg
type AssignMM::Ins   = dst:Mem src:Mem
type AssignRR::Ins   = dst:Reg src:Reg
type IoRR::Ins         = dst:Reg src:Reg
type IoIR::Ins       = dst:Immed src:Reg
type IoRI::Ins       = dst:Reg src:Immed
type BitSet::Ins     = src:Immed dst:Reg
type CmpRI::Ins      = fst:Immed snd:Reg
type CmpRR::Ins      = fst:Reg snd:Reg
type CmpRM::Ins      = fst:Mem snd:Reg
type CmpMR::Ins      = fst:Reg snd:Mem
type CmpMI::Ins      = fst:Immed snd:Mem
type XchgRR::Ins     = fst:Reg snd:Reg
type XchgMR::Ins     = fst:Mem snd:Reg
type Prolog::Ins     = framesz:Immed
type Interrupt::Ins  = dst:Immed
type IReturn::Ins
type Epilog::Ins
type Stop::Ins
type Nop::Ins
type FlagR::Ins
type FlagW::Ins
```

Appendix D. Binary-to-LIR ERESI Code

```
#!evarista/evarista32
#intel-backend.esh

# IA-32 registers
define EAX 0
define ECX 1
define EDX 2
define EBX 3
define ESP 4
define EBP 5
define ESI 6
define EDI 7
define EFLAGS -1

# IA-32 exceptional instruction
define NEG ASM_NEG

# IA-32 prolog instruction
define SUB ASM_SUB

# IA-32 epilog instruction
define MOV ASM_MOV

# This create on-demand the block instruction list in the eresi runtime
reflect $1

set $curblock $_
set $curaddr $curblock.vaddr

# Just debug printing
#inspect $curblock
#profile enable warn

# Start the transformation
foreach $instr in $hash[instrlists:$curaddr]

print Transforming instruction: $instr

transform $instr into

# INC, DEC (reg)
case instr_t(type:ar-id-wf, nb_op:1, op1(type:reg)) ->
     TernopRI(addr:$curaddr, dst(id:$instr.op1.baser),
     rsrc(id:$instr.op1.baser), isrc(val:1), uflags:$instr.wflags)

# INC, DEC (mem)
case instr_t(type:ar-id-wf, nb_op:1, op1(type:mem)) ->
     TernopMI(addr:$curaddr, dst(id:$instr.op1.baser),
     msrc(name:$instr.op1.name base(id:$instr.op1.baser),
     index(id:$instr.op1.indexr), scale(val:$instr.op1.scale),
     off(val:$instr.op1.imm)), isrc(val:1), uflags:$instr.wflags)

# AAA, AAD, AAM, AAS, DAD, DAS
```

```
case instr_t(type:ar-wf, nb_op:0) -> TernopR3(addr:$curaddr,
     dst(id:EAX), rsrc1(id:EAX), rsrc2(id:EAX), uflags:$instr.wflags)

# ADC, ADD, AND, OR, RCR, ROL ROR, SAR, SBB, SHL, SHR, SUB, XOR (reg,
     reg)
case instr_t(type:ar-wf, nb_op:2, op1(type:reg), op2(type:reg)) ->
     TernopR3(addr:$curaddr, dst(id:$instr.op2.baser),
     rsrc1(id:$instr.op2.baser), rsrc2(id:$instr.op1.baser),
     uflags:$instr.wflags)

# ADC, ADD, AND, OR, RCR, ROL ROR, SAR, SBB, SHL, SHR, SUB, XOR (reg,
     imm)
case instr_t(type:ar-wf, nb_op:2, op1(type:imm), op2(type:reg)) ->
     TernopRI(addr:$curaddr, dst(id:$instr.op2.baser),
     rsrc(id:$instr.op2.baser), isrc(val:$instr.op1.imm),
     uflags:$instr.wflags)

# ADC, ADD, AND, OR, RCR, ROL ROR, SAR, SBB, SHL, SHR, SUB, XOR (reg,
     mem)
case instr_t(type:ar-wf, nb_op:2, op1(type:mem), op2(type:reg)) ->
     TernopRM(addr:$curaddr, dst(id:$instr.op2.baser),
     rsrc(id:$instr.op2.baser), msrc(name:$instr.op1.name
     base(id:$instr.op1.baser), index(id:$instr.op1.indexr),
     scale(val:$instr.op1.scale), off(val:$instr.op1.imm)),
     uflags:$instr.wflags)

# ADC, ADD, AND, OR, RCR, ROL ROR, SAR, SBB, SHL, SHR, SUB, XOR (mem,
     imm)
case instr_t(type:ar-wf, nb_op:2, op1(type:imm), op2(type:mem)) ->
     TernopMI(addr:$curaddr, dst(name:$instr.op2.name
     base(id:$instr.op2.baser), index(id:$instr.op2.indexr),
     scale(val:$instr.op2.scale), off(val:$instr.op2.imm)),
     msrc(name:$instr.op2.name base(id:$instr.op2.baser),
     index(id:$instr.op2.indexr), scale(val:$instr.op2.scale),
     off(val:$instr.op2.imm)), isrc(val:$instr.op1.imm),
     uflags:$instr.wflags)

# ADC, ADD, AND, OR, RCR, ROL ROR, SAR, SBB, SHL, SHR, SUB, XOR (mem,
     reg)
case instr_t(type:ar-wf, nb_op:2, op1(type:reg), op2(type:mem)) ->
     TernopMR(addr:$curaddr, dst(name:$instr.op2.name
     base(id:$instr.op2.baser), index(id:$instr.op2.indexr),
     scale(val:$instr.op2.scale), off(val:$instr.op2.imm)),
     msrc(name:$instr.op2.name base(id:$instr.op2.baser),
     index(id:$instr.op2.indexr), scale(val:$instr.op2.scale),
     off(val:$instr.op2.imm)), rsrc(id:$instr.op1.baser),
     uflags:$instr.wflags)

# IMUL (reg, reg, imm)
case instr_t(type:ar-wf, nb_op:3, op1(type:imm), op2(type:reg),
     op3(type:reg)) -> TernopRI(addr:$curaddr, dst(id:$instr.op3.baser),
     rsrc(id:$instr.op2.baser), isrc(val:$instr.op1.imm),
     uflags:$instr.wflags)

# IMUL (reg, mem, imm)
case instr_t(type:ar-wf, nb_op:3, op1(type:imm), op2(type:mem),
     op3(type:reg)) -> TernopRMI(addr:$curaddr,
     dst(id:$instr.op3.baser), msrc(name:$instr.op2.name
     base(id:$instr.op2.baser), index(id:$instr.op2.indexr),
     scale(val:$instr.op2.scale), off(val:$instr.op2.imm)),
     isrc(val:$instr.op1.imm), uflags:$instr.wflags)

# NEG (reg)
case instr_t(instr:NEG, op1(type:reg)) -> TernopR3(addr:$curaddr,
     dst(id:$instr.op1.baser), rsrc1(id:$instr.op1.baser),
     rsrc2(id:$instr.op1.baser), uflags:$instr.wflags)

# NEG (mem)
case instr_t(instr:NEG, op1(type:mem)) -> TernopM3(addr:$curaddr,
     dst((name:$instr.op1.name base(id:$instr.op1.baser),
     index(id:$instr.op1.indexr), scale(val:$instr.op1.scale),
     off(val:$instr.op1.imm))), msrc1((name:$instr.op1.name
     base(id:$instr.op1.baser), index(id:$instr.op1.indexr),
     scale(val:$instr.op1.scale), off(val:$instr.op1.imm))),
     msrc2((name:$instr.op1.name base(id:$instr.op1.baser),
     index(id:$instr.op1.indexr), scale(val:$instr.op1.scale),
     off(val:$instr.op1.imm))), uflags:$instr.wflags)

# MUL, IMUL (reg)
case instr_t(type:ar-wf, nb_op:1, op1(type:reg)) ->
     TernopR3(addr:$curaddr, dst(id:EAX), rsrc1(id:$instr.op1.baser),
     rsrc2(id:EAX), uflags:$instr.wflags)

# MUL, IMUL (mem)
case instr_t(type:ar-wf, nb_op:1, op1(type:mem)) ->
     TernopR3(addr:$curaddr, dst(id:EAX), rsrc1((name:$instr.op1.name
     base(id:$instr.op1.baser), index(id:$instr.op1.indexr),
     scale(val:$instr.op1.scale), off(val:$instr.op1.imm))),
     rsrc2(id:EAX), uflags:$instr.wflags)

# CWD, CBW
case instr_t(type:ar, nb_op:0) -> TernopR3(addr:$curaddr, dst(id:EAX),
     rsrc1(id:EAX), rsrc2(id:EAX), uflags:0)

# LEA (reg, mem)
case instr_t(type:ar, nb_op:2) -> TernopRM(addr:$curaddr,
     dst(id:$instr.op1.baser), rsrc(id:$instr.op1.baser),
     msrc((name:$instr.op1.name base(id:$instr.op1.baser),
     index(id:$instr.op1.indexr), scale(val:$instr.op1.scale),
     off(val:$instr.op1.imm))), uflags:0)

# NOT, DIV, IDIV (reg)
case instr_t(type:ar, nb_op:1, op1(type:reg)) ->
     TernopR3(addr:$curaddr, dst(id:EAX), rsrc1(id:EAX),
     rsrc2(id:$instr.op1.baser), uflags:0)

# NOT, DIV, IDIV (mem)
case instr_t(type:ar, nb_op:1, op1(type:mem)) ->
     TernopRM(addr:$curaddr, dst(id:EAX), rsrc(id:EAX),
     msrc((name:$instr.op1.name base(id:$instr.op1.baser),
     index(id:$instr.op1.indexr), scale(val:$instr.op1.scale),
     off(val:$instr.op1.imm))), uflags:0)

# CALL (reg)
```

```
case instr_t(type:c, op1(type:reg)) -> IndCallR(addr:$curaddr,
     dst(id:$instr.op1.baser), uflags:0)

# CALL (mem)
case instr_t(type:c, op1(type:mem)) -> IndCallM(addr:$curaddr,
     dst((name:$instr.op1.name base(id:$instr.op1.baser),
     index(id:$instr.op1.indexr), scale(val:$instr.op1.scale),
     off(val:$instr.op1.imm))), uflags:0)

# CALL (imm)
case instr_t(type:c, op1(type:imm)) -> Call(addr:$curaddr,
     dst(val:$instr.op1.imm), uflags:0)

# CMP, TEST (reg, imm)
case instr_t(type:cmp-wr, op1(type:imm), op2(type:reg)) ->
     CmpRI(addr:$curaddr, snd(id:$instr.op2.baser),
     fst(val:$instr.op1.imm), uflags:$instr.wflags)

# CMP, TEST, CMPSB, CMPSD, SCASB, SCASD (reg, reg)
case instr_t(type:cmp-wr, op1(type:reg), op2(type:reg)) ->
     CmpRR(addr:$curaddr, snd(id:$instr.op2.baser),
     fst(id:$instr.op1.baser), uflags:$instr.wflags)

# CMP (reg, mem)
case instr_t(type:cmp-wr, op1(type:mem), op2(type:reg)) ->
     CmpRM(addr:$curaddr, snd(id:$instr.op2.baser),
     fst((name:$instr.op1.name base(id:$instr.op1.baser),
     index(id:$instr.op1.indexr), scale(val:$instr.op1.scale),
     off(val:$instr.op1.imm))), uflags:$instr.wflags)

# CMP, TEST (mem, reg)
case instr_t(type:cmp-wr, op1(type:reg), op2(type:mem)) ->
     CmpMR(addr:$curaddr, snd((name:$instr.op2.name
     base(id:$instr.op2.baser), index(id:$instr.op2.indexr),
     scale(val:$instr.op2.scale), off(val:$instr.op2.imm))),
     fst(id:$instr.op1.baser), uflags:$instr.wflags)

# TEST (mem, imm)
case instr_t(type:cmp-wr, op1(type:imm), op2(type:mem)) ->
     CmpMI(addr:$curaddr, snd((name:$instr.op2.name
     base(id:$instr.op2.baser), index(id:$instr.op2.indexr),
     scale(val:$instr.op2.scale), off(val:$instr.op2.imm))),
     fst(val:$instr.op1.imm), uflags:$instr.wflags)

# HLT
case instr_t(type:s) -> Stop(addr:$curaddr, uflags:0)

# IN (reg, reg)
case instr_t(type:io-rm, op1(type:reg)) -> IoRR(addr:$curaddr,
     dst(id:$instr.op2.baser), src(id:$instr.op1.baser), uflags:0)

# IN (reg, imm)
case instr_t(type:io-rm, op1(type:imm)) -> IoRI(addr:$curaddr,
     dst(id:$instr.op2.baser), src(val:$instr.op1.imm), uflags:0)

# INTO
```

```
case instr_t(type:i-rf-bt) -> Interrupt(addr:$curaddr, dst(val:4),
     uflags:0)

# IRET
case instr_t(type:i-r) -> IReturn(addr:$curaddr, uflags:0)

# INT3
case inst_t(type:i, nb_op:0) -> Interrupt(addr:$curaddr, dst(val:3),
     uflags:0)

# INT (imm)
case instr_t(type:i) -> Interrupt(addr:$curaddr,
     dst(val:$instr.op1.imm), uflags:0)

# Jxx, LOOP, LOOPE, LOOPNE (imm)
# Jxx = JA, JAE, JB, JBE, JE, JECXZ, JG, JGE, JL, JLE, JNE, JNO, JNP,
     JNS, JO, JP, JS
case instr_t(type:cb) -> Branch(addr:$curaddr, dst(val:$instr.op1.imm),
     uflags:0)

# JMP (imm)
case instr_t(type:b, op1(type:imm)) -> Branch(addr:$curaddr,
     dst(val:$instr.op1.imm), uflags:0)

# JMP (reg)
case instr_t(type:b, op1(type:reg)) -> IndBranchR(addr:$curaddr,
     dst(id:$instr.op1.baser), uflags:0)

# JMP (mem)
case instr_t(type:b, op1(type:mem)) -> IndBranchM(addr:$curaddr,
     dst((name:$instr.op1.name base(id:$instr.op1.baser),
     index(id:$instr.op1.indexr), scale(val:$instr.op1.scale),
     off(val:$instr.op1.imm))), uflags:0)

# MOVSB, MOVSD (mem, mem)
case instr_t(type:a-rm-wm) -> AssignMM(addr:$curaddr,
     dst((name:$instr.op2.name base(id:$instr.op2.baser),
     index(id:$instr.op2.indexr), scale(val:$instr.op2.scale),
     off(val:$instr.op2.imm))), src((name:$instr.op1.name
     base(id:$instr.op1.baser), index(id:$instr.op1.indexr),
     scale(val:$instr.op1.scale), off(val:$instr.op1.imm))), uflags:0)

# NOP, WAIT/FWAIT
case instr_t(type:n) -> Nop(addr:$curaddr, uflags:0)

# OUT (reg, reg)
case instr_t(type:io-wm, op2(type:reg)) -> IoRR(addr:$curaddr,
     dst(id:$instr.op2.baser), src(id:$instr.op1.baser), uflags:0)

# OUT (imm, reg)
case instr_t(type:io-wm, op2(type:imm)) -> IoIR(addr:$curaddr,
     dst(val:$instr.op2.imm), src(id:$instr.op1.baser), uflags:0)

# POP (reg)
case instr_t(type:sp-rm, op1(type:reg)) -> AssignRM(addr:$curaddr,
     dst(id:$instr.op1.baser), src(base(id:ESP)), uflags:0)
```

```
# POP (mem)
case instr_t(type:sp-rm, op1(type:mem)) -> AssignMM(addr:$curaddr,
     dst((name:$instr.op1.name base(id:$instr.op1.baser),
     index(id:$instr.op1.indexr), scale(val:$instr.op1.scale),
     off(val:$instr.op1.imm))), src(base(id:ESP)), uflags:0)

# PUSH (reg)
case instr_t(type:sp-wm, op1(type:reg)) -> AssignMR(addr:$curaddr,
     dst(base(id:ESP)), src(id:$instr.op1.baser), uflags:0)

# PUSH (imm)
case instr_t(type:sp-wm, op1(type:imm)) -> AssignMI(addr:$curaddr,
     dst(base(id:ESP)), src(val:$instr.op1.imm), uflags:0)

# POPF
case instr_t(type:sp-rm-wf) -> AssignRM(addr:$curaddr, dst(id:EFLAGS),
     src(base(id:ESP)), uflags:$instr.wflags)

# PUSHF
case instr_t(type:sp-wm-rf) -> AssignMR(addr:$curaddr,
     dst(base(id:ESP)), src(id:EFLAGS), uflags:$instr.rflags)

# RET, RETF
case instr_t(type:r) -> Return(addr:$curaddr, dst(val:0), uflags:0)

# XCHG (reg, reg)
case instr_t(type:wm-rm, op2(type:reg)) -> XchgRR(addr:$curaddr,
     fst(id:$instr.op2.baser), snd(id:$instr.op1.baser), uflags:0)

# XCHG (mem, reg)
case instr_t(type:wm-rm, op2(type:mem)) -> XchgMR(addr:$curaddr,
     fst((name:$instr.op2.name base(id:$instr.op2.baser),
     index(id:$instr.op2.indexr), scale(val:$instr.op2.scale),
     off(val:$instr.op2.imm))), snd(id:$instr.op1.baser), uflags:0)

# XLATB
case instr_t(type:a-rm) -> AssignRM(addr:$curaddr, dst(id:EAX),
     src(base(id:EBX), index(id:EAX)), uflags:0)

# STOSB, STOSD (mem, reg)
case instr_t(type:wm) -> AssignMR(addr:$curaddr,
     dst((name:$instr.op2.name base(id:$instr.op2.baser),
     index(id:$instr.op2.indexr), scale(val:$instr.op2.scale),
     off(val:$instr.op2.imm))), src(id:$instr.op1.baser), uflags:0)

# MOV (reg, imm)
case instr_t(type:a, op1(type:imm), op2(type:reg)) ->
     AssignRI(addr:$curaddr, dst(id:$instr.op2.baser),
     src(val:$instr.op1.imm), uflags:0)

# MOV (reg, mem)
case instr_t(type:a, op1(type:mem), op2(type:reg)) ->
     AssignRM(addr:$curaddr, dst(id:$instr.op2.baser),
     src((name:$instr.op1.name base(id:$instr.op1.baser),
     index(id:$instr.op1.indexr), scale(val:$instr.op1.scale),
     off(val:$instr.op1.imm))), uflags:0)
```

```
# MOV (reg, reg)
case instr_t(type:a, op1(type:reg), op2(type:reg)) ->
     AssignRR(addr:$curaddr, dst(id:$instr.op2.baser),
     src(id:$instr.op1.baser), uflags:0)

# MOV (mem, reg)
case instr_t(type:a, op1(type:reg), op2(type:mem)) ->
     AssignMR(addr:$curaddr, dst((name:$instr.op2.name
     base(id:$instr.op2.baser), index(id:$instr.op2.indexr),
     scale(val:$instr.op2.scale), off(val:$instr.op2.imm))),
     src(id:$instr.op1.baser), uflags:0)

# MOV (mem, imm)
case instr_t(type:a, op1(type:imm), op2(type:mem)) ->
     AssignMI(addr:$curaddr, dst((name:$instr.op2.name
     base(id:$instr.op2.baser), index(id:$instr.op2.indexr),
     scale(val:$instr.op2.scale), off(val:$instr.op2.imm))),
     src(val:$instr.op1.imm), uflags:0)

# LDS, LES (reg, imm)
case instr_t(type:rm, op1(type:imm)) -> AssignRI(addr:$curaddr,
     dst(id:$instr.op2.baser), src(val:$instr.op1.imm), uflags:0)

# LODSB, LODSD (reg, mem)
case instr_t(type:rm) -> AssignRM(addr:$curaddr,
     dst(id:$instr.op2.baser), src((name:$instr.op1.name
     base(id:$instr.op1.baser), index(id:$instr.op1.indexr),
     scale(val:$instr.op1.scale), off(val:$instr.op1.imm))), uflags:0)

# LAHF
case instr_t(type:rf) -> FlagR(addr:$curaddr, uflags:$instr.rflags)

# CLC, CLD, CLI, CMC, SAHF, STC, STD, STI
case instr_t(type:wf) -> FlagW(addr:$curaddr, uflags:$instr.rflags)

# Prolog - SUB (ESP, imm)
case instr_t(instr:SUB, op2(baser:ESP), op1(type:imm)) ->
     Prolog(addr:$curaddr, framesz(val:$instr.op1.imm), uflags:0)

# Epilog - MOV (ESP, EBP)
case instr_t(instr:MOV, op2(baser:ESP), op1(baser:EBP)) ->
     Epilog(addr:$curaddr, uflags:0)

# Defaultcase
default print Unsupported instruction at address $curaddr

endtrans

add $curaddr $instr.len

endfor
```