Automatic Synthesis of Instruction Set Semantics and its Applications

A Dissertation presented

by

Niranjan Sudhir Hasabnis

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

August 2015

Stony Brook University

The Graduate School

Niranjan Sudhir Hasabnis

We, the dissertation committee for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation

Dr. R. Sekar - Dissertation Advisor
Professor, Computer Science

Dr. Mike Ferdman - Chairperson of Defense
Assistant Professor, Computer Science

Dr. Michalis Polychronakis
Assistant Professor, Computer Science

Dr. Suresh Srinivas
Principal Engineer, Intel Corporation

This dissertation is accepted by the Graduate School

Charles Taber
Dean of the Graduate School

Abstract of the Dissertation

Automatic Synthesis of Instruction Set Semantics and its Applications

by

Niranjan Sudhir Hasabnis

Doctor of Philosophy

in

Computer Science

Stony Brook University

2015

Binary analysis, translation and instrumentation tools play an important role in software security. To support binaries for different processors, it is necessary to incorporate the semantics of every processor's instruction set into the tool. Unfortunately, the complexity of modern instruction sets makes the common approach of manual semantics modeling cumbersome and error-prone. Furthermore, it limits the number of processors as well as the fraction of the instruction set that is supported.

In this dissertation, we propose novel architecture-neutral techniques for automatically synthesizing the semantics of instruction sets. Our approach relies on the observation that modern compilers such as GCC and LLVM already contain detailed knowledge about the semantics of many instruction sets. We therefore develop two techniques for extracting this knowledge. Our first technique relies on a learning process: observing examples of translation between a compiler's architecture-neutral internal representation and machine instructions, and inferring the mapping from these examples. We then develop a second (and complementary) method that develops symbolic execution techniques to extract this mapping from the code generator source. Unlike previous symbolic execution systems that specialize in generating a single solution to a set of constraints, our problem requires a compact representation of all possible solutions. We describe the development of such a system, based on source-to-source transformation of C-code and a runtime system that is implemented in C and Prolog with a finite-domain constraint solver (CLP-FD).

To demonstrate the applicability of synthesized instruction-set semantics, we develop two applications. In the first application, we use synthesized semantics to test correctness of code generators. Specifically, we develop a new testing approach that generates and executes test cases based on the derived semantic model for each instruction. We uncovered nontrivial bugs in the GCC code generator using this technique. As a second application, we have used these models to lift binaries for x86, ARM and AVR (used in Arduino and other microcontroller) architectures to intermediate code, which can then be analyzed or instrumented in an architecture-independent manner.

Dedicated to,

Aai, Baba, and Nilesh (my parents and my brother)
Without their support I would not have started this journey.

&

Anuja (my wife and love of my life)
Without her support I would not have been able to complete it.

Table of Contents

Contents

List of Figures

Acknowledgements

I would like to take this opportunity to express my deepest appreciation and sincere gratitude to my advisor, Dr. R. Sekar, for his guidance, motivation, and support during my dissertation research. I would also like to sincerely thank him for providing financial support during my research and also for providing help in a number of personal issues. His suggestions, comments and insightful discussions have not only taught me how to be a good researcher, but have also enriched me for the rest of my life.

I would also like to thank my committee members, Dr. Mike Ferdman, Dr. Michalis Polychronakis, and Dr. Suresh Srinivas for dedicating their precious time to be a part of my committee. I am gracious for their constant support and constructive advice during my research.

I would also like to thank all the faculty and staff in the Department of Computer Science here at Stony Brook for their help during my research. I would especially like to thank Dr. Scott Stoller and Dr. Robert Johnson for giving me an opportunity to work with them during my Masters. Discussions with them were always very encouraging and enjoyable, and they helped me learn a lot. I would also like to thank Brian Tria for our conversations. Talking to him was always fun and much needed after intense work.

I would also like to thank all the people in Secure Systems Lab with whom I got a chance to work. Special thank goes to Alireza, Tung, Riccardo, Peter, Mingwei, Rui, Laszlo, Ashish, and Jeet. The time that we spent together solving problems, discussing research papers, or having fun will always be remembered. They all made my stay in Stony Brook very pleasurable. I would sincerely like to thank Tung and my close friend, Saurabh, who helped me a lot in a number of personal problems.

Last but not the least, I am so grateful to my parents, my brother Nilesh, and my wife Anuja for their love and support throughout the entire life. I am especially grateful to my father without whose encouragement I would not have started my PhD studies. I am also eternally grateful to my wife who sacrificed many things for my studies. Without her love and support, I would not have been able to complete my dissertation. Lastly, I would also like to thank my in-laws for encouraging and supporting me during my studies. Much of this work reflects the values that they all have installed in me. Without their love and support, I would not have been able to complete my dissertation. I am forever indebted to all of them. I dedicate my dissertation to all of them.

1 Introduction

Binary analysis, translation, and instrumentation techniques have been phenomenal in solving a number of important problems in software engineering. Binary translation is the fundamental technique in dynamic malware analysis tools that are used by the anti-malware industry daily. Dynamic malware analysis requires malware execution in a controlled environment, so that effects of the execution are confined, and they do not spread into the underlying system. System emulators (such as QEMU [15]) and virtual machines (such as VMware [93]), which provide such controlled environment use binary translation as the underlying technology. In the past, binary translators have also been used to solve the problem of cross-platform portability and rapid deployment of software [22]. Binary instrumentation has been a popular technique with a number of important applications such as program debugging and program analysis. Dynamic binary instrumentation systems such as Pin [55], Valgrind [67], and DynamoRio [17] are popular among programmers for debugging and analyzing their programs.

A number of popular software systems that exist in the world today need to deal with the architecture-level (also called processor-level or low-level) details routinely. Technically, we can say that these systems need to model the *semantics* of target architectures that they want to support. Compilers are a good and well-known example of such systems. To be precise, many modern compilers support a number of target architectures, and their backends contain some architecture-specific components such as code generators. Binary analysis, translation, and instrumentation systems are another example of such software systems.

Modeling low-level architecture details in software is a very tedious task. Moreover, this task is made complicated by the fact that modern instruction sets are complex. For instance, Intel's instruction set manual (v52) [47] consists of around 2000 pages split across 3 volumes. To add to the complexity, new specialized instruction set extensions are proposed each year. To complicate the problem further, the systems described earlier usually rely on manual modeling of architecture semantics. Specifically, these systems often target architecture-neutrality — which is one of the desired properties of these systems — by translating low-level machine instructions from input binaries into higher-level intermediate representation (IR) instructions. Working at the IR level makes analysis and instrumentation architecture-neutral and hence applicable to all supported architectures. Unfortunately, most of these systems build the assembly-to-IR translators manually. Manual development of assembly-to-IR translators translate to a number of limitations for these systems. First, considerable effort is needed to port these systems to new architectures, and as a result, most of these systems support a limited number of architectures. For instance, QEMU, a popular system emulator, supports 17 target architectures. Second, most of these systems do not support many instructions from the recently added instruction set extensions even for popular architectures such as x86. For instance, Valgrind[1], one of the most popular dynamic binary instrumentation tools, still lacks support for several classes of x86 instruction sets such as AVX, FMA4, and SSE4.1, even after being in development for 10 years.

1.1 Overview of Approaches and Dissertation Organization

In this dissertation, we address the problem of modeling the semantics of modern and complex architectures by developing two novel *architecture-neutral* approaches for *automatic* semantics extraction. Our approaches start with the goal of eliminating manual modeling effort completely. When this objective cannot be achieved, then they answer the question of how much manual effort can be reduced.

Automatic extraction of instruction set semantics is a very challenging problem given that the common approach for extraction is to rely on actual hardware. Tendency to rely on hardware stems from (a) the desire make the models correct and accurate, and (b) the fact that hardware is the best source of accurate and correct semantic information (Though hardware manuals may also serve as a good source of correct and accurate information, generally, it is easier to automate the extraction process with hardware than the manuals.) Unfortunately, synthesizing instruction semantics using hardware demands execution of every machine instruction under every possible machine state, which requires one to explore the huge input space of every target instruction. This challenge has prevented the development of systems for automatic semantics extraction for complex architectures such as the x86. Nonetheless, research works have attempted to address this problem by using manual effort to make traversal of the input space practically feasible. We will talk about these research efforts in Section 2.

Given that automatic extraction by using hardware is infeasible without manual effort, in this dissertation, we do not use hardware. Instead, we hypothesize that we can use modern compilers for such extraction. Modern compilers such as GCC [3] and LLVM [8]

[1]v3.10.0, 2014

already contain detailed knowledge about the semantics of many architectures. Specifically, architecture-set specifications used by the code generators of these compilers contain a mapping between target assembly instructions and their semantics in IR. These mapping tables are used by the code generators to translate IR instructions into target assembly instructions. We hypothesize that the knowledge encoded in compilers can be extracted automatically and develop a white-box and a black-box approach for the extraction. We chose compilers for such extraction because they support a number of architectures, and their extensive testing minimizes the chances of semantic modeling errors in them. For instance, GCC and LLVM both support more than 40 target architectures — considerably more than those supported by QEMU. Moreover, being actively developed, these compilers are tested by large test suites.

One approach to extract instruction-set semantics from code generators is to use their architecture specifications directly. As we will see in Section 2, the existing approaches for writing these specifications are such that their direct use is complicated. The problem is that the compilers encode a lot of architecture-specific knowledge in their source code in addition to their specifications. Since direct use of specifications is complicated, a feasible approach would be to use the code generators in their common usage scenario of compiling source programs. Our black-box approach, called `LISC` (Section 3), relies on the observation that, while compiling source programs, modern compilers can output IR-to-assembly mapping rules used by their code generators. IR instructions in these rules specify the semantics of the corresponding assembly instructions. `LISC` *learns* the instruction-set semantic model encoded in the code generators by referring to their compilation logs.

`LISC` faces two interesting challenges in using compilation logs for model extraction. First, modern compilers may not support all of the target assembly instructions. For example privileged instructions are generally not supported by the compilers. Moreover, privileged instructions may not be the only instructions that are not supported. Because of this, models extracted from compilers will be incomplete. Second, even if compilers supported all of the target assembly instructions, the extraction process would need to ensure that all of the code generator mapping pairs are covered. It is likely that, for a typical compilation, a code generator may not refer to all of the target assembly instructions. For instance, GCC does not use instructions from most of the advanced x86 instruction set extensions unless a programmer directs otherwise. Both of these coverage challenges need to be addressed, because the extracted semantic models *should* contain all of the target assembly instructions. Unsupported instructions may be covered by using other compilers for extraction, but achieving complete coverage of complex software such as code generators has been one of the challenging problems in software testing. We will address these challenges in Section 3.

As an alternative to LISC, we develop a white-box approach, called EISSEC, to extract semantic model through symbolic execution of the code generators. Symbolic execution is a technique in the software testing domain to improve program coverage. Unfortunately, it suffers from the "explosion" problem, where the exponentially increasing number of program paths make symbolic execution of complex programs infeasible. Recent research [19, 24, 21] addresses this challenge, but, unfortunately, we have not come across cases of symbolic execution being applied to complex source programs such as code generators. In EISSEC, we try to assess how we can scale symbolic execution to code generators. We believe that the solutions which we propose in EISSEC will make important contributions towards improving applicability of symbolic execution to complex programs. We will describe EISSEC in Section 4.

Correctness and completeness are the two desired properties of the models extracted using LISC and EISSEC. This is because the correct results from the applications built using these models depend upon the correctness of these models. Completeness, on the other hand, is a nice-to-have but not necessary property because its absence results in missing assembly instructions in the models. We develop two approaches to ensure correctness and completeness of the models. First, we empirically evaluate these properties in Section 3. Second, we develop an approach, called ArCheck, to check the correctness of extracted models. ArCheck formalizes the correctness checking problem as the problem of checking the semantic equivalence of every assembly and its corresponding IR instruction from the extracted models. Since the semantics of an assembly instruction can only be obtained by concretely evaluating that instruction (unless we want to manually model the semantics, which we want to avoid), approaches such as symbolic equivalence checking (a fundamental approach used in software verification) cannot be used for our purpose. Our semantic equivalence test thus involves comparison of IR and assembly semantics obtained by their concrete evaluation. But for concrete evaluation, instead of employing a random testing approach, ArCheck develops a systematic coverage testing strategy to produce test cases and to maximize confidence in the correctness of results. Solutions proposed by ArCheck make two important contributions to the compiler testing problem. First, ArCheck solves the problem of checking the correctness of code generators (for which, to the best of our knowledge, no verification work exists). Second, it provides a practical approach that can be applied to the code generator of any compiler. We will describe `ArCheck` in Section 5.

Once the extracted semantic models are checked for correctness, we demonstrate their applicability to developing assembly-to-IR translators. By building these translators automatically, we demonstrate that these systems can quickly support new architectures as well as new instructions from already supported architectures.

This dissertation is organized as follows. Section 2 discusses various approaches for automatic synthesis of instruction-set semantics and compares them with our proposed approach. Section 2.3.1 provides background on modern compilers relevant to this dissertation. Section 3 and Section 4 discuss our approaches for automatic instruction-set semantics extraction. Section 5 discusses our approach to check the correctness of extracted semantic models and code generators. Section 6 discusses an application of extracted models in building assembly-to-IR translators. Lastly, Section 7 concludes this dissertation and discusses future extensions of this work.

1.2 High-Level Overview of Contributions

One of the important contributions of this dissertation is to approach the problem of *automatic* instruction-set semantic extraction by developing two novel *architecture-neutral* approaches. The solutions proposed in this dissertation have a number of important applications such as building binary analysis, instrumentation and translation systems, code generators, and CPU emulators. Furthermore, since these systems are expected to model the low-level semantics faithfully, the extracted semantic models can be used to test these systems and check if this expectation is valid.

In addition to developing approaches for automatic extraction of semantics, in this dissertation, we demonstrate two applications of the extracted models. In the first application, we use the models to *check the correctness of compiler code generators*. Although compilers are tested extensively, they are known to contain bugs. We observed that code generators of modern compilers contained semantic modeling bugs. Given the critical role played by the compilers, it is necessary that such modeling bugs are caught in the development. Our solution thus contributes to the important problem of testing compilers.

In the second application, we use the extracted models to *build assembly-to-IR translators used by the architecture-neutral binary analysis, translation and instrumentation systems automatically*. A common approach to build assembly-to-IR translators is to build architecture-neutral translators and drive them using manually-written target descriptions.

For such systems, automatically extracted semantic models can be used to replace manually-written architecture specifications. Our solution helps in improving the effectiveness of these systems by allowing them to target more architectures and also supporting existing architectures completely.

Although we described the problem of manually-developing assembly-to-IR translators by using examples of existing systems such as Valgrind and QEMU, this dissertation does not consider the problem of building a complete end-to-end binary translation, instrumentation and analysis system. It instead demonstrates the applicability of the extracted models in building the assembly-to-IR translators (which are one of the building blocks of these systems). Moreover, it is not within the scope of this dissertation to address the limitations of a specific system (e.g., Valgrind). By translating compiler's IR to the specific tool's IR, one may use the assembly-to-IR translators built using our models to support a specific system.

2 Overview of approaches for automatic synthesis of instruction set semantics

In this section, we will discuss various approaches to extract instruction set semantics models automatically. We will then discuss limitations of these approaches before moving on to our approach. But before we discuss these approaches, we need to specify precisely what is the level of detail that we want in the extracted semantics.

2.1 Level of Abstractions Required In the Extracted Semantics

Semantics of an instruction can be specified and modeled at different levels of abstraction. Too much detailed semantics may not be desirable to applications which actually expect slightly abstracted semantics. Thus, it is important to establish the level of details that we want the extracted semantics to confirm. The desired level of details of the extracted semantics is actually dictated by the applications which will consume the semantic models at that level. Thus to establish the level of abstractions, we need to understand the kind of applications that will be consumers of our extracted models. In this dissertation, we target applications which work at the level of binaries but which want to target multiple architectures without much porting efforts. An example of such application could be a binary analysis framework. Typically, such a framework operates on binaries, but the kind of analyses performed are mostly architecture-neutral. So such frameworks operate on architecture-neutral representation of the input binary. Such a representation essentially makes the framework applicable to binaries from any architecture. Our extracted model will be used by such a framework to translate input binary into an architecture-neutral representation. Let us now see what level of detail these applications expect from the architecture-neutral representation.

Semantics of an assembly instruction can be represented at various levels of detail. For instance, semantics of `addl %eax,%ebx` could be described in the following ways:

D1 : add the value of `eax` to `ebx` and store the result in `ebx`, or

D2 : one could simply enumerate possible values of `eax` and `ebx`, and could specify the outcome of the execution of the instruction (treating instruction semantics as a function between possible input values and output values), or

D3 : add the value in first input operand with the value in second input operand, and store the result in the output operand.

In this dissertation, we are interested in representing the semantics at the level of detail captured in the first option. In fact, that is the level of details that the applications like binary analysis are interested in. Semantics specified in the second option are too detailed, and a binary analysis system may not really process such level of details[2]. The third option, on the other hand, captures the semantics at a very abstract level, and thus is not desirable for binary analysis systems either. Various approaches for semantics extraction that we will describe now extract the semantics at different levels. In addition to this mismatch, they have some limitations which make them unsuitable for our purposes.

Notations and terms. Before we describe our approaches, we define the notations and terms which will be used in the rest of the dissertation.

We use notation A to represent an assembly instruction. An assembly instruction is *abstract* when the operands of the assembly instruction are abstract. For instance, "`add %1, %2`" is an abstract assembly instruction for the x86 `add` instruction. A *concrete* assembly instruction, on the other hand, has concrete operands. For instance, "`add %eax, %ebx`" is a concrete x86 `add` instruction. We use A_c to represent a concrete assembly instruction, and A_a to represent an abstract assembly instruction. Notation A represents both abstract and concrete assembly instruction.

We use notation σ to represent the semantics of an instruction. The semantics of concrete assembly instructions can be defined concretely in terms of the processor state changes caused by their execution. The semantics of abstract assembly instructions, on the other hand, cannot be defined concretely. For instance, the semantics of x86 `add` instruction as it is given in the Intel manual is shown in Figure 1.

We use notation R to represent a mapping between an assembly instruction A and its semantics σ. We represent R as $R = \langle A, \sigma \rangle$.

We use notation $\mathscr{M}$ to represent semantic model. A semantic model is a collection of mapping rules between all the assembly instructions for some target architecture and their semantics. Thus, a semantic model can be formally defined as $\mathscr{M} = \{\langle A_i, \sigma_i \rangle\} \forall i \in \mathscr{T}$, where $\mathscr{T}$ is a target architecture.

[2]This is the case because systems which perform binary analysis operate on IR instructions which contain operators like `plus` to model semantics of addition. So they do not need to know what is the meaning of addition.

Description:
Adds the destination operand (first operand) and the source operand (second operand) and then stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADD instruction performs integer addition. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate a carry (overflow) in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation:
DEST $\leftarrow$ DEST + SRC;

Flags Affected:
The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

Figure 1: **Semantics of x86 `add` instruction taken from Intel manual**

2.2 Possible Approaches

The high-level idea behind these approaches is to rely on authentic sources of instruction semantics. There exist two sources of such information: manuals supplied by the processor vendors (e.g., Intel's instruction set manual [5]), and the CPU itself. So these approaches rely on either of them, with tendency to prefer CPU over the manuals. Recently, there have been approaches which rely on "software" form of a CPU. By "software" form of a CPU, we mean some software which mimics the functionality of an actual CPU (e.g., CPU emulators). "Software" CPU is used because actual CPU imposes some restriction on its usage. We will see what these restrictions are, and also other approaches now.

2.2.1 Extraction from instruction-set manuals

Vendor-supplied instruction-set manuals are one of the authentic sources of information about instruction semantics. In fact, supplying such information is one of the main purposes of such manuals. So it is perfectly logical to think of such manuals as a first stop for the instruction semantics. We saw an example of how these manuals contain instruction semantics by describing the entry found for `addl` x86 instruction in Intel manual. Most manuals contain pseudo code for instruction semantics in addition to English description. Unfortunately, we believe that extracting semantics from such pseudo code is not a trivial task. More important issue is that such an approach will not be architecture-neutral as we need to tailor-made such approach for each and every architecture. Given these limitations, we decided not to pursue such approach at all.

2.2.2 Extraction from the hardware

Given that semantics extraction approach based on CPU can be systematically applied to any other CPU, this is generally a preferred approach over use of instruction-set manuals. Such approaches rely on the observation that the hardware CPU can be treated as a hardware oracle to obtain instruction semantics. Intuitively, to obtain instruction semantics from the CPU, one can simply execute that instruction on the CPU, and compare the pre- and post-state of the CPU to learn the semantics. Although, this approach is simple and intuitive, unfortunately, it is infeasible. Specifically, in order to learn the semantics of some instruction completely, one needs to execute that instruction with all possible input values (if not under various possible operand combinations additionally) and under various possible pre-states of the CPU. Moreover, this option captures the semantics at the level **D2**, which makes it unsuitable for the applications that we are interested in.

To give an idea of the infeasibility, let us consider an example of x86 `add` instruction. x86 `add` instruction takes two operands, one of which being a register or memory location, and the other being either a register, a memory location, or an immediate. The total number of possible operand combinations and their values for `add` are $(8+2^{32})*(6+2^{32}+2^{32})$ — 8 for the number general-purpose registers on x86, 2^{32} for the number of memory locations that can appear in the instruction (considering 4GB virtual address space), and 2^{32} for the total number of 32-bit immediates that can appear. The idea here is that if we execute `add` instruction on x86 processor with these many operand combinations in all possible states of the processor, then we can essentially learn the complete semantics of `add` instruction.

It is easy to see that the number of operand combinations needed for exhaustive enumeration approach are far too many to be feasible in practice.

2.2.3 Extraction by exhaustive testing of CPU

The drawback of exhaustive enumeration is that it needs to cover the huge space of possible inputs of an architecture instruction in order to obtain its semantics. A natural question to ask then is: can we reduce this space somehow? Some research efforts such as [41] try to answer this question by relying on manual efforts. Specifically, they manually define instruction semantics templates in such a way that they are (1) generic enough that few such templates suffice for the whole instruction set, (2) but at the same time specific enough that they reduce the input space considerably. Work described in [41] uses 6 such templates to model semantics of 580 x86 instructions (8-bit, 16-bit and 32-bit). For this work, semantics of an instruction can be thought of as a function which maps the inputs of the instruction to its outputs. Instead of searching the function implemented by an architecture instruction in the huge space of possible functions, they restrict their search to the manually-defined function templates. The contribution of this work is an efficient approach which makes the function synthesis from I/O samples of 32-bit instructions possible. In order to discriminate multiple functions which may satisfy a subset of all possible I/O samples of an instruction, they develop a notion of discriminating inputs (called as smart inputs).

Although their approach is effective in synthesizing functions of 580 x86 instructions, it suffers from following limitations. First of all, the level of details captured in the extracted semantics is still at level **D2** which makes this approach unsuitable for our purpose. While their goal is to encode the extracted semantics in symbolic formulas which can be used by symbolic execution systems, the applications that we are interested in do not look for such level of details. Second, we believe that designing the templates and the smart inputs demands considerable understanding of the target assembly instructions. In their work, they had to study instruction-set manuals and had to manually develop the templates and the inputs. It is unclear how easy/difficult it is to come up with these templates and inputs for complex instruction set extensions such as SSE and AVX for x86. Moreover, it is unclear how much time does it take for them to support a new architecture. As we will see later in this dissertation, as compared to this approach, our approaches require very minimal (if not none) architecture knowledge to extract their semantics model. Third, designing templates to model semantics of 64-bit instructions, floating points and advanced floating point instructions is very complex and challenging (They also admit about this complexity.) The approaches that we propose in this dissertation are not limited by the size of instruction's input space. Thus, our approaches can easily scale to complex instruction sets.

2.2.4 Extraction by symbolic execution of CPU specification

Symbolic execution [49] is a very powerful technique proposed in the literature to exactly solve the kind of problems for which exhaustive enumeration is practically infeasible. It exploits the fact that most of the program inputs follow same program path. Thus, instead of considering each of such inputs separately, one can simply put them in one partition of the input space, and consider only one representative input for the whole partition. By defining few partitions to cover complete input space and by considering only one input per partition, symbolic execution can drastically reduce the number of inputs that we need to consider to obtain complete semantics. Using symbolic execution, it is thus possible to avoid exhaustive enumeration and extract instruction semantics. We will see what are its limitations when applied to the semantics extraction problem.

To extract a complete and accurate instruction set semantics model, one can simply perform symbolic execution of CPU hardware. Theoretically, this idea works fine, but, practically, it is infeasible for the following reasons. First, to perform symbolic execution of a program, one needs access to its source code or binary. In other words, the program must be available in a software form; unfortunately, CPU is hardware. Second, symbolic execution involves setting the program inputs symbolically rather than concretely. Unfortunately, CPU requires concrete inputs instead of symbolic[3]. Symbolic execution of CPU hardware is thus practically infeasible.

Symbolic execution of a publicly-available high-fidelity CPU emulator. A practically feasible idea, which can achieve the same effect as that of symbolic execution of CPU hardware, is to perform symbolic execution of a publicly-available high-fidelity CPU emulator. After all, CPU emulators are expected to faithfully and accurately emulate instruction semantics of the underlying CPU. PokeEmu [57] proposes such idea, and demonstrates how can one use popular emulators such as Bochs [85] for such purpose. Although, such ap-

[3]It is most likely true that CPU manufacturers have software forms of the CPU which are used to validate functionality of the CPU before it is manufactured. As a result, it is conceivable that one could perform symbolic execution of such software forms to perform symbolic execution of the CPU. But, unfortunately, such software forms are always intellectual properties of the respective manufacturers and are not available to general public.

proach can be used to extract level of semantics suitable for our purpose, unfortunately, such an approach suffers from the limitations of the emulators. First and foremost, most of the emulators support a very limited number of architectures. An approach based on such emulators would naturally be applicable to only few architectures. Second, even though high-fidelity emulators exist, they are also shown to contain a number of bugs [57]. So the accuracy of the semantics models extracted from such emulators is an questionable.

2.3 Our Approach

In the last section, we saw how can one use instruction semantics manuals, hardware CPU and software CPU for semantics extraction. We also discussed the limitations of each of them in the semantics extraction process. Given that discussion, we will first specify a number of requirements from the semantics information source that we want to use for extraction. Such requirements can make the comparative study of our approach easy. Then, we will see what our approach is. The requirements from the semantics information source are as below.

R1 : *Accurate and faithful representation of CPU semantics.* This is the most basic and important requirement of the system which we want to use to extract instruction set semantics.

R2 : *Completeness: representation of semantics for all CPU instructions.* It is desirable that the system that we want to use for extraction supports all the instructions from the target architecture. A frequent addition of a number of instruction set extensions to modern architectures underlines the importance of this requirement.

R3 : *Support for diverse architectures.* Since our objective is to extract instruction set semantics automatically, we would like to apply the solution to as many architectures as we can. This requirement also ensures that the applications built using the extracted semantics will be applicable to many architectures. For instance, system emulators will be able to support architectures that are not supported generally. This requirement thus translates into portability of the applications built using extracted instruction set semantics.

Given those requirements above, we found that the high-fidelity CPU emulators satisfy most of the requirements except **R3**. Lack of support for diverse architectures limits the applicability of semantics extraction to very few architectures, thus limiting the overall effectiveness of applications built using extracted semantics models. Modern compilers, such as GCC [3] and LLVM [8], on the other hand, satisfy all the requirements in a better manner than the emulators. Specifically, modern compilers offer following advantages:

- Modern compilers support a number of architectures (requirement **R3**) — many more than those supported by most of the commonly used emulators.
- They are tested extensively through rigorous test suites, thus minimizing the chances of bugs in them (requirement **R1**).
- These compilers are usually very quick to provide support to newly added instruction set extensions (requirement **R2**).
- Infrastructure developed by these compilers such as optimization passes can potentially be reused in other architecture-independent systems. For instance, a quick way to build a retargetable binary translator using extracted instruction set semantics could be to reuse existing compiler infrastructure such as optimizer, register allocator, instruction scheduler, directly. As a result, we will not need to build the components of the binary translator from scratch. Such an approach to build a binary translator has a number of advantages.
  - The process of developing a translator from assembly-to-IR would be greatly simplified because the mapping from assembly-to-IR can be viewed as an inverse of the mapping from IR-to-assembly.
  - The binary translator is inherently retargetable as it inherits retargetability from the compiler. When the compiler is retargeted to a new platform, new backend can be used to facilitate retargeting of the translator.
  - As the binary translator and the compiler share same IR, we can simply reuse compiler optimizations and other passes (such as register allocation, instruction selection). Binary instrumentation frameworks, such as Valgrind, builds such components manually from scratch.
  - Compiler IRs are well designed. Ability of these compilers to compile source programs written in a number of high-level programming languages and generate binaries for a number of platforms proves that their IRs are comprehensive

enough to represent features from all languages and architectures. So, instead of designing a new IR, the binary translator can simply rely on compiler IR for its purpose.

Given the advantages offered by modern compilers, we decided to use GCC compiler for extracting instruction set semantics. Even though we have used GCC for our research purpose, the ideas and techniques discussed later are equally applicable to LLVM. To understand how these compilers offer a better solution, we need to discuss some of their internal details. A reader familiar with such details can skip this section.

2.3.1 Background on code generation in modern compilers

Structure of a Modern Compiler. Compiler researchers have long worked to develop architecture-independent code generators [33, 32]. These code generators start with an intermediate representation (IR), and emit assembly code. Ideally, the code generator is driven by an instruction set specification called a *machine description* (MD). This MD may take the form of a set of rules, each mapping a snippet of IR into an assembly instruction. Such a rule-based approach can generate inefficient code since it fails to take into account the context in which translation takes place, e.g., it may generate many redundant loads and stores. These inefficiencies can be mitigated by performing several optimizing transformations on the IR, and by driving instruction selection using some cost metrics. Such an approach moves the complexity to IR optimization passes that are shared across different architectures, while simplifying architecture-specific MDs.

Code generators in contemporary compilers such as GCC and LLVM follow this general outline. Due to GCC's maturity and support for many instruction sets, we chose GCC in our implementation. But an important feature of our approach is that it does not depend on GCC specifics, and hence can be applied to other retargetable compiler backends.

Figure 2 shows the key steps in the compilation of a high-level language (a C-program, in this case) by GCC into machine code. These steps are illustrated using a simple example: an assignment statement in C that eventually gets translated into two assembly instructions. The front-end of the compiler translates from different source languages to a high-level intermediate form called GIMPLE. Several optimizations are performed on GIMPLE and then it is translated into GCC's IR, which is called RTL (register transfer language). (These details of the front-end are not relevant for the purposes of this report, but are shown here simply to provide more context.) Then the back-end uses MDs to translate RTL snippets

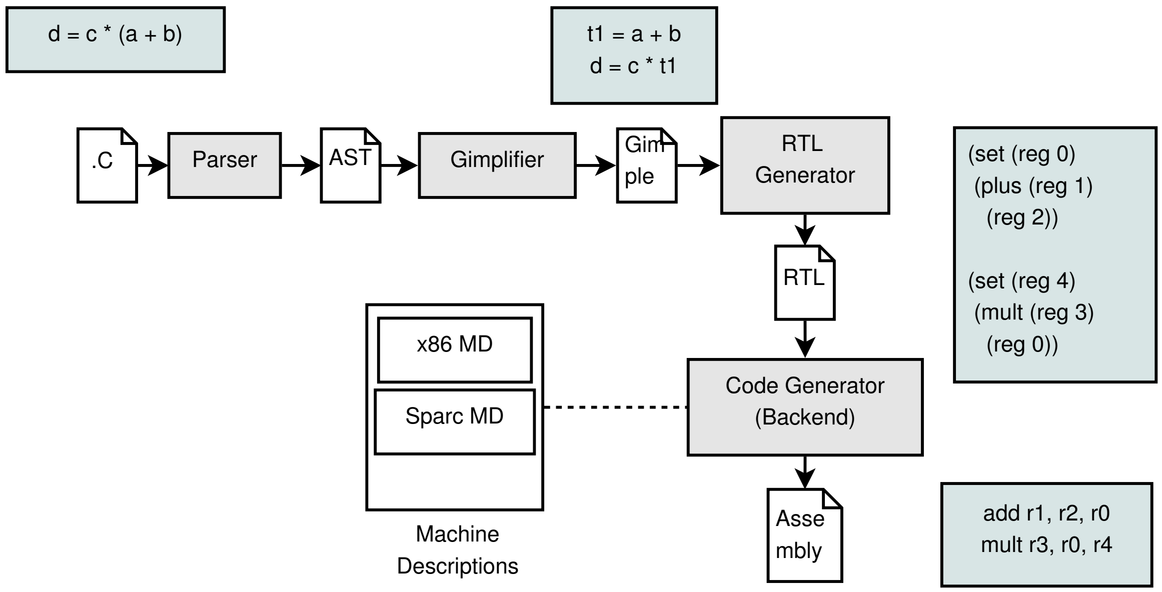


Figure 2: **Key steps in GCC's translation of a source program**

into assembly. As noted earlier, the back-end in GCC incorporates over 40 optimization passes over RTL, and also performs related tasks such as register allocation.

Machine Descriptions. A machine description (MD) specifies all aspects of a target machine architecture, including the number of registers, endianness, parameter passing conventions, and a mapping from RTL snippets to equivalent assembly instruction(s). This mapping is the most important part of an MD, and poses most of the interesting research challenges, so we focus on it below.

As mentioned earlier, code generation is a pattern-driven process: it is specified using rules of the form *RPat* $\longrightarrow$ *Asm* where *RPat* represents an *RTL pattern*, and *Asm* represents equivalent assembly code. Typically, *Asm* is a single instruction, but there are occasional cases where it may contain several instructions. An RTL pattern includes an RTL snippet, together with conditions on when the rule can be applied. The code generator starts with RTL code generated by the compiler front-end. MD rules are matched with snippets of this RTL, and any matching snippet is replaced by the corresponding assembly code. This rewriting process is continued until no more RTL is left. The order in which MD rules are tried may be governed by a cost metric associated with a rule. In addition, the code

```
(set (match_operand: 0 “reg_opnd” “a”)
 (div (match_operand: 1 “reg_opnd” “0”)
  (match_operand: 2 “nonimm_opnd” “qm”)))
   (clobber (reg: FLAGS_REG))
⟶ "div %2"
```

Figure 3: **An MD entry for x86 `div` instruction**

generator may modify RTL under translation in order to find a match, e.g., moving a value from one register (or memory) to another register in order to satisfy conditions specified in RTL patterns.

An example MD rule for the x86 instruction set is shown in Figure 3. To reduce clutter, we have abstracted away some details that are not critical for the purpose of explaining our approach.

This instruction corresponds to x86 divide instruction. The RTL pattern specifies the semantics of this instruction in RTL, namely, that it sets operand 0 to be the result of dividing operand 1 by operand 2. It also states that the flag registers are modified. Since the compiler does not require the flag values after a divide operation, it is simpler to state that flags are modified, instead of detailing the exact change. Nonetheless, when an RTL instruction checks a particular bit of flag values, its preceding instruction sets that particular bit precisely. This pattern is applicable when input RTL matches the `match_operand` conditions, which are broken into a ***predicate*** and a **constraint**. It is not necessary for our discussion to understand the full semantics of predicates and constraints, but we will explain one of them for illustration. The predicate on operand 1 states that the operand should correspond to a register, while the constraint indicates that operand 1 should be the same as operand 0. Moreover, the constraint on operand 0 states that it should be the EAX register. Thus it is quite easy to see that the above example captures the semantics of x86 `div` instruction correctly.

Given the above structure of MD rules, one might simply think that the MD files already represent extracted instruction set semantics. Consequently, one can simply refer to MD files for the extracted semantics and can avoid synthesizing the instruction set semantics altogether. For instance, semantics of `div` instruction can be learned from the RTL contained in the rule. Unfortunately, this simplistic approach runs into many difficulties:

- Some aspects of RTL corresponding to an assembly instruction are defined by the virtue of predicates and constraints, e.g., the condition that the destination operand is the EAX register, or that the two operands of RTL instruction should be identical. While this could potentially be handled by incorporating the semantics of constraints and predicates into a reversing algorithm, such an approach suffers from the drawback that the semantics of many of the predicates and constraints differ across architectures.

- For many rules, the *Asm* part does not directly specify assembly code, but is instead a snippet of C-code that, when executed, will generate a string representation of assembly code. Clearly, it is difficult to statically determine the string(s) that may be generated by a snippet of C-code.

To overcome above problems, we develop approaches that avoid referring the specifics of MD rules. Such approaches eliminate dependencies on architecture specifics in implementation. More importantly, such approaches can be general, and applicable to compilers other than GCC that may use more programmatic MD rules.

Even though direct use of MD files is complex task, it is a very important observation to note that the MD files already contain extracted instruction set semantics. We put this observation along with few other important observations about modern compilers to tackle the problem of automatically extracting instruction set semantics.

2.3.2 Possible challenges for our approaches

Use of compilers for model extraction can lead to some interesting challenges because of the way compilers operate. Specifically, semantics models extracted using code generators may be incomplete and/or incorrect also. Incompleteness of the models comes from two different sources.

First source of incompleteness relates to a possibility that the compilers may not support all of the target architecture instructions. Consequently, such unsupported instructions would also be missing from the extracted models. We expect this to be a case for newly added architecture instructions. Nonetheless, we expect such number of unsupported instructions to be small — actively-developed modern compilers such as GCC and LLVM are generally quick to support newly added instructions (including those from advanced instruction sets).

Second source of incompleteness relates to a common observation that the compilers may not model semantics of every instruction in all the details. Unfortunately, missing details would also be lacking in the models extracted by our approaches. For instance, GCC's machine description may not model *how* CPU flags are modified as a result of some operation, but it may just model the fact that the flags are modified. For instance, GCC's MD entry for x86's `div` instruction discussed in the Section 2.3.1 contains (`clobber` : (`reg` : `FLAGS_REG`)) instead of detailing out *which* bits of `EFLAGS` are modified and *how* they are modified. Such loose semantics modeling is sound for compilers because most of the compiler static analyses may only be interested to know if a particular instruction modifies flags. That is why compilers may capture over-approximation of the instruction semantics. Compilers may not always capture over-approximate semantics all the times — whenever required they may as well capture precise semantics. For instance, in x86, when the `div` instruction is followed by a conditional jump instruction, IR for the `div` instruction will precisely update the bits of `EFLAGS` that are needed by the jump instruction.

Incompleteness of the extracted models can be addressed in multiple ways. First, we can choose a compiler that uses most (if not all) of the target machine instructions. Moreover, the models extracted by our approaches can be augmented with manual specifications — by cutting down manual specifications to a small minority of instructions, we can achieve significant savings over existing approaches. On the other hand, manual specification of the details may not be needed as most of the applications of the extracted models, such as binary analysis and instrumentation systems, do not need much level of details. Nonetheless, in those rare cases where additional precision is desired, the models extracted by our approach can be manually augmented with the missing details.

Incorrectness of the models stems from the fact that even though modern compilers are tested rigorously, given the limitations of software testing, they are not completely bug-free. Consequently, the semantics obtained from GCC's code generator may have some correctness issues. We will talk about different types of possible correctness issues in the appropriate section of this dissertation. We will also talk about how to rectify these issues as and when relevant. But before getting into challenges, we will talk about our overall approach now.

2.3.3 Approach details

Our overall approach to synthesize instruction set semantics automatically using GCC consists of two phases.

- *Phase 1: extract the instruction set semantics using GCC's code generator.*

For this phase, we have developed two approaches, namely LISC and EISSEC, to extract instruction set semantics from GCC's code generator. Both these approaches are discussed in depth in the Section 3 and Section 4. EISSEC uses symbolic execution of code generator to extract the semantics model. So at a high-level, this approach is similar to symbolic execution of CPU emulator performed by PokeEmu. LISC, on the other hand, employs a completely different technique than EISSEC. Specifically, it treats code generator as a black-box and *learns* the semantic model from it. Since code generators generate semantically-equivalent assembly instruction(s) for the input IR instruction(s), the outcome of this phase is a mapping between semantically equivalent IR-to-assembly pairs for all compiler-supported target assembly instructions. Such a mapping precisely defines the instruction set semantics model followed by the code generator.

- *Phase 2: Checking correctness of extracted instruction set semantics.*

  Even though we are able to extract instruction set semantics from code generators automatically, there is one important challenge that we need to tackle before such semantics can be used in some applications. Specifically, to address the correctness issues, we have developed an approach named ArCheck (Section 5). Since we assume that the IR to assembly instruction pairs present in the model should be semantically equivalent (because compilers are expected to preserve semantic equivalence during compilation), we enforce semantic equivalence test as the correctness check. The outcome of this phase is a new model such that most of the correctness issues from the input model are resolved.

2.4 Summary

In this section, we described various approaches for automatic extraction of instruction-set semantics. We also described various levels of semantic details that one can extract, and how and why they may not be relevant for our purpose in this dissertation. We then described the advantages offered by modern compilers, and why we believe that they are appropriate for extracting instruction-set semantics. Finally, we described our overall approach of semantic extraction using code generators of modern compilers. In the next couple of sections, we will see our semantic extraction approaches in detail.

3 LISC: LEARNING INSTRUCTION SEMANTICS FROM CODE GENERATORS

As we saw in the last section, the direct use of architecture specifications for extracting instruction set semantics model is hard. Moreover, it is undesirable since it would tie the approach of model extraction to a specific compiler. A much better approach for model extraction is to treat compilers (and code generators) as a black-box and to decouple from the specifics of their architecture specifications. Fortunately, the common practice of dumping the inputs and outputs of intermediate stages of modern compilers makes such an approach feasible. To be precise, code generators of modern compilers provide options to log IR instructions and their corresponding assembly instructions generated by the code generators during compilation of source programs. As a result, it seems possible to extract semantics models used by the code generators by referring to their logs. Our approach for semantics model extraction, called LISC (stands for Learning Instruction Semantics from Code generators), is exactly based on these observations.

A sample log obtained from GCC's x86 code generator during the compilation of a typical C code (for fibonacci series) is shown in Figure 4. The log contains x86 assembly instructions and their corresponding RTLs (GCC's IR) instruction. (It is not necessary at this stage to understand details of RTL instructions. So we will not discuss them here.)

```
int fib(int n) {
  if (n ≤ 1)
    return n;
  else
    return fib(n-1) + fib(n-2);
}
```

(a) **C program**

No	Assembly	RTL instruction
1	`pushl %ebp`	`(set (mem:SI (pre_dec:SI (reg:SI esp))) (reg:SI ebp)))`
2	`movl %esp,%ebp`	`(set (reg:SI ebp) (reg:SI esp))`
3	`pushl %ebx`	`(set (mem:SI (pre_dec:SI (reg:SI esp))) (reg:SI ebx)))`
4	`subl $20,%esp`	`(parallel [(set (reg:SI esp) (plus (reg:SI esp) (const_int −20))) (clobber (reg:CC eflags))])`
5	`jg .L2`	`(set (pc)(if_then_else (gt (reg:CCGC eflags) (const_int 0))) (label_ref .L2)(pc))`

(b) **Compilation log**

Figure 4: **Sample compilation log produced by GCC's x86 code generator during compilation of fibonacci's C code**

Given the details of a code generator logs, it is logical to ask: why the logs themselves cannot be treated as semantic models? Specifically, the logs themselves contain semantics of assembly instructions in terms of compiler IRs, and thus it is logical to think that logs can be treated as models also.

Unfortunately, there is one important problem in using logs as models. Specifically, the issue is that the logs contain *concrete* instructions, and to use these logs as a model, we would need huge number of such concrete instructions. Specifically, by concrete instructions, we mean when the operands of the instructions are concrete. For instance, `mov %eax, %ebx` is a concrete instruction, because `eax` and `ebx` are concrete operands. The opposite of concrete instructions would be *parameterized* instructions. For instance, `mov %1, %2` is a parameterized instruction. This is because operands of this instructions are now parameters[4].

It is desirable that the extracted models contain parameterized pairs, otherwise, we would need a pair for every possible operand value and the combinations of different operands. This is because, otherwise, we will not know the semantics of target instructions missing from the log file. For instance, Figure 4 contains a pair with "`subl $20, %esp`" instruction. If we use this pair to generate a model, we will not know the semantics of "`subl $10, %esp`". To solve this problem, we would need to collect 2^{32} similar pairs for different immediate values that can appear in place of 20. While storage space taken by such a huge number of pairs might not be an issue, being able to generate all such pairs is a serious challenge. In contrast, an approach that uses parameterized pairs to generate models would avoid this problem. Specifically, parameterized pairs would allow us to understand the semantics of "`subl $10, %esp`" even though it is missing from the log. `LISC` thus uses parameterized pairs to represent semantic models. The parameterized pairs are produced by *learning* the semantics of target instructions. We will see the details of `LISC` soon.

Unfortunately, `LISC` also faces two challenges. First challenge relates to the *completeness* of the model, while the second one relates to the correctness of the model.

- **Completeness**

  Given the high-level description of our approach, it is easy to see that in order for the extracted models to contain all of the target instructions, the compilation logs need

[4]Intuitively, this can also be thought of as a instruction template which when instantiated with concrete operands makes concrete instruction.

to contain all those instructions. Unfortunately, the compilation logs may not contain all of the target instructions for following reasons:

1. *A source program may not need some of the machine instructions.* Compilation log in Figure 4 demonstrates this point. Compiling a variety of source packages may address this issue.

2. *Compilers might produce some instructions only when specific compilation options are used.*

   For instance, GCC generates x86 SSE instructions only when "`−msse`" option is used. Otherwise, it generates `i387` instructions for floating point computations. It seems that by compiling a variety of source packages with different compilation options, we can possibly increase the number of supported instructions. But we need to experimentally validate our hypothesis.

- **Soundness**

  Unfortunately, the approach of using parameterized pairs in extracted models comes with its own challenge. In particular, the process of generating parameterized pairs from concrete pairs would be *sound* only if the semantics represented by parameterized pairs is exactly the semantics of the corresponding concrete pairs. For instance, if we generate a parameterized pair shown in Figure 5 from the concrete pair shown in the same figure, then it is necessary that we have referred to 2^{32} values[5] that can appear in place of 10. This is because the parameterized rule would represent all such cases, and we need to verify that it is indeed correct. But if we do not have all of 2^{32} concrete pairs, then the process of obtaining parameterized pairs from available pairs can be called as *speculative generalization*. It is speculative because, we are speculating the semantics of missing pairs. Such speculative generalization can be unsound. For instance, it might be possible that, for "`add $−10, %eax`", a compiler may produce an IR which subtracts value 10 instead of adding -10. Intuitively, the solution we are looking for would allow us to keep the scope of generalization narrow enough to avoid unsound mappings, yet broad enough to cover pairs with all possible operand combinations. Moreover, at this stage it is important to mention that generalization is the fundamental property of all learning-based approaches.

[5]assuming 32-bit integers

Concrete IR instruction	Concrete Assembly instruction
`[(set (reg:SI ax) (plus (reg:SI ax) (const_int 10))) (clobber (reg: FLAGS))]`	`add $10, %eax`
Parameterized IR instruction	Parameterized Assembly instruction
`[(set (reg:SI ax) (plus (reg:SI ax) (const_int` **P**`))) (clobber (reg: FLAGS))]`	`add` **$P**, `%eax`

Figure 5: **Concrete pair for x86 `add` assembly instruction and its parameterized version**

3.1 `LISC` Overall Approach

Given the challenges faced by `LISC`, we would now discuss how we address them. `LISC` extracts the semantics model from GCC's code generator logs in three steps described below. The block diagram of `LISC` is shown in Figure 6. Note that our current implementation targets GCC, but it should be possible to extend it to other compilers too.

1. **Extracting IR and assembly pairs.**

   The goal of this step is to extract pairs of IR instructions and the corresponding assembly instructions from the code generator log files.

2. **Parameterization.**

   Once concrete pairs are obtained, the goal of this step is generate parameterized pairs from them. This step helps in producing parameterized pairs by learning how to identify parameters in assembly and RTL instructions. Generally, operands and operators of the assembly and IR instructions are parameters, but we do not encode architecture-specific ways to identify parameters in `LISC` at all. If one can imagine the parse trees of assembly and RTL instruction, then `LISC` follows an approach of treating their leaves as operand instead. Such approach helps in making `LISC` architecture-neutral. Once parameters are identified, they are then replaced by variables to produce parameterized pairs. For instance, in assembly instruction `add %eax, %ebx`, if `LISC` learns that `eax` and `ebx` are parameters, and if it assigns variable `X` for `eax` and `Y` for `ebx`, then the parameterized assembly instruction

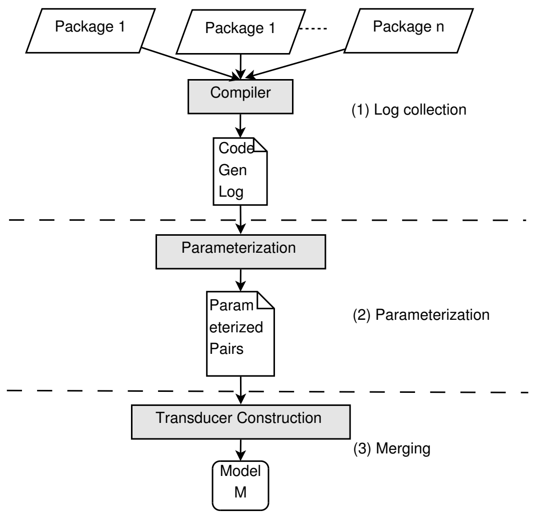


Figure 6: `LISC` **block diagram**

would be “`add %X, %Y`”. Correspondingly substituting `eax` and `ebx` from the RTL `(set (reg ebx) (plus (reg ebx) (reg eax)))`, we would get `(set (reg Y)(plus (reg Y) (reg X)))` as the parameterized RTL. Parameterized pairs (parameterized assembly and its RTL) provide a way to represent multiple concrete pairs compactly. These pairs then form the extracted semantics model.

3. **Transducer construction.**

   A set of parameterized pairs can be considered as a extracted model. But we observed that many parameterized pairs have similar syntactic structures or sub-structures. For instance, both “`add %eax, %ebx`” and “`add %ecx, %edx`” produce same parameterized assembly (“`add %X, %Y`”), and thus have same syntactic structure. To eliminate such duplications, one can extract out common parts across pairs and only consider uncommon parts. Such optimization essentially helps in generating compact semantics models. We call this notion as *merging* and implement it by developing an approach to construct a transducer (automata with outputs) to merge parameterized pairs together. The output of this step is the final extracted semantics model.

We will now describe each of these steps in detail.

3.2 Extracting IR and Assembly Pairs

We first use GCC to compile many source code packages and collect concrete pairs of IR instructions and the corresponding assembly instructions from its code generator log files. Earlier we gave an example of pairs obtained from a compilation log of C code for Fibonacci series. All such captured IR-to-assembly pairs are then fed to the next step in the process.

3.3 Parameterization

The goal of parameterization is to generate a parameterized pair of IR and assembly instruction from their concrete pair. Such a generation of parameterized pairs demands that parameters in IR and assembly are identified and also that the mapping functions between parameters are identified. For instance, to produce ⟨"`add $X, %Y`", `(set (reg Y)(plus (reg Y) (const_int X)))`⟩ parameterized pair from ⟨"`add $10, %eax`", `(set (reg eax)(plus (reg eax) (const_int 10)))` ⟩ concrete pair, we first need to identify that `eax` and `10` are parameters. Moreover, we need to identify that immediate value in assembly instruction (10) is same as that in the RTL (10); in other words, the mapping functions between immediates is the identity function ($f(x) = x$). Identifying the mapping functions between parameters is necessary so that parameterized pair can be used to generate a concrete IR or assembly when the extracted semantics model is used in some application. For instance, when we use above parameterized pair to translate "`add $20, %ebx`" assembly instruction (from a disassembled binary) into RTL, we will simply *bind* variables `X` and `Y` from the parameterized pair with `20` and `ebx` resp, and apply the mapping functions for both of them to produce `(set (reg ebx)(plus (reg ebx)(const_int 20)))` as the concrete RTL.

Parameterization step thus has two objectives: (1) identify parameters in the instructions, and (2) find out how input parameters are mapped to output parameters. To achieve these objectives, `LISC` operates on the syntactic structures of the concrete assembly and RTL instructions. We use parse trees to represent syntactic structures. The tree structure allows grouping of related information together, and it also simplifies the task of finding common nodes/subtrees between multiple trees.

Construction of first-order terms (parse trees). Instead of detailing out how we build parse trees for assembly and IR instructions, in Figure 7a we specify the grammar for representing these trees as first-order terms. In the grammar, we have considered assembly instructions with limited types of operands such as register, memory, and immediates[6]. Additionally, for memory, we consider only two types of operands: indirect reference via a register and indirect reference via a register plus some constant. To simplify our discussion here, we have restricted grammar to the basic set of operands. RTL syntax is slightly different from assembly instructions but semantically both are close. Note the use of *1 and *2 in memory operations of assembly instruction. These functors encode the arity of the terms, and are used to distinguish uses of the same operator with different number of operands. Subterms with zero arity are leaves in the parse tree, so we call such terms as *leaf terms*. Examples of concrete pairs for x86 which follow this grammar are shown in Figure 8a.

Parameter identification. Once IR and assembly instructions are represented as first-order terms, the next step is to identify parameters in these terms. Note that any field of a term can be a parameter. For the purpose of such identification, one can simply encode the knowledge of what is a parameter into the system. Specifically, one can encode names of architecture registers and use them to identify which fields of a term are registers (e.g., `eax` is a parameter). Unfortunately, such an approach is architecture-specific. `LISC` thus avoids such approach and instead relies on a simple approach of treating leaf terms as potential parameters. Such an approach does not treat non-leaf terms such as “`(*1 (reg eax))`” (which stands for assembly operand “`(%eax)`”) as parameters. We found such treatment of parameters enough for the purpose of our problem.

Identifying the mapping functions. Once the parameters in IR and assembly terms are identified, the next step is to learn how parameters in assembly maps to those in the RTL. Intuitively, the objective of this step is to find out how the semantics of an assembly instruction is mapped by its corresponding RTL instruction. Concretely, mappings between

[6]Note that `LISC` implementation supports all types of operands of RTL and assembly instructions such as floating point value, strings, etc.

$$
\begin{array}{rcl}
\mathit{Pair} & ::= & \langle \mathit{Assembly}\ \mathit{RTL} \rangle \\
\\
\mathit{Assembly} & := & (\mathit{Id}\ \mathit{AsmOpList}) \\
\mathit{AsmOpList} & := & /*\mathit{empty}*/ \mid \mathit{AsmOp}\ \mathit{AsmOpList} \\
\mathit{AsmOp} & ::= & \mathit{Reg} \mid \mathit{Mem} \mid \mathit{ConstInt} \\
\mathit{Reg} & ::= & \mathit{STRING} \\
\mathit{Mem} & ::= & (*1\ \mathit{Reg}) \mid (*2\ \mathit{Reg}\ \mathit{ConstInt}) \\
\mathit{ConstInt} & ::= & \mathit{INT} \\
\mathit{Id} & := & \mathit{STRING} \\
\\
\mathit{RTL} & ::= & (\mathit{Id}\ \mathit{RTLOpList}) \\
\mathit{RTLOpList} & := & /*\mathit{empty}*/ \mid \mathit{RTLOp}\ \mathit{RTLOpList} \\
\mathit{RTLOp} & ::= & \mathit{Reg}' \mid \mathit{Mem}' \mid \mathit{ConstInt}' \\
\mathit{Reg}' & ::= & (\texttt{reg}\ \mathit{STRING}) \\
\mathit{Mem}' & ::= & (\texttt{mem}\ \mathit{MemOp}') \\
\mathit{MemOp}' & ::= & (\mathit{Reg}') \mid (\texttt{plus}\ \mathit{Reg}'\ \mathit{ConstInt}') \\
\mathit{ConstInt}' & ::= & (\texttt{const\_int}\ \mathit{INT})
\end{array}
$$

(a) **Grammar for concrete pairs**

$$
\begin{array}{rcl}
\mathit{Reg}' & ::= & (\texttt{reg}\ \mathit{STRING}\ \mathbf{MapFuncs}) \\
\mathit{ConstInt}' & ::= & (\texttt{const\_int}\ \mathit{INT}\ \mathbf{MapFuncs}) \\
\\
\mathbf{MapFuncs} & := & [\,] \mid [\mathit{MapFunc}] \\
\mathbf{MapFunc} & ::= & \mathit{Eq} \mid \mathit{Plus}\ \mathit{ConstInt}' \mid \mathit{Minus}\ \mathit{ConstInt}' \mid \mathit{Mult}\ \mathit{ConstInt}' \mid \\
 & & \quad \mathit{Div}\ \mathit{ConstInt}'
\end{array}
$$

(b) **Grammar for parameterized pairs** (updated with support for representing mapping functions)

Figure 7: **Grammar for concrete and parameterized pairs**

Pair 1	`(add 20 eax)`	`(set (reg eax) (plus (reg eax) (const_int 20)))`
Pair 2	`(add 10 eax)`	`(set (reg eax) (plus (reg eax) (const_int 10)))`
Pair 3	`(add (*1 ebx) eax)`	`(set (reg eax) (plus (reg eax) (mem (reg ebx))))`
Pair 4	`(add (*2 ebx 40) eax)`	`(set (reg eax) (plus (reg eax) (mem (plus (reg ebx) (const_int 30)))))`
Pair 5	`(mov 10 eax)`	`(set (reg eax)(const_int 10))`
Pair 6	`(mov ebx eax)`	`(set (reg eax)(reg ebx))`
Pair 7	`(mov (*1 ebx) eax)`	`(set (reg eax)(mem (reg ebx)))`

(a) **Concrete pairs**

Pair 1	`(add X Y)`	`(set (reg (Eq Y)) (plus (reg (Eq Y)) (const_int (Eq X))))`
Pair 2	`(add X Y)`	`(set (reg (Eq Y)) (plus (reg (Eq Y)) (const_int (Eq X))))`
Pair 3	`(add (*1 X) Y)`	`(set (reg (Eq Y)) (plus (reg (Eq Y) (mem (reg (Eq X)))))`
Pair 4	`(add (*2 X Y) Z)`	`(set (reg (Eq Z)) (plus (reg (Eq Z)) (mem (plus (reg (Eq X)) (const_int (Minus Y 10))))))`
Pair 5	`(mov X Y)`	`(set (reg (Eq Y))(const_int (Eq X)))`
Pair 6	`(mov X Y)`	`(set (reg (Eq Y))(reg (Eq X)))`
Pair 7	`(mov (*1 X) Y)`	`(set (reg (Eq Y))(mem (reg (Eq X))))`

(b) **Parameterized pairs**

Figure 8: **Examples of concrete and parameterized pairs for a subset of x86**

assembly and RTL instructions are mapping functions between their terms. For instance, we can see that the term for a register `eax` in assembly maps to the term (`reg eax`) in RTL.

In general, two values can be mapped using a number of mapping functions, but fortunately, we do not need to find out all of the possible mapping functions. Instead, we can rely on the common observations about assembly and IR instructions to restrict them to few. Specifically, different types of values used in assembly/IR instructions follow different mapping functions: most parameter types such as register names, floating point values, immediates, string, etc, almost always follow equality relation (which we designate as $=$). Registers sometimes might follow subtype relation where in the output register is a subregister of the input register (e.g., if `ax` is the output register and `eax` is the input register, then we have a subtype relation.) In addition to equality relation, numeric values such as integers and floats might follow few others, such as addition of a constant, subtraction, multiplication, or division of input with some constant value. We need to consider additional mapping functions for numeric values because, even though it is mostly the case that constants appear unchanged between IR and assembly, it may not always be true. Given the description about mapping functions, the problem of finding a mapping function f between two values x and y can be formulated as $f \mid f(x) = y$, and in our implementation, we restrict relations between x and y to $y = x$, or $y = x\ op\ C$, where op is one of $+$, $-$, $\times$, and $\div$, and C is some constant value. Note that multiple mapping functions are possible between two values. For instance, if an assembly instruction in a pair contains 10 (represented by variable x) and the corresponding IR contains 20 (represented by variable y), then there are two possible mapping functions: $y = x\ +\ 10$ and $y = x\ \times\ 2$. Our goal thus is to find out all of the possible mapping functions. This is needed because by just processing one pair we do not know which one is the exact mapping function for that pair. When we will come across more instances of similar pairs, then we can find out which mapping function actually holds. For instance, if we come across same pair with assembly containing 30 (represented by x) and IR containing 40 (represented by y), then it is more likely that $y = x\ +\ 10$ holds over $y = x\ \times\ 2$. So, to begin with, we will find out all possible mapping functions between two values, and later prune out those that do not hold. Such an approach ensures that we do not lose our ability to generalize by being able to find common mapping functions between pairs.

Algorithm and its description. Once we can represent all subterms of a term by variables, we systematically visit all of the assembly's leaf terms, and check if they can be mapped with any of the RTL's leaf terms. If such mappings are found, RTL's leaf terms are updated with the found mapping functions. Specifically, if y is one of the RTL's leaf terms, and x is one of the assembly's leaf terms, and $y = f(x)$, where f is some mapping function, then y is updated with $f(X)$, where X is the variable that represents any value that appears in the position of x. Note that since leaf terms of an RTL's parse tree might be updated to capture the mapping function, we need to update our grammar to support these functions. Grammar productions which are updated to capture mapping functions are shown in Figure 7b, with newly added productions shown in bold (rest of the productions are same as that of grammar for concrete pairs). Also note that mapping functions are added to RTL's leaf terms, while assembly's leaf terms remain same. Lastly, in RTL, we have updated definitions of registers and constant integers only; mapping functions are not added to `mem` node because memory is not a leaf term. To illustrate mapping functions: for assembly instruction "`add $10, %eax`" when the RTL instruction contains `20` instead of `10`, then it can be learned that the RTL adds 10 to the value that appears in place of 10 (in the assembly instruction) or it multiplies it by 2. If we represent parameter 20 in the RTL by `Y` and parameter 10 in the assembly by `X`, then both these mapping functions for `Y` are captured as [`Plus X 10`; `MultX 2`]. ; is used to indicate that multiple mapping functions are possible. Our general policy for finding the mapping functions also handles tricky cases such as different number of input and output parameters, an input parameter appears at multiple output positions, etc.

Given a list of variables and input and output leaf terms represented by those variables, the algorithm to identify mapping functions between them is shown in Figure 9. For each of the output terms, the algorithm checks to see if it is some function of any input term. If such a function is found, then it is recorded as the mapping between that output term and the position of the input term. One notation might need a bit clarification. + operator is used to represent list concatenation. Note that t'_o is a variable to represent list of terms. Thus, + allows us to combine multiple mapping functions in a term. This is needed when term contains "`Eq X`" already and we want to add "`Div Y 2`" to make "[`Eq X` ; `Div Y 2`]". When the algorithm is applied to concrete mapping pairs from Figure 8a, we get parameterized pairs from Figure 8b.

3.4 Transducer Construction

The output of parameterization is a set of parameterized pairs for input concrete pairs. But since parameterization produces a single parameterized pair for every concrete pair, we will essentially get as many parameterized pairs as that of concrete pairs. This is undesirable since it would mean a huge number of parameterized pairs. A more serious issue is the matching time of the model constructed using these pairs, which would be linear in terms of number pairs.

```
// $\mathscr{I}$ is a set of pairs of a variable and a leaf term of an input term.
// $\mathscr{O}$ is a set of pairs of a variable and a leaf term of an output term.
// Algorithm returns a set of all mapping functions between terms of $\mathscr{I}$ and $\mathscr{O}$.

procedure find_mappings($\mathscr{I}$, $\mathscr{O}$):
$\mathscr{M} = \{\}$ // set of mapping functions
$\mathscr{C}_{AS} = \{-16,..,16\}$ // Constants for addition and subtraction function
$\mathscr{C}_{M} = \{-65536,..,65536\}$ // Constants for multiplication function
$\mathscr{C}_{D} = \{1,..,65536\}$ // Constants for division function
foreach $\langle v_o,t_o\rangle \in \mathscr{O}$
do
    $t'_o = [t_o]$   // $t'_o$ is a local variable to represent a list of terms
    foreach $\langle v_i,t_i\rangle \in \mathscr{I}$
    do
      if $t_o = t_i$ then
            $t'_o = t'_o + [Eq\ v_i]$ // $+$ is a function for list concatenation
      if $t_o = t_i + c_a$, where $\exists\ c_a \in \mathscr{C}_{AS}$ then
            $t'_o = t'_o + [Plus\ v_i\ c_a]$
      if $t_o = t_i - c_s$, where $\exists\ c_s \in \mathscr{C}_{AS}$ then
            $t'_o = t'_o + [Minus\ v_i\ c_s]$
      if $t_o = t_i * c_m$, where $\exists\ c_m \in \mathscr{C}_{M}$ then
            $t'_o = t'_o + [Mult\ v_i\ c_m]$
      if $t_o = t_i / c_d$, where $\exists\ c_d \in \mathscr{C}_{D}$ then
            $t'_o = t'_o + [Div\ v_i\ c_d]$
      // if none of the above conditions match, then unsupported function
      fi
    done
    $\mathscr{M} = \mathscr{M} \cup \{\langle v_o,t'_o\rangle\}$
done
return $\mathscr{M}$
```

Figure 9: **Pseudo code for identifying mapping functions between values of input and output parameters**

A more efficient approach to construct a model that has matching time which is independent of the number of pairs is to construct a DFA (deterministic-finite state automata). Since many parameterized pairs have same syntactic structure, the automata construction can extract out common parts across pairs. For instance, note that pair 5 and 6 in Figure 8b have same syntactic structure of `(set (reg (Eq Y)), Z)`, where `Z` represents parameter values where they differ. So, intuitively, it seems that we can extract out common parts between pairs, and generate checks to identify uncommon parts. For instance, we can generate check such as `if X = ebx then Z = (reg ebx) else Z = (const_int 10)` to distinguish pair 5 and 6. Extraction of common part across pairs helps in two ways: (1) it avoids duplication, and it allows us to emit parts of an output even without looking at all the parts of an input (for instance, in the above example, we can output `(set (reg (Eq Y))` even without looking at the input), (2) the reduction in the size of the automata reduces the size of the extracted semantics model.

Tree automata (an automata which operates on trees) is a DFA which accepts trees as input. Tree automata which produces output trees in addition to accepting the input trees is called as tree transducer. Since we generate IR parse trees as output by accepting assembly parse trees as input, our problem can be classified as the problem of constructing a tree transducer from the set of parameterized pairs.

A tree transducer constructed for the example discussed above is shown in Figure 10. Circular nodes in the graph indicate the input variables that are inspected, while the edge labels represent the test being performed for those input variables. For instance, edge labelled $= 10$ checks if value of variable X is 10. Box type nodes, on the other hand, represent parts of the output that will be emitted by the transducer when the input has matched all the tests from that node to the start node. Double-bordered box nodes represent the end states of the transducer in which all the input variables have matched some assembly term and thus the transducer has produced corresponding IR term as output. Intuitively, what we have done here is to emit common part of the output between both the pairs (i.e., `(set (reg eax) Z)`, and we emit it immediately when we know that the assembly instruction is `mov` and its parameter for Y is `eax`.

Tree transducers offer two important advantages for our problem. First, Being able to extract out such common parts across pairs helps in reducing input matching time. For instance, if we were to match the input term with the assembly terms of the above two pairs, then we would need to do it one by one (sequential matching). The matching time complexity in this case would be $O(n)$ where n is the number of pairs. The matching

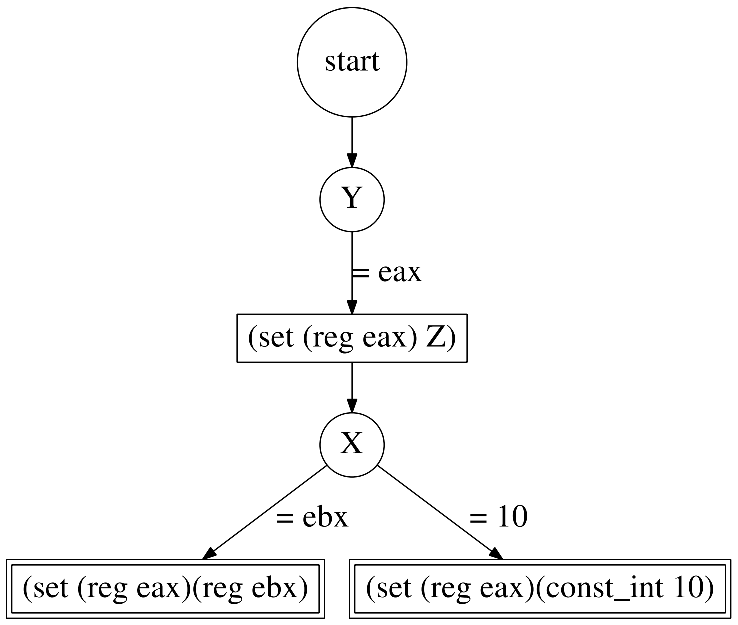


Figure 10: **A tree transducer for the example**

complexity of tree transducer from the figure, on the other hand, is *independent* of the number of pairs, and is proportional to the length of the input. Second, by building a transducer we can match the input in one scan — in case of sequential matching, we need to revisit the input when the match has failed for the first pair.

3.4.1 Background

Before we discuss our algorithm to construct tree transducer from the set of parameterized pairs, we need to develop the notations and concepts that will be used in the algorithm. We assume familiarity with the basic concept of a *term*. The symbols in a term are drawn from a nonempty set of alphabets Σ and a countable set of variables $\mathscr{X}$. We will use w, x, y, and z (with or without subscripts) to denote variables and f to denote non-variable symbols (also called as *function symbol*). Additionally, we use t and u to denote a term.

Term t over Σ can then be formed according to following syntactic rules:

$$t ::= x, f(t_1, t_2, \ldots, t_n),\ \textit{where}\ x \in \mathscr{X}\ \&\ f \in \Sigma$$

Parameter positions. In order to refer to subterms of a assembly or IR terms, we develop the concept of a position. For instance, for `(add 2 eax)`, we need a way to designate 2 uniquely. Then only we can say that value v that appears at the position of 2 from the assembly follows some mapping function, say f, and maps to $f(v)$ in RTL. Fortunately, inductive nature of term structure provide an easy way to uniquely identify subterms. We call unique identifier for each subterm as its *position* [80].

Definition 1 (Position) *A position of a subterm t, denoted P_t, in term T is a string represented as $p.i$, where p is a position and i is an integer. Position p satisfies all of the following criteria: (1) if T is a term for assembly instruction, then its root symbol has $p.i = 1$, (2) if T is a term for RTL instruction, then its root symbol has $p.i = 2$, (3) otherwise, if T is of form $f(t_1, t_2, \ldots, t_n)$, where $n > 0$, and p is position of T, then $p.1$ is position of t_1, $p.2$ is position of t_2, and $p.n$ is position of t_n.*

String representation of P_t is obtained by prefixing '@' to P_t.

Note that in the context of our problem, set $\mathscr{X}$ of variables is a set of strings representing positions. For instance, one value of $\mathscr{X}$ could be $\{@1.1, @1.2, @1.3\}$.

A *ground* term is a term with no occurrences of variables. A set of all ground terms of t in Σ is represented as $\mathscr{T}_\Sigma(t)$.

Lastly, $\top$ and $\bot$ are special terms such that $\mathscr{T}_\Sigma(\top)$ is the universal set of terms over Σ and $\mathscr{T}_\Sigma(\bot)$ is the empty set of terms on Σ.

To illustrate terms and ground terms:

- `(mov eax ebx)` is a term representing assembly instruction, while `(set (reg eax (Eq @1.1)) (reg ebx (Eq @1.2)))` is a term representing RTL instruction for that assembly instruction. Note that term for the assembly instruction is a ground term since it does not contain any variables, while the term for RTL instruction is not a ground term.

Now that we have defined how to represent parameterized pairs as terms, we now need to define how can we merge two terms together. Intuitively, a term which is general than the terms that are being merged together should be the result of merging. In order to find such a general term, we need to define the notion of substitution.

A *substitution* is a mapping from variables to terms. Given a substitution δ, we denote by $\delta(t)$ the term obtained by replacing every variable x in t by $\delta(x)$. We say that t is an *instance* of u if $\delta(u) = t$ for some substitution δ. If t is an instance of u, then we denote it by $t <= u$. Intuitively, if $\delta(u) = t$, then u is *more general* than t, or t is *more specific* than u. On the other hand, if t and u differ only in the variable names, then they are said to be equal to each other, and denoted as $t = u$. Also, $\top$ and $\bot$ are special terms such that $\top$ is more general than any other term, and $\bot$ is more specific than any other term.

Substitutions can also be written as $[x_1 \mapsto t_1, x_2 \mapsto t_2, \ldots, x_n \mapsto t_n]$, with x_i pairwise distinct, and then mapping function δ can be denoted as:

$$[x_1 \mapsto t_1, x_2 \mapsto t_2, \ldots, x_n \mapsto t_n](y) = \begin{cases} t_i, \; if \; y = x_i \\ y, \; otherwise \end{cases}$$

In some instances, we write $x\delta$ for $\delta(x)$.

To illustrate substitution and instances of a term,

- $[x \mapsto eax, y \mapsto ebx]$`(set (reg x) (reg y))` = `(set (reg eax)(reg ebx))`.

- $[x \mapsto eax]$`(set (reg x) (reg y))` = `(set (reg eax)(reg y))`.

- $[z \mapsto eax]$`(set (reg x) (reg y))` = `(set (reg x)(reg y))`.

The notion of substitution now allows us to define partial ordering among terms, and leads to forming of a lattice over terms. We will denote such lattice by $\mathscr{L}$. A sample lattice for a set of parameterized assembly instructions is shown in Figure 11.

Now that we have described how to construct a lattice over a set of terms for parameterized rules, we can now describe how to compute *mcp* and *residue*.

3.4.2 Maximal common prefix (mcp)

Intuitively, computing *mcp* of parameterized pairs can be considered as synonyms to union operation on sets — as union operation allows one to merge multiple sets together, *mcp* computation allows us to merge multiple parameterized pairs together. Note that mcp computation would be *sound* only if the pair obtained by combining multiple pairs captures the complete semantics of all those pairs. Union operation satisfies this criteria.

Definition 2 (mcp) *Conceptually,* mcp *of terms t_1 and t_2, denoted as* $\mathrm{mcp}(t_1, t_2)$*, is the least-upper-bound of t_1 and t_2 in lattice $\mathscr{L}$.*

Formally, $\mathrm{mcp}(t_1, t_2)$ *is another term t such that both of the following conditions are satisfied:*

$$t_1 <= t \; and \; t_2 <= t, \; and \; t <= t', \; \forall t' \mid t_1 <= t' \wedge t_2 <= t'$$

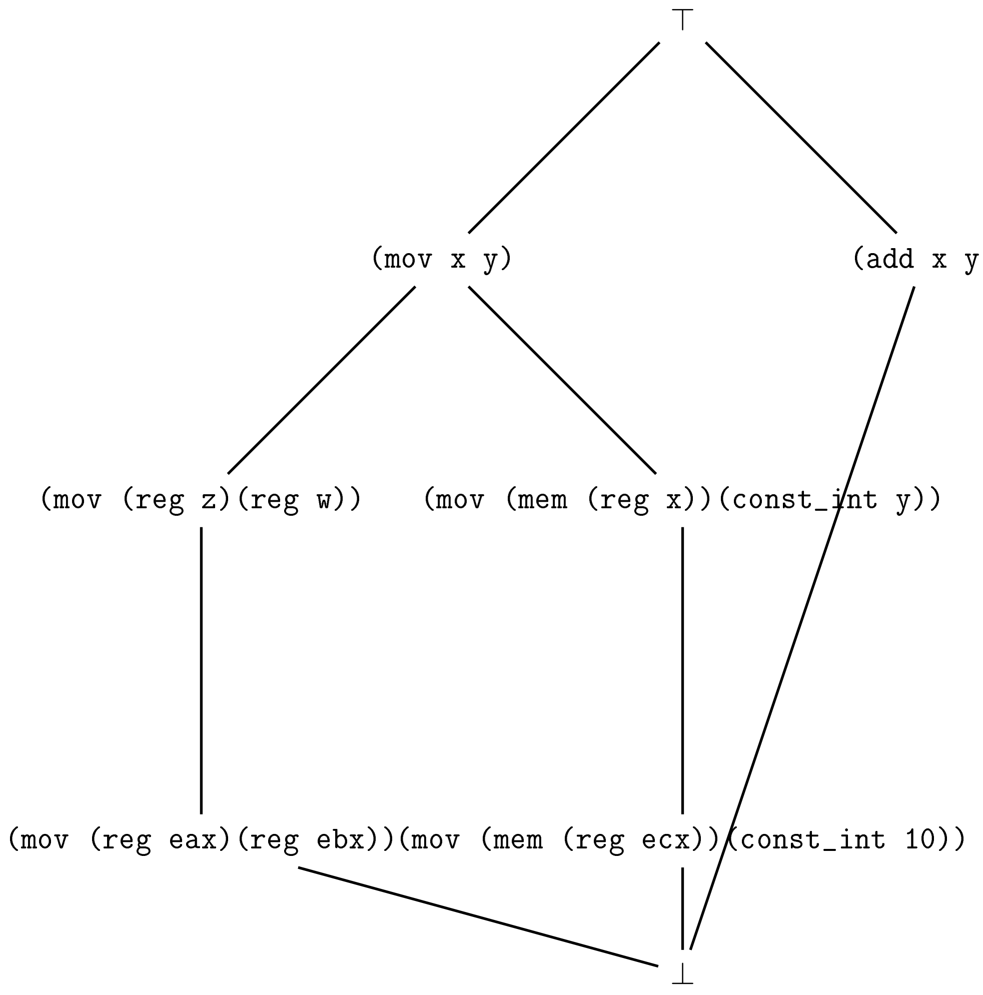

⊤
(mov x y)
(add x y)
(mov (reg z)(reg w))
(mov (mem (reg x))(const_int y))
(mov (reg eax)(reg ebx))
(mov (mem (reg ecx))(const_int 10))
⊥

Figure 11: **Lattice over a set of parameterized x86 assembly instructions**

3.4.3 Residue

Since residue is the uncommon part of a term with respect to another term, we can define residue as follows.

Definition 3 (Residue) *Residue of a term t_1 with respect to another term t_2, denoted as residue(t_1, t_2), can be defined as:*

$$residue(t_1, t_2) = \delta \iff \delta(t_1) = t_2$$

Residue of a set of terms $\mathscr{T}$ with respect to some term t' can be defined as

$$\{residue(t,t') \; \forall \; t \; \in \; \mathscr{T}\}$$

To illustrate *mcp* and *residue*, consider following set of terms.

- **t1**: (`set` (`reg eax`) x)

- **t2**: (`set` (`reg eax`) (`reg ecx`))

- **t3**: (`set` (`reg eax`) (`const_int 10`))

- **t4**: (`mem` (`reg eax`) (`const_int 10`))

Then

- $mcp(t_1,t_2) = t_1$, and $residue(t_1,t_2) = [x \mapsto$(`reg ecx`)$]$
- $mcp(t_1,t_3) = t_1$, and $residue(t_1,t_3) = [x \mapsto$(`const_int 10`)$]$
- $mcp(t_1,t_4) = \top$, and $residue(t_1,t_4) = []$
- $mcp(t_2,t_1) = t_1$, and $residue(t_2,t_1) = []$
- $mcp(t_2,t_2) = t_2$, and $residue(t_2,t_2) = []$
- $mcp(t_2,t_3) =$ (`set` (`reg eax`) x), and $residue(t_2,t_3) = []$
- $mcp(t_3,t_1) =$ (`set` (`reg eax`) x), and $residue(t_3,t_1) = []$
- $mcp(t_3,t_4) = \top$, and $residue(t_3,t_4) = []$
- $mcp(t_4,t_1) = \top$, and $residue(t_4,t_1) = []$
- $mcp(t_4,t_3) = \top$, and $residue(t_4,t_3) = []$

Note that actually $mcp(t_3,t_4)$ could have been (x (`reg eax`)(`const_int 10`)), but since our grammar allows variables (positions) to appear only in leaves of the parse trees, $mcp(t_3,t_4)$ is $\top$.

Speculative generalization. Notion of *mcp* essentially introduces generalization in the extracted semantics model. And, naturally, generalization can compromise soundness of the extracted models. For instance, *mcp*((`set (reg eax) (reg ebx)`, (`set (reg eax) (reg ecx)`)) is (`set (reg eax)(reg` *x*). By having variable *x* represent any registers beyond set {`ebx`, `ecx`}, we are essentially generalizing the rule. Nonetheless, we are also loosing the information that we have seen the IR to assembly pair for only `ebx` and `ecx`. So saying that the RTL is applicable to `edx` (while we have not seen it for `edx`) can lead to unsound RTL.

Note that if we want to ensure that generalization does not compromise soundness at all, then we need to remember the values of the operands for which every pair holds. So, for instance, in the above example, we would need to remember that *mcp* holds only for {`ebx`, `ecx`}. Unfortunately, maintaining sets of values can lead to situations where these sets contain plenty of elements. For instance, for immediates, there could be a huge number of elements in the set. More importantly, this leads to problem of *over-fitting* as it is called in machine learning domain. In particular, if we simply remember the values of the operands for which pair holds, then we are not allowing generalization at all. Such pairs will thus be applicable to pairs we have already seen (training data) and will not generalize well to unseen pairs (test data).

A solution for keeping the set size restricted and also not to lose the information is to use intervals to represent sets. Specifically, after ordering elements of a set, we can take the first and last element of a set to form a interval. For instance, {3, 1, 4, 10, 5, 12, 100} can be represented compactly as [1, 100]. Intuitively, intervals introduce a restricted form of generalization. That is why using intervals are much better (in terms of soundness) than saying that a pair is applicable to any integer value, which would be same as saying we have seen values in the range of $[-\infty, \infty]$ (or for a 32-bit integer [INT32_MIN, INT32_MAX] to be precise).

Additionally, we rely on some observations about operand usage across architectures to ensure soundness even with speculative generalization. To be precise, it seems that one can classify operands of assembly instructions into various functionally-similar categories. By functionally-similar categories, we mean operands belonging to these categories function similarly when used in instructions. For instance, in `add $20,%eax` and `add $1,%eax`, both `1` and `20` perform similar role of immediate operands; in many x86 instructions registers `eax`, `ebx`, `ecx` and `edx` can be used interchangeably. We put this observation to use by keeping operands of the same type in the intervals.

Additionally, to keep the scope of generalization restricted, we define *mcp* as the *least* upper bound of two terms, and not any other upper bound.

Lastly, we will talk about our soundness test in the evaluation section.

3.4.4 Discriminating tests

Intuitively, discriminating tests help us distinguish uncommon parts of mapping pairs from one another. There could exist many discriminating tests for a given set of pairs. For instance, for a pair of RTLs `(set (reg eax) (reg ebx))` and `(set (reg eax) (plus (reg eax)) (reg ecx))`, one could distinguish them on the basis of the structure of the pairs (note that `plus` subtree does not exist in the first RTL), presence of `ecx` register in the second pair, etc. We would then need to decide which tests to prefer over others. One more important point to note is that the discriminating tests need not always partition a set of uncommon parts in two, but they could partition a set in more than two. In other words, discriminating tests need not only be binary tests like *does rule contain ecx*, but instead can be *N*-ary tests like *partition pairs into those having mnemonic* `mov`*,* `add`*, and* `sub`.

Note that the important requirement of discriminating tests is that these tests must create disjoint subsets of a set of uncommon parts. This requirement is needed in order to keep a polynomial bound on the size of the extracted model in the worst case. Specifically, for a set of *N* uncommon parts, if discriminating tests do not create disjoint subsets (such tests would actually be called as non-discriminating tests), then in the worst case they can lead to creation of 2^N different subsets (corresponding to the number of possible (overlapping and disjoint) subsets) of the set of *N* uncommon parts. In such case, size of the extracted model would have non-polynomial worst case complexity. That is why we need discriminating tests to be such that these tests would produce maximum of *N* disjoint subsets for a set of *N* uncommon parts.

Given the requirement of discriminating tests, we will now see how do we satisfy it. We follow a specific order of preferences for choosing discriminating tests:

1. *Mnemonic of an assembly instruction.* If residues of pairs differ in their assembly mnemonic, then we discriminate them based on assembly mnemonic.

2. *Number of operands of an assembly instruction.* Instructions having same mnemonic are then partitioned on the basis of a number of operands.

3. *Operand arity (rule structure).* Once pairs are partitioned on the basis of assembly

mnemonic and the number of operands, we then look at the arities of operands. Arity of an operand is the arity of the functor term for that operand. For instance, `8(%eax)` has two operands, while immediate `8` has only one operand.

4. *Operand type.* Once operands are split based on arity, operands of same arity are then split based on their type — whether operand is string or integer, etc. For operands of arity more than one, we apply operand type test on their sub-fields recursively in the left-to-right order till we find first discriminating test. Employing operand type test also falls in line with the notion of functionally-similar operands.

5. *Operand value.* Lastly, all pairs whose operands are indistinguishable after applying above four tests, are split based on the values of their operands.

The particular order of choosing discriminating tests arise from the observation that we do not want to merge pairs having different syntactic structures. That is why first three tests above look for syntactic differences. Even the fourth test can be considered as following the same principle. This is because, in RTL, operands of different types have different syntactic structure. For instance, integers are represented as (`const_int 10`) while strings are represented as (`reg eax`) or (`stringref "printf"`). Once we have exploited all syntactic differences to discriminate pairs, we are left with no other choice but to look at operand values to distinguish.

Also note that when we discriminate based on operand values, there might exist multiple operands positions where values could differ. When such a case arises we have multiple choices and then a question arises about which operand position to select. Our criteria in such case is guided by our requirement to keep the size of extracted semantics model minimum. To satisfy this requirement, we follow a simple greedy strategy of choosing an operand position which would lead to a minimum number of discriminating tests. Note that such a strategy may not always produce the minimal-size semantics model, but given the simplicity of the approach we prefer it over others.

Also note that choices 1–4 are actually discriminating tests. This is because all of these tests *always* divide the input set into disjoint subsets. There is a chance that the last choice of splitting based on operand value can lead to non-discriminating test, if the test for splitting based on value is not chosen properly. To ensure that such case does not arise, we follow a simple approach of ordering the operand values by defining an ordering operator (such as <), and splitting the ordered set such that first half of the ordered set forms one set,

```
procedure $Build(s_c, m, \mathscr{R})$:
1. if $\mathscr{R}$ is empty then
2.     $final[s] = m$
3. else
4.     $\mathscr{T} = select(\mathscr{R})$
5.     for each $T_i \in \mathscr{T}$ do
6.         $\mathscr{R}_i = \{R \in \mathscr{R} \mid T_i(R) = true\}$ // Partitioning of $\mathscr{R}$
7.         $\mathscr{R} = \{R \in \mathscr{R} \mid T_i(R) = false\}$ // Consider remaining residues for next test
8.         $m_i = mcp(\mathscr{R}_i)$
9.         $\mathscr{R}_{s_i} = residue(\mathscr{R}_i, m_i)$
10.        if state $s_i$ corresponding to $(m_i, \mathscr{R}_{s_i})$ is not present then
11.            Construct state $s_i$
12.            $Build(s_i, m_i, \mathscr{R}_{s_i})$
13.        fi
14.        Construct an edge from $s_c$ to $s_i$ labeled with $T_i$.
15.    done
16. fi
17. return $s_c$
```

Figure 12: **An algorithm for constructing automaton**

and second half forms another. Generation of two disjoint subsets ensures that such test is always a discriminating test.

3.4.5 Algorithm for constructing automaton

The algorithm *Build* for constructing a tree transducer is shown in Figure 12. Note that tree transducer is a generalization of a decision tree, which would have been a straightforward representation of merged pairs (edges in the tree would represent discriminating tests selected.) We specifically chose automaton over decision tree because automaton opens up the possibility of sharing nodes. States in our transducer represent *mcp* of input pairs, with final states representing a complete pair learned by `LISC`. An important property of this transducer is that it eagerly emits output — as soon as some parts of the output are known, it is emitted, without having to wait to reach a final state.

Build starts with a dummy start state of the transducer (s_c), *mcp* (m) of the set of parameterized pairs to be merged, and their *residue*($\mathscr{R}$). Specifically, first call to *Build* would be $Build(s_c,\ mcp(\mathscr{I}),\ residue(\mathscr{I}, nil))$, where $\mathscr{I}$ is the set of parameterized pairs to be merged. Eventually, *Build* returns a start state s_c as the output. *Build* is a recursive procedure which then constructs sub-transducer rooted at s_c. When $\mathscr{R}$ is empty, we construct final state of the transducer which is marked by part of the output (*mcp*) accepted by the state.

When $\mathscr{R}$ is not empty, it means we still have some part of pairs to be processed. In that case, we first call *select* to obtain a set $\mathscr{T}$ of discriminating tests to distinguish "uncommon" parts in $\mathscr{R}$. For every test T_i from $\mathscr{T}$, we partition $\mathscr{R}$ into two sets by applying the test. Elements of $\mathscr{R}$ which do not satisfy the test are considered for further partitioning by remaining tests of $\mathscr{T}$ in the next iterations. We obtain *mcp* and *residue* of $\mathscr{R}_i$, the partition which contains all elements of $\mathscr{R}$ which passed the test T_i. Since we construct automaton instead of a decision tree, we can share common states of the automaton. We do it by checking if a state corresponding to the newly obtained *mcp* and *residue* is already generated, and if it is, then we simply obtain that state and generate a transition from the current state to that state labeled by the test T_i. In case such state is not present already, then we create it, and call *Build* again to explore sub-automaton rooted at newly created state.

To illustrate the algorithm, consider the set of parameterized pairs shown in Figure 13a. The transducer constructed for these set of pairs is shown in Figure 13b. Notice that in all the pairs the part of the output i.e., (`set(reg eax & @12`) is common, and that is why the algorithm emits it as the *mcp* before input is seen. The only sub-term of the output term that the transducer needs to find is the one at position @2.2. Since discriminating tests prioritizes mnemonic of the assembly instructions over others, algorithm choose position @1 of the input for branching (at line 4). Specifically, it first selects all pairs who have $@1 = add\_2$. The set of pairs which match this test would be pairs 1–4. These four pairs again have common sub-structures, specifically *eax* at @12 and `plus(reg (eax & @12)` at @22. The algorithm emits this *mcp* (line 8). Pairs which do not satisfy $@1 = add\_2$ test (line 7) are considered for next iteration of **for** loop. In the next iteration, the algorithm would select the discriminating test of $@1 = mov\_2$. It should now be easy to understand how the algorithm constructs rest of the transducer. One thing though needs an explanation. Discriminating tests such as $<> *2$ or $= *2$ are arity-based discriminating tests which check for the arity of the input at that position. For instance, $@11 = *2$ test would check that the input term at position @11 has arity of 2. Similarly, tests of type $= StringType$ or $<> StringType$ are type-based discriminating test which check the type of input at that position. Notice that the input and output which satisfy these tests (e.g., $@11 : 10,\ 22 : (const\_int\ 10)$) are syntactically different than those which do not satisfy these tests (e.g.,

$11 : ebx,\ 22 : (reg\ ebx)$).

3.4.6 Error detection

In the final step, we flag any inconsistencies identified in the parameterized pairs. Two types of inconsistencies are possible. The first involves multiple distinct parameterized assembly instructions produce same IR. This is not a cause for concern, as there are typically multiple assembly instructions that have the same effect, such as `xor %eax, %eax` and `mov $0, %eax`. The second inconsistency involves mapping of same assembly instruction to distinct IRs. Unless the two IRs are semantically equivalent, this inconsistency likely indicates an error in the mapping. Our system flags such errors if they are found. In our experiments, we have not had to deal with this inconsistency.

3.5 Implementation

`LISC` prototype implementation works on Linux, and we have used it to synthesize instruction set semantics for x86, ARM and AVR[7] architectures. We chose these architectures because x86 CPUs are found in many desktops, laptops, notebooks, and servers, while ARM and AVR are used in embedded systems with ARM being popular especially in mobile phones and smartphones.

The breakdown of total implementation effort involved is shown in Figure 14. The implementation of log collection phase is architecture-neutral and is done using a GCC plugin that is is implemented in approximately 70 lines of C code. To collect IR to assembly mapping for `foo.c` in `log.dump`, one would use the command:

```
gcc -dP -fplugin=rule_collection.so -fplugin-arg-out-file=log.dump foo.c
```

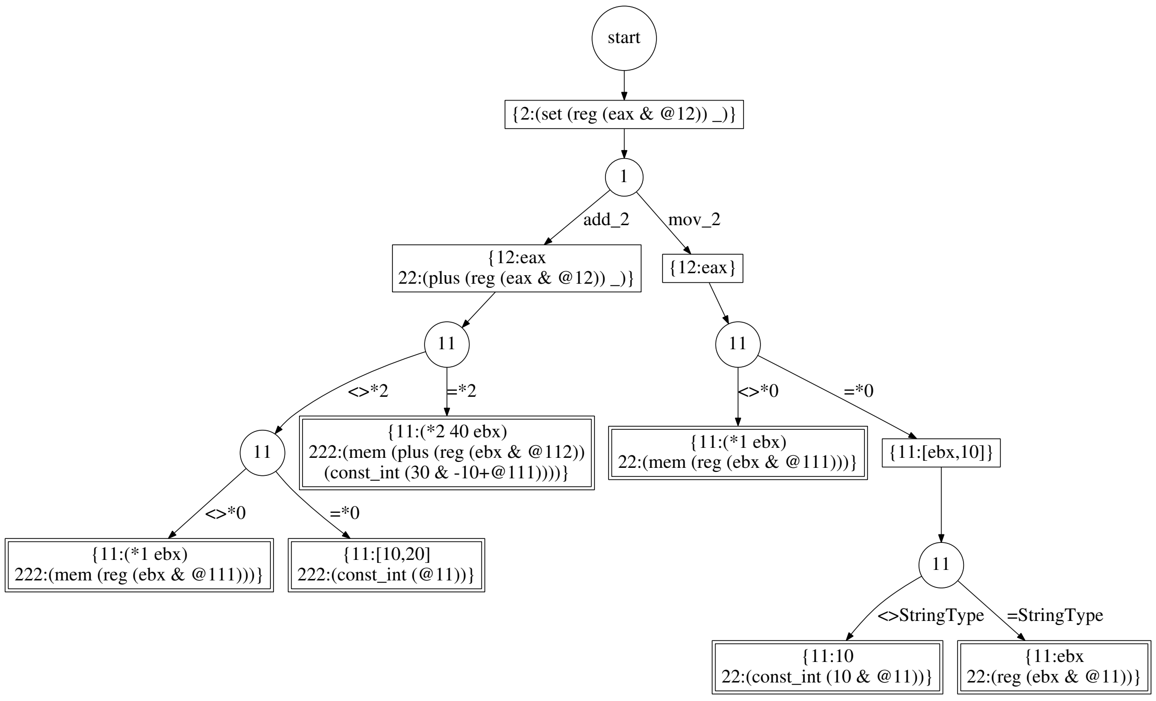
Here `−dP` is a standard GCC option to tell GCC to dump the IR corresponding to each assembly instruction as a comment. Thus, the log collection phase easily integrates with `configure` and `make` based package compilation process used commonly on Linux.

The implementation of parameterization and transducer construction is independent of GCC and is done using 2400 lines of OCaml code. All of the OCaml code is architecture-independent. But we do require minimal architecture-specific code to parse the log files

[7] AVR is a modified Harvard architecture 8-bit RISC single chip microcontroller which was developed by Atmel in 1996.

Pair 1	`(add X Y)`	`(set (reg (Eq Y)) (plus (reg (Eq Y)) (const_int (Eq X))))`
Pair 2	`(add X Y)`	`(set (reg (Eq Y)) (plus (reg (Eq Y)) (const_int (Eq X))))`
Pair 3	`(add (*1 X) Y)`	`(set (reg (Eq Y)) (plus (reg (Eq Y) (mem (reg (Eq X)))))`
Pair 4	`(add (*2 X Y) Z)`	`(set (reg (Eq Z)) (plus (reg (Eq Z)) (mem (plus (reg (Eq X)) (const_int (Minus Y 10))))))`
Pair 5	`(mov X Y)`	`(set (reg (Eq Y))(const_int (Eq X)))`
Pair 6	`(mov X Y)`	`(set (reg (Eq Y))(reg (Eq X)))`
Pair 7	`(mov (*1 X) Y)`	`(set (reg (Eq Y))(mem (reg (Eq X))))`

(a) **Example of parameterized pairs**


start
{2:(set (reg (eax & @12)) _)}
1
add_2
mov_2
{12:eax
22:(plus (reg (eax & @12)) _)}
{12:eax}
11
<>*2
=*2
11
{11:(*2 40 ebx)
222:(mem (plus (reg (ebx & @112))
(const_int (30 & -10+@111))))}
11
<>*0
=*0
{11:(*1 ebx)
22:(mem (reg (ebx & @111)))}
{11:[ebx,10]}
<>*0
=*0
{11:(*1 ebx)
222:(mem (reg (ebx & @111)))}
{11:[10,20]
222:(const_int (@11))}
11
<>StringType
=StringType
{11:10
22:(const_int (10 & @11))}
{11:ebx
22:(reg (ebx & @11))}

(b) **Tranducer constructed by** `LISC`

Figure 13: **Example of transducer construction by** `LISC`

Component	Architecture-neutral code	x86	ARM	AVR
Log collection (C)	70	-	-	-
Parameterization and transducer construction (OCaml)	2400	-	-	-
Assembly lexer (Ocamllex)	74	10	12	7
Assembly parser (OCamlyacc)	76	117	103	89
Utility code (scripts)	500	50	16	15

Figure 14: **Breakdown of LISC implementation effort**

and extract RTL and assembly pairs from the logs. Moreover, we have written architecture-specific parsers to parse assembly instructions and build first-order terms from them. Parsers for all three architectures in total constitute around 350 lines of OCamllex and OCamlyacc code written in OCaml. Once the RTL and assembly instructions are parsed and converted into first-order terms, rest of the processing is completely architecture-neutral.

One of the complications that arise when we decide to apply the extracted model for assembly-to-IR translation is that the assembly output produced by a compiler may be slightly different than the one produced by a disassembler. One such case is that of concrete memory addresses in the assembly instructions produced by a disassembler (e.g., `movl 0x80000000,%eax`). Such memory addresses originally would contain a label or a symbol reference in the assembly output of a compiler. As a result of this issue, the code generator logs will not be enough to translate such assembly instructions. To solve this issue, we simply treat concrete memory addresses as if they are label references. This is done by treating concrete memory addresses specially in architecture-specific parsers.

Rest of the utility code to disassemble binaries in the correct format, and apply the extracted models to the assembly instructions is written in `bash` scripts. These scripts are around 500 lines in total.

We added a number of features, such as the ability to save the generated transducer into a file, ability to load the transducer into memory from file, etc, into LISC. These features save effort to build a transducer repeatedly, and thus make it easier to use the generated transducer multiple times.

3.6 Evaluation

To evaluate effectiveness of `LISC`, we used it to extract semantics models for x86, ARM and AVR. We used multiple of packages to obtain compilation logs for model extraction. All experiments were performed on a quad-core Intel i7 processor running 32-bit Ubuntu-14.04 OS. Compilers and other tools needed for ARM and AVR were obtained using cross-compilers for these architectures[8].

3.6.1 Completeness of the model

A systematic evaluation of completeness requires knowledge of the target architecture, or else we cannot be sure whether all possible instructions have been used. While it is relatively easy to enumerate all possible opcodes, it is nontrivial to identify all possible operands, especially in the case of complex instruction sets. Moreover, we need to identify a collection of applications that use all of these instructions, which is another nontrivial task. For this reason, we have used an indirect approach that is commonly used for evaluating learning-based approaches: we use a set of programs (P_{train}) to generate model, and test them using another set (P_{test}) of programs. In particular, we determine what fraction of the instructions in the binaries for P_{test} can be lifted by the model learned from P_{train}. Specifically, for our experiments, P_{train} consists of multiple packages. We will talk about our package selection policy for P_{train} soon. P_{test}, on the other hand, consisted of all executable programs (not scripts) in the standard desktop OS. Specifically, we disassembled all binaries found on standard OS distributions (such as Ubuntu and Debian) using `objdump` and used the disassembled instructions to translate to RTL. Though such a P_{test} may not cover all possible target instructions and their operand combinations, we believe that it covers most commonly used instructions and their operand combinations.

As a comparison point, we implemented a naive *exact recall* (ER) approach, which would look for exact match of assembly instructions from P_{test} binaries in the pairs obtained from P_{train}. If a match is found, the corresponding RTL instruction in the pair is emitted as the translation of the input assembly instruction. Note that *Exact Recall* also provides a point for comparing correctness of the translations performed using training data, and the amount of generalization performed during such translations. Intuitively, translations performed using exact recall would be good but will have no generalization.

[8]On Ubuntu-14.04, one can simply install `binutils-arm-linux-gnueabi` package for ARM and `binutils-avr` and `gcc-avr` packages for AVR to obtain them.

Package selection policy. Package selection policy for P_{train} might seem to be less of an interesting issue, given that one can find tons of packages for many open-source OSes these days. Though such random selection of packages can serve the purpose, we devised an iterative package selection policy which uses the results of performed tests to make better choices for package selection than simply random selection. Specifically, package selection policy affects the completeness numbers — the ultimate goal of `LISC` evaluation is to demonstrate that all of the assembly instructions could be translated to RTL. In order to achieve 100% completeness number as quickly as possible, our package selection policy is to select a package that has lowest completeness number in the current iteration for compilation in the next iteration. Such selection policy *attempts* to improve coverage number — because of hand-written assembly instructions and presence of concrete memory addresses, lowest completeness number might not be a faithful indicator all the time. Nonetheless, we have found this policy to work as expected in our experiments.

P_{train}	**% of insns lifted**		LISC(%)		Missing mnemonics (absolute numbers)					
	ER	LISC	**Missing Mnemonics**	**Missing Operands**	**Basic**	**i387**	**SSE**	**AVX**	**Other**	**Total (%)**
`openssl-1.0.1f` `+binutils-2.22`	63.72	98.46	1.05	0.49	126	94	147	50	47	464 (39.09%)
+ `ffmpeg-2.3.3`	68.21	98.74	1.03	0.23	101	87	102	45	42	377 (31.76%)
+ `glibc-2.21`	68.74	98.80	1.01	0.19	92	70	100	43	41	346 (29.14%)
+ `ffmpeg-2.3.3`[a]	69.07	98.89	0.88	0.23	85	52	88	40	38	303 (25.52%)
+ `gstreamer-1.4.5`	71.07	99.10	0.79	0.11	66	37	57	25	36	221 (18.61%)
+ `qt-5.4.1`	72.45	99.21	0.69	0.09	47	29	35	16	34	161 (13.56%)
+ `linux kern-3.19`	73.97	99.49	0.44	0.07	17	5	1	2	24	49 (4.12%)
+ `Manual`	74.04	100.00	0.00	0.00	0	0	0	0	0	0 (0%)

(a) **Completeness results of all x86 binaries (GCC compiled) found in Ubuntu-14.04 distribution**

[a]This time we compiled `ffmpeg` with SSE, i387, AVX enabled one-by-one.

Type	**Number**
Executables	1796
Shared libraries	3260
Object files	87
Kernel modules	4089
Static archive	5
Total	**9237**

(b) **Breakdown of all Ubuntu-14.04 x86 binaries**

Type	**Number**
Missing mnemonics	49
Missing operand combinations	42
Total	**91**

(c) **Breakdown of manually-added rules**

	Total mnemonics in x86	**Mnemonics not supported by** `gcc-4.6.4`
Basic	395	17
i387	141	5
SSE	239	1
AVX	276	2
Others	136	24
Total	**1187**	**49**

(d) **Breakdown of all Ubuntu-14.04 x86 binaries**

Figure 15: **Detailed completeness evaluation results for x86**

Results for x86. The result for x86 is shown in Figure 15a. Rows in the figure list the packages that we added to the training data set in each round. So `+ffmpeg` in the second experiment means that `ffmpeg` was added to the base training data of `openssl` and `binutils` in second iteration. We used `gcc-4.6.4` for 32-bit x86 to compile these packages. For testing, we used all (9237) x86 binaries (Refer 15b for details.) found on Ubuntu-14.04 standard desktop OS installation. These binaries were disassembled using `objdump-v2.24` for x86. To the best of our knowledge, Ubuntu uses GCC (exact version not known) to produce these binaries. Columns in the figure list following in order: percentage of assembly instructions which could be translated to RTL by *Exact Recall*, percentage of assembly instructions which could be translated to RTL by `LISC`, the percentage breakdown of the missing instructions (calculated with respect to mnemonics from Intel manuals [5]) in terms of causes, and the classification of mnemonics of instructions which could not be lifted because those mnemonics were not in the training data.

After training `LISC` with a combination of `openssl` and `binutils` as a base, we could lift 98.46% instructions in all of x86 binaries on Ubuntu-14.04. Of the remaining 1.54% instructions, 1.05% instructions could not be lifted to RTL because their mnemonics were not present in P_{train}, while 0.49% of the instructions could not be lifted to RTL because their operand combinations were not in P_{train}. By missing operand combination, we mean if training data did not contain some instruction with a specific type of operand then we cannot translate such assembly instruction from test data. For instance, `push` with immediate data instruction (i.e., `push $imm`) was not found in training data, and hence could not be translated to IR. Of the 1.05% missing mnemonics, many of them belonged to the basic (126) and SSE (147) instruction set. In these instruction counts, we have counted a single mnemonic with different modes (byte vs word) separately. Thus, `movl` is a different mnemonic than `movb`. With our way of counting the mnemonics, there are a total of 1187 mnemonics in the Intel manual (details in 15d). The percentage of total missing mnemonics is obtained with reference to the 1187 total mnemonics in the x86. We found that Ubuntu-14.04 binaries cover all of the 1187 mnemonics. Therefore, the number of missing mnemonics obtained from P_{test} is same as the number of missing mnemonics obtained with reference to the Intel manual.

By associating missing instructions with the packages that they belonged to, we found that `ffmpeg` was the package with the highest percentage of missing instructions. That is why we selected `ffmpeg` package to add to P_{train} in the next round. Notice that with the addition of `ffmpeg` and the other packages in the subsequent rounds, the completeness number converges towards 100%. An interesting observation to make about the improvements in the completeness number is the changes in the breakdown of the missing operand combination percentages. Specifically, note that in the fourth experiment, the percentage of instructions with missing operand combinations went up from 0.19% to 0.23%. This is because as new packages are added, new mnemonics are discovered. These new mnemonics reduce the number of missing mnemonics, but they might cover only few operand combinations. So the percentage of missing mnemonics can only decrease when new packages are added, but the percentage of instructions with missing operand combinations can increase.

We followed this repetitive package addition till we reached a point where addition of new packages did not improve the completeness number any further. This was the point when the completeness number was 99.49% with 0.51% of instructions could not be lifted. Our analysis revealed that instructions which belonged to the missing mnemonics category were such that they were either hand-written or GCC did not support them. Instructions that are not generated by GCC are: instructions (such as `nop`, `enter`, etc) that are generated by the assembler, some arithmetic instructions (`aaa`, `aad`, `aam`, and `aas`), some instructions which set/clear bits of `EFLAGS` (`cld`, `cli`, `cmc`, `clac`), and low-level instructions (such as `cpuid`, `invpcid` which invalidates entries in TLB, `rdtsc`, etc)[9]. In all, we manually modeled 91 pairs of assembly instructions and their RTLs. Of these 91 pairs, 49 pairs modeled the semantics of instructions with missing mnemonic, and 42 pairs modeled the semantics of instructions with missing operand combinations. The breakdown of these 91 pairs is shown in Figure 15c. After the manual modeling, we were able to translate 100% of the assembly instructions from P_{test}. Modeling semantics of 49 mnemonics out of 1187 mnemonics manually underlines our intuition that compilers cover most of the target instructions, if not all.

Results for ARM. After performing the completeness experiment for x86, we repeated the same experiment for ARM. We followed the same package selection policy that we followed for x86 with the base P_{train} containing `openssl` and `binutils`. P_{test} contained all ARM assembly instructions obtained from disassembling all binaries (7.3K in total) on a Debian-7.8.0 desktop installation. We used the `gcc-4.7.3` cross-compiler for 32-bit ARM cortex-v7A [12] and `arm-linux-gnueabi-objdump-v2.24` as the ARM disassem-

[9]Note that the list of assembly instructions that are not supported by `gcc-4.6.4` is based on our analysis of GCC's source code and its x86 architecture-specifications. To the best of our knowledge, there is no official GCC documentation to confirm our findings. This is also applicable to other results such as those of ARM and LLVM presented next.

bler on x86 Linux. The results are shown in Figure 16. Most of the numbers can now be interpreted easily in a similar manner as the numbers for x86. The classification of instructions not supported by `gcc-4.7.3` is: few miscellaneous instructions (such as `dbg`, `dmb`, etc), few low-level instructions (such as `isb`, `mcr2`, `mrs`, `wfi`, etc), and few other advanced instructions (such as `vtbl`, `vtbx`, `vstm`, etc). The total time, including time to port the architecture-specific part of our implementation, was around 9.5 man hours.

P_{train}	% of insns lifted		`LISC`(%)		Missing mnemonics (absolute numbers)				
	ER	`LISC`	**Missing Mnemonics**	**Missing Operands**	**Basic**	**VFP**	**SIMD**	**Misc**	**Total (%)**
`Openssl` + `binutils`	34.58	88.21	6.86	4.94	673	48	57	36	814 (71.52%)
`+libfftw`	44.60	92.77	6.01	2.22	436	37	54	26	553 (48.59%)
`+swig`	51.43	95.67	3.72	0.61	321	32	47	22	422 (37.08%)
`+gcc`	62.54	96.26	1.73	2.01	275	28	43	20	366 (32.16%)
`+libc`	65.41	97.87	1.37	0.76	231	25	40	18	314 (27.59%)
`+gs`	67.69	98.92	1.02	0.04	203	22	37	14	276 (24.25%)
`+slapd`	69.54	98.95	0.85	0.2	183	19	31	13	246 (21.61%)
`+busybox`	71.45	99.61	0.3	0.09	141	17	26	12	196 (17.22%)
`+libpoppler.so`	71.90	99.66	0.3	0.04	85	13	19	9	126 (11.07%)
`+lib7z.so`	72.10	99.78	0.21	0.01	41	7	9	19	76 (6.67%)
`+manual`	72.23	100.00	0.00	0.00	0	0	0	0	0 (0%)

(a) **Completeness results of all ARM binaries found in Debian-7.8.0 distribution**

Type	**Number**
Executables	4349
Shared libraries	1844
Object files	631
Kernel modules	452
Static archive	50
Total	**7326**

(b) **Breakdown of all Debian-7.8.0 ARM binaries**

Type	**Number**
Missing mnemonics	76
Missing operand combinations	21
Total	**97**

(c) **Breakdown of manually-added rules**

	Total mnemonics in ARM	**Mnemonics not supported by `gcc-4.7.3`**
basic	951	41
VFP	66	7
SIMD	75	9
Misc	46	19
Total	**1138**	**76**

(d) **Breakdown of Debian-7.8.0 binaries**

Figure 16: **Detailed completeness evaluation results for ARM**

Results for AVR. We then selected the AVR [31] processor. We trained `LISC` by using same base packages (training data) as the earlier experiments and using `avr-gcc-v4.8.2` cross-compiler to compile them. We then tested the extracted model on few of the core-utils binaries (`ls`, `cp`, `cat`, `echo` and `head`) for AVR. We used `avr-objdump-v2.23.1` disassembler to disassemble these binaries. Unfortunately, we could not get complete `coreutils` package cross-compile using GCC's cross-compiler. The extracted model had around 700 abstract pairs for 4.3K concrete pairs from the compilation log, and it covered 72 out of the total 76 mnemonics. Our system was able to translate all of the assembly instructions from the input binaries. Our manual modeling effort was restricted to 4 missing mnemonics (`break`, `nop`, `wdr`, and `sleep`, which we found that are not supported by GCC) and 3 operand combinations. The total time taken including the time to port architecture-specific part of our implementation was hardly 3 man hours.

3.6.2 Soundness

Soundness of $A \rightarrow I$ translation is one of the fundamental properties on which the applications built using the assembly-to-IR translator would rely on. Thus, it is necessary to ensure the soundness of the translation. Given the number of generalizations performed by `LISC`, an interesting question is how many translations performed via generalization are incorrect. Intuitively, generalization increases the chances of producing incorrect translations. We employ different ways to ensure soundness — going from simple and straight-forward ones to the one with more formal proofs.

Cross-testing and self-testing. `LISC` generalizes $A \rightarrow I$ translations from $\langle I, A \rangle$ pairs found in the code generator logs. A simple soundness test is thus to check whether $A' \rightarrow I'$ translation holds for some $\langle I', A' \rangle$ pair. This is where cross-testing and self-testing come into picture.

If the model built using P_{train} is denoted by M, then these testing methods use the following definition of soundness:

$$\forall \langle I, A \rangle \in P_{test} : M(A) = I$$

In other words, we translate assembly A to new IR I' using the learned model M (this is represented by $M(A)$), and then check if new IR I' is same as original IR I. Conceptually, above definition checks following: given that a code generator has mapped IR I into assembly A (i.e., $I \rightarrow A$ holds), does $A \rightarrow I$ hold as per model M?

Cross-testing is one of the most basic method of testing the learned model (hypothesis) in machine learning. It consists of separate P_{train} and P_{test} set, and using the model learned from P_{train} to test it on P_{test}. Self-testing, on the other hand, uses P_{train} as P_{test}.

Semantic equivalence. Self-testing and Cross-testing are good techniques to evaluate the effect of generalization and to ensure that model learned by `LISC` does not over-fit or under-fit the training data. These techniques, unfortunately, suffer from an important limitation: if a single assembly instruction A maps to multiple IR instructions I_1 and I_2, then the model can map A to only either of them, and such a translation may not satisfy the above definition of soundness.

In order to address the limitation of self and cross-testing, we need to modify our definition of soundness. Specifically, compilers are supposed to perform semantic-preserving IR to assembly translations. Similarly if we ensure that assembly-to-IR translations are also semantic-preserving, then because of these two reasons, both $A \rightarrow I_1$ and $A \rightarrow I_2$ translations are sound.

Thus, a formal way to ensure the soundness of assembly-to-IR translation would be to ensure that every assembly instruction and its translated IR are *semantically-equivalent*. If semantics of an instruction is denoted by function *Sem*, then we can define soundness with semantic-equivalence as:

$$\forall \langle I, A \rangle \in P_{test} : Sem(M(A)) = Sem(I)$$

We will define the *Sem* function and how to perform semantic equivalence in the next section. But to make it easier to understand this section, we will give an intuitive explanation of semantic equivalence. Intuitively, IR I is considered semantically-equivalent to assembly instruction A, if semantics represented by I is *equal to* the semantics represented by A. Semantics of an instruction can roughly be thought of as the effect of the execution of that instruction on the CPU for that instruction. Note that since above definition demands equality of semantics, it is the strongest definition of semantic equivalence. In other words, if the above definition holds for some $A \rightarrow I$ translation, then such a translation is *sound*. We call this a *equivalence definition* of soundness.

Unfortunately, compilers do not model semantics of assembly instructions precisely in their intermediate representations. That is why often the above definition of semantic-equivalence would fail. To handle such situations, we modify the notion of semantic-equivalence as follows: IR I is considered semantically-equivalent to assembly instruction

A, if semantics represented by I is either *equal to* or the *approximation* of the semantics represented by A. An $A \rightarrow I$ translation is *sound* if above definition of semantic-equivalence applies for A and I. We call this a *abstraction definition* of soundness. As a comparison to the equivalence definition of soundness, notice that the abstraction definition is simply losing some precision in terms of semantics during translation. Otherwise, both definitions guarantee soundness.

Some applications built using an assembly-to-IR translator can still work with the abstraction definition of soundness[10] (e.g., static analysis). Other applications, however, demand a equivalence definition of soundness (e.g., IR instrumentation). To satisfy these demands, we first check if the equivalence definition holds for the $A \rightarrow I$ translation, and in case it does not, then check for the abstraction definition of soundness. It is perfectly fine to provide $A \rightarrow I$ which satisfy the equivalence definition of soundness to the applications which can work with the abstraction definition of soundness. But the opposite can lead to some issues. For instance, if the IR instrumentation system uses parts of IR semantics which are an over-approximation of the semantics of an assembly instruction, the results are undefined. In such cases, it is thus necessary that such over-approximate parts of IR semantics are not used by these applications.

Note that semantic equivalence gives stronger guarantees of soundness than any of the testing techniques mentioned earlier. This is because the testing techniques mentioned above rely on the assumption that $A \rightarrow I$ translation is *semantic preserving* — a compiler is supposed to preserve semantics during $I \rightarrow A$ translation. Semantic equivalence checking, on the other hand, does not make such assumptions. Instead, it checks whether compiler preserves semantics during translation. As a side note, because of this property of semantic equivalence checking, one can use it to test code generators. We demonstrate this in the next section.

Handling one-to-many assembly-to-IR translation. Note that the techniques mentioned above work in the context of one-to-one or many-to-one assembly-to-IR translation. But one-to-many assembly-to-IR translations presents an additional challenge for `LISC`. Specifically, the challenge comes from the confusion of which IR instruction to choose as a part of translation.

Intuitively, when multiple IRs map to single assembly, we want to choose an IR which

[10]These applications generally are conservative in deriving results, and this is how they can cope up with the abstraction definition of soundness.

is both *sound* and the *most precise* among those IRs. When a single assembly instruction maps to multiple IR instructions, then following are the possible cases:

- *All IRs are semantically-equivalent to the assembly instructions as per the equivalence definition of soundness, or some IRs are semantically-equivalent to the assembly instruction as per the equivalence definition, while others are semantically-equivalent as per the abstraction definition.* In such case, we can choose any IR out of those who satisfy the equivalence definition, and still guarantee soundness of $A \rightarrow I$ translation. Note that we can also choose an IR which satisfies the abstraction definition, but because we want to select an IR which is also more precise, we select the one which satisfies the equivalence definition.

- *All IRs are semantically-equivalent to the assembly instruction as per the abstraction definition of soundness.* In such a case, instead of selecting one randomly, we choose an IR which is the most precise among the possible IRs. Specifically, if assembly A maps to multiple IRs $I_1, ..., I_n$ represented as $A \rightarrow I_1$ & $A \rightarrow I_2$ & .. & $A \rightarrow I_n$, and all IRs are semantically-equivalent, then we choose I such that $s(I) = s(I_1) \cap s(I_2) \cap .. \cap s(I_n)$, where s is the semantics function that we define in the next section.

- *Some IRs do not satisfy the semantic-equivalence test.* Our abstraction definition of soundness is general enough that this case does not arise in practice. But when it does, it indicates that the compiler did not preserve semantics during translation. Thus it can indicate the presence of a bug in the compiler.

In our completeness experiment for x86, we found that of the total 1187 mnemonics in x86, around 24% map to multiple possible RTLs. So for the remaining 66%, self- and cross-testing are enough for a soundness check. The percentage of mnemonics with semantically-equivalent (as per the equivalence definition) RTLs were 22%. For remaining 2%, we simply cannot choose any RTL randomly. Instead, we have to choose an IR by defining $\cap$ operation on two IRs. One of the very common cases where we use the $\cap$ operation is when one RTL says (`clobber EFLAGS`) and another one precisely says which bits of `EFLAGS` are modified. The $\cap$ operation in such cases selects the RTL which says which bits of `EFLAGS` are modified. After defining this operation for 2% of the mnemonics, 1.8% of the mnemonics passed semantic-equivalence check (the abstraction definition). 0.2% (or 2 mnemonics in absolute numbers) failed even on the abstraction definition. Our analysis revealed that IR for such assembly instructions capture the semantics which is not modeled explicitly in assembly. Call and indirect jump that call a variable-argument function represent these two mnemonics. In such a case, the assembly instruction looks like any other call instruction, such as, `call func`. But the RTL contains explicit reference to the number of parameters passed in the call, for example, (`call` (`symbol_ref func`)(`const_int 10`)). The $\cap$ operation of multiple (`const_int`) is `nil`. That is why the soundness check fails for such pairs.

Similarly for ARM, of around 1200 assembly mnemonics, 180 had multiple possible RTLs. Of these 180, 171 (95%) had semantically-equivalent RTLs. 5% of the mnemonics were mapping to RTLs which were semantically-inequivalent (as per the equivalence definition). After using the $\cap$ operation on multiple RTLs, we were able to check the equivalence of 7 mnemonics. The remaining 2 mnemonics (`bl` and `blx`) follow the same story of call and jump from the x86.

Although we cannot check the semantic-equivalence in case of indirect call, the applications utilizing the translated IRs can mitigate the concern by relying on analysis to uncover operands of call. Recovering the number of operands of a call from the binary is an investigated research problem [26, 36, 37].

3.6.3 Compiler independence

To find out how many of the instructions produced by compilers other than GCC can be lifted by `LISC`, we used the LLVM compiler (`clang-3.3`) to build `coreutils-2.23` and SPEC 2000 binaries. We lifted them to IR using the GCC x86 model that achieved 100% completeness on Ubuntu-14.04 binaries. Results of this evaluation are summarized in Figure 17. Our system was able to lift an average of around 96.03% of the instructions. We found that 3.97% instructions that we could not lift were using 7 mnemonics (such as `nopw`) and 12 operand combinations (such as `nopl 0(%eax)` produced by LLVM vs only `nopl` produced by GCC) that were not covered in the training data[11]. This finding is also reflected in the result of *exact recall*, which is around 57% for LLVM produced binaries, while it is around 74% for GCC produced binaries.

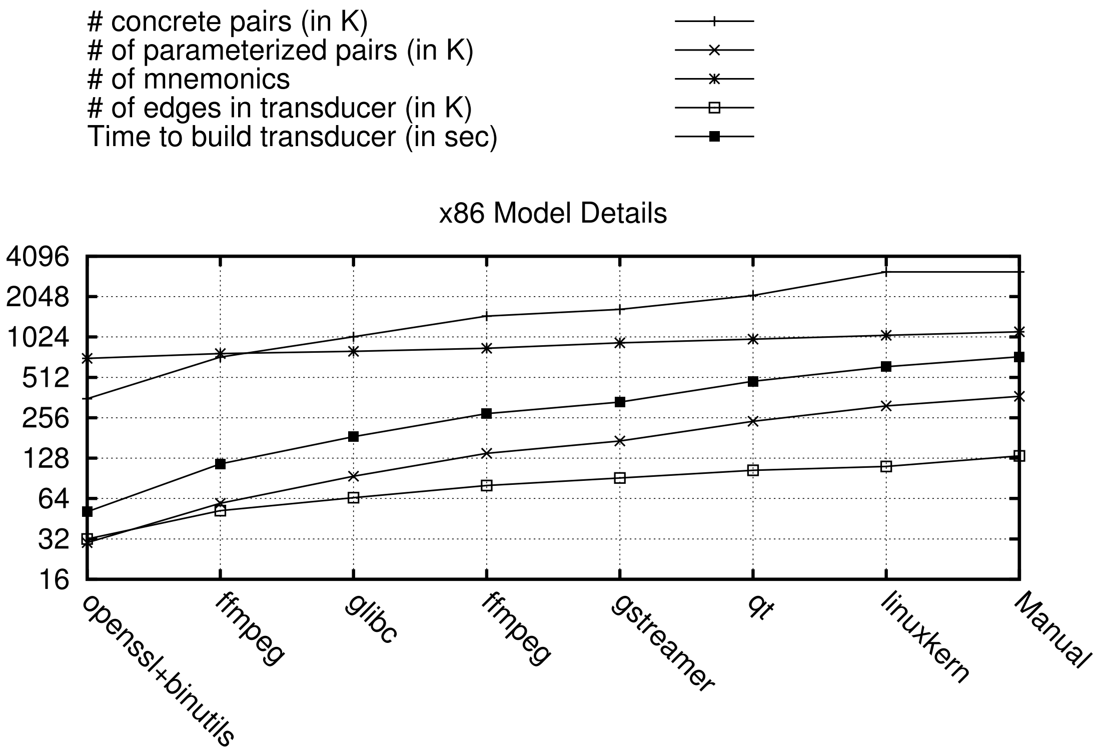
Binary	# of insns	% of insns lifted	
		ER	LISC
chown	9939	59.01	97.79
chgrp	9630	59.45	95.17
chmod	9018	58.94	95.73
cp	18043	57.41	97.16
cat	6676	57.94	97.35
cut	4986	55.45	95.76
dir	20489	57.54	97.65
echo	3800	56.10	95.16
head	6919	63.94	96.76
ln	8121	58.35	95.46
ls	20489	57.37	97.74
mkdir	8021	55.65	97.45
mv	18375	57.87	97.08
pwd	4136	58.44	95.16
rm	9348	59.76	95.42
sort	18510	59.54	94.56
tail	11868	58.55	95.04
uname	3733	56.77	94.34
ammp	26444	51.36	94.01
art	3262	47.76	92.96
bzip2	7832	56.15	95.65
crafty	36028	57.76	97.88
equake	4233	59.53	96.54
gap	100598	57.47	96.88
gzip	7916	59.65	96.51
mcf	2334	48.53	96.27
mesa	99805	51.47	95.67
parser	25260	56.56	95.53
vpr	25310	59.78	96.06
Total	521184	**(Avg)** 57.03	**(Avg)** 96.03

Figure 17: **Completeness results of LLVM compiled binaries for x86**

3.6.4 Sizes of the models and performance of LISC

Figure 18 captures the effect of adding different packages to P_{train} on the number of concrete and parameterized pairs, model size, and the number of mnemonics. We used four different metrics for comparison: the number of concrete pairs in the log files, the number of parameterized pairs derived by LISC from the concrete pairs, the number of mnemonics

[11]We found that LLVM generated x86 binaries use multi-byte nop sequences, while those generated by GCC contain a sequence of single-byte nop to represent multi-byte nop.



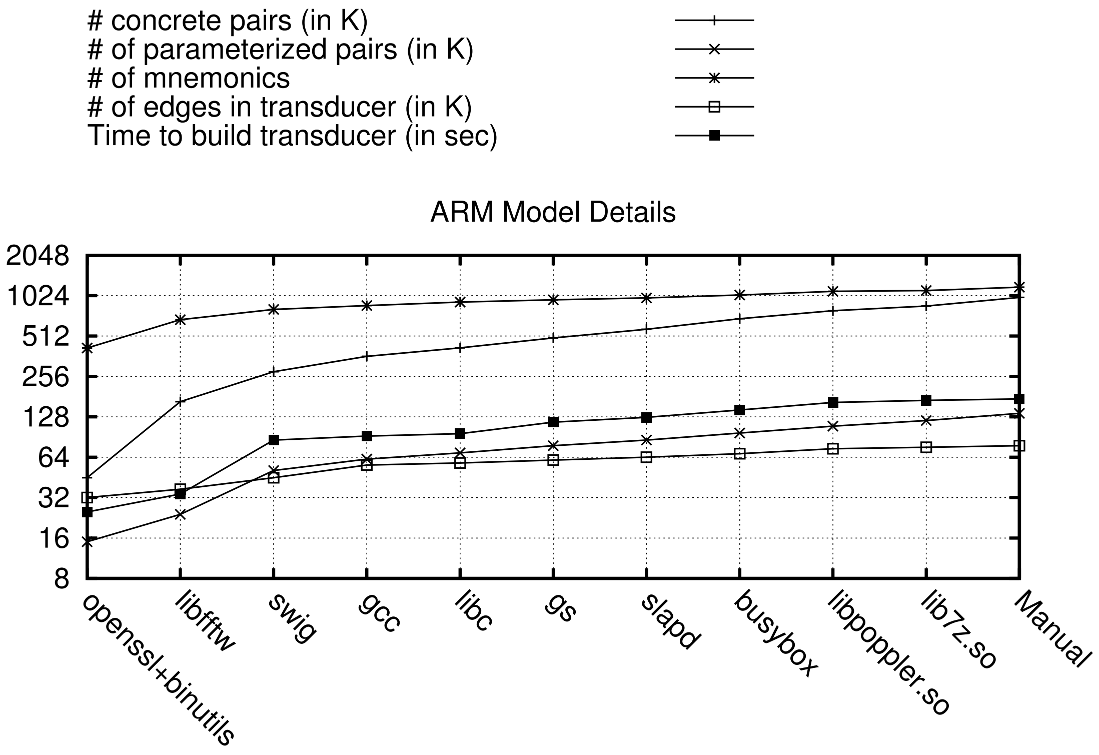


Figure 18: **Extracted model details**

covered by the derived models, the size of the transducer in terms of number of edges, and the time taken by LISC to build the transducer. Interesting things to note about the figure is that LISC obtains between 3X–10X reduction in the number of pairs by merging pairs together. On the other hand, the time taken by LISC is linear to the number of parameterized pairs. The increase in the number of concrete pairs indicate that many new operand combinations were covered. These combinations also resulted in the increase in the number of parameterized pairs when the logs obtained from new packages are added to the existing list.

3.7 Related Work

Although there does not exist an approach to extract instruction-set semantics from code generator logs automatically, LISC relates to a number of well-known topics. Because it is a learning-based approach that tries to extract the mapping functions between IR and assembly, it relates to machine learning [62, 64] in general. The transducer constructed by LISC is very close to the concept of decision trees [62] from data mining [76]. In the natural language processing (NLP) domain, the transducer that translates input to output is called as Finite State Transducer (FST) [63]. FSTs are very popular in the NLP domain and LISC's objective of extracting the mapping function can be thought of as an attempt to build FST automatically. We explain each of these related areas in detail.

3.7.1 Relation to machine learning

Learning hypothesis. Although we say that LISC learns the translation of RTL instructions into corresponding assembly instructions, in terms of the notion of learning used in the standard machine learning (ML) domain, we are not learning the translation. Specifically, note that LISC employs learning only when it needs to understand how the leaves (operand values) in RTL and corresponding assembly instruction map to each other. This mapping is actually what is called as *hypothesis* in the ML domain. Because the operand values are continuous, our learning problem can be thought of as a *regression* problem. In particular, given as input operand i and output operand o, we want to find f such that $f(i) = o$. Relying on the observation that operand values between RTL and assembly mostly undergo a linear relation, the problem of finding f can be classified as *linear regression* problem. One can then employ *Gradient Descent* algorithm to find f. In relation to these standard practices in the ML domain, LISC does not even employ Gradient Descent. Instead, by relying on the observation that operand values in RTL follow an identity function (e.g., $f(x) = x$), we have simply enumerated five different mapping functions (one identity function and four arithmetic functions for addition, subtraction, multiplication, and division), and `LISC` searches for the hypothesis function among these five functions. In other words, the problem addressed by `LISC` is such that it does not really need to employ standard ML practices for solving the problem.

Evaluating hypothesis. Although we do not use standard machine learning algorithms for solving our problem, we do employ standard ML evaluation criteria such as self-testing and cross-testing. A standard practice in the ML domain to evaluate the quality of the learned hypothesis is to split the available data into three sets: training, test, and cross-validation. The training set is used to learn the hypothesis; the test set is used to evaluate the hypothesis; the cross-validation set is used to learn parameters in the model selection process (such as degree of polynomial for hypothesis, regularization parameter, etc). Because model selection is not a problem for `LISC`, we do not use separate validation set for our evaluation.

One of the simple ways to evaluate a hypothesis function is called *self-testing*. In self-testing, both the training set (used to derive the hypothesis) and the testing set (used to test the hypothesis) are the same. Self-testing helps one evaluate how good the hypothesis "fits" the training data. If a hypothesis fits the training data poorly (also called *under-fitting*), then one might need to change the model selection parameters to derive new hypothesis which might fit better. Self-testing is thus a good way to detect under-fitting. On the other hand, self-testing has the limitation that it cannot detect *over-fitting* — over-fitting occurs when the hypothesis fits the training data too strictly such that it does not fit "unseen" data (data not seen in training data). For instance, if a 2nd degree polynomial is enough to fit the training data, but the hypothesis is a 4th degree polynomial which fits the training data well, then over-fitting might occur. A standard machine learning practice to detect over-fitting is to evaluate hypothesis against unseen data. This is called *cross-testing*. Specifically, the hypothesis function derived from a training set is tested against a test set which is separate from the training set. This is also called *generalization* in the ML domain.

Self-testing helps us evaluate training error, while cross-testing helps us evaluate testing (or generalization) error. In ML problems, it is very common for one or both of these errors to be high (when both are high, it is called *high bias*. On the other hand, when the training error is low, but the testing error is high, it is called *high variance*.) In the context of our problem, note that we require both training and testing error to be zero! This is because the extracted semantic models will be employed in the context of applications such as binary analysis, instrumentation and translation problems, where the assembly-to-IR translator needs to guarantee "correct" translations. In other words, in the context of errors in learning, our problem is different than the one faced by standard ML problems. Fortunately, the hypothesis function learned by `LISC` is simple enough that errors do not creep in. This is also demonstrated by the evaluation of `LISC`.

Availability of learning data. It is not unusual that the standard ML problems need to work with limited training data. Limited training data can lead to the training/testing errors discussed above. Fortunately, `LISC` does not face a limited data problem. We can easily find new packages to compile and generate new dataset for `LISC`. The question of how many packages to add to the training data is addressed by evaluating against a test data. To be precise, in the evaluation we discussed how many packages to be added to training data to lift all binaries on Ubuntu-14.04 distribution. Note that, as is the case with most learning systems, it might happen that we can find new test data for which we are not able to translate 100% of the instructions. In such cases, we can simply add more packages to the training data. In other words, there will not be training data which is enough to handle any random test data. In most machine learning problems, adding new training data helps improve the accuracy of learning outcome. For `LISC`, this is also true.

3.7.2 Decision tree learning

Decision tree learning [62, 2] is a method used in data mining and machine learning for inferring high-level rules from the gathered data (or training data). The gathered data needs to be of the form of collection of records where individual entries in the record are attribute-value pairs. For instance, the data below

$$[\textbf{Outlook} = \textit{Sunny},\ \textbf{Temperature} = \textit{High},\ \textbf{Playtennis} = \textit{Yes}]$$
$$[\textbf{Outlook} = \textit{Sunny},\ \textbf{Temperature} = \textit{Low},\ \textbf{Playtennis} = \textit{No}]$$
$$[\textbf{Outlook} = \textit{Overcast},\ \textbf{Temperature} = \textit{High},\ \textbf{Playtennis} = \textit{No}]$$
$$[\textbf{Outlook} = \textit{Overcast},\ \textbf{Temperature} = \textit{Low},\ \textbf{Playtennis} = \textit{No}]$$

is a set of records about weather conditions and the decision to play tennis. Here **Outlook** is an attribute which takes the values from the set $\{\textit{Sunny}, \textit{Overcast}\}$. Similarly,

Temperature and **Playtennis** are other attributes. If we treat **Playtennis** as an output attribute (that we want to infer from input attributes **Outlook** and **Temperature**) then the data can be seen as the following rule:

$$\textbf{Outlook} = \textit{Overcast} \rightarrow \textbf{Playtennis} = \textit{No} \wedge$$
$$\textbf{Outlook} = \textit{Sunny} \wedge \textbf{Temperature} = \textit{Low} \rightarrow \textbf{Playtennis} = \textit{No} \wedge$$
$$\textbf{Outlook} = \textit{Sunny} \wedge \textbf{Temperature} = \textit{High} \rightarrow \textbf{Playtennis} = \textit{Yes}$$

A decision tree is a tree that semantically represents rules that can infer output attribute from the values of the input attributes. Interior nodes in such a tree are input attributes from the data and the edges represent the possible values of these attributes. Finally, the leaves of a decision tree represent the value of an output attribute that can be inferred from the conditions met along the path from the leaf to the root. A decision tree for the above data set is shown in Figure 19. Notice that every clause in the above rule is represented by a single path from the root to a leaf in the tree, and the complete tree represents the complete rule.

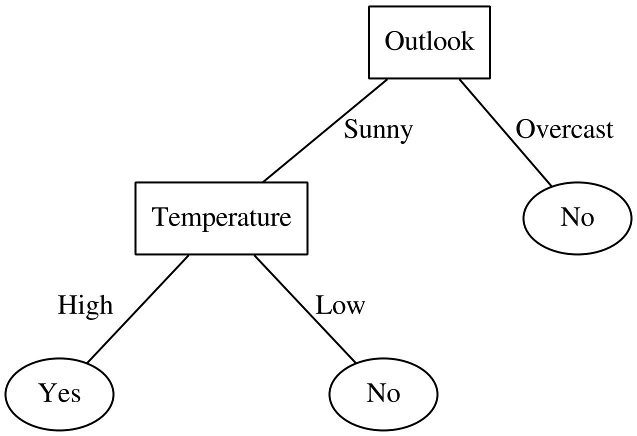


Figure 19: **Decision tree for decision to play tennis based on weather data**

More generally, rules in decision trees take several attributes (features in machine learning) of the data as input and *classify* the value of the output attribute based on the input attributes. Decision trees can thus also be thought of as a way to approach *classification* problem in machine learning (Decision tree learning can be thought of as a method to approximate a discrete-valued function, in which the hypothesis is represented as a decision tree.) Generally, decision tree learning problems work with data sets where both input as well as output attributes have discrete values. Nonetheless, extensions have been proposed where decision trees can be learned from continuous-valued input attributes as well.

Inferring such rules from data can be useful in understanding the data. Additionally, it allows us to understand whether one is likely to play tennis or not given the weather conditions not in the data set. The overall idea of learning rules from the data set is also called *association rule learning*.

This section is not intended to be a comprehensive discussion on Decision Trees (For a detailed discussion on Decision Trees, see [62].) It is meant instead to be a comparison between decision tree construction algorithms, their properties, etc, and the algorithm used by `LISC`.

At a high level, `LISC` also builds a decision tree to map an input RTL instructions and their operands to an output assembly instructions and their operands. Attributes in such a tree are actually the *position variables* in the term for an input RTL instruction. Nonetheless, there exists a number of differences between `LISC` decision trees and the decision trees discussed in the literature. We discuss these differences below.

Decision tree construction. One of the earliest algorithm for decision tree construction is the ID3 algorithm [73] proposed by Quinlan[12]. Throughout the algorithm, the decision tree is constructed by selecting an input attribute to branch on and partitioning data based on different values of the selected attribute. The question addressed by ID3 in attribute selection for partitioning is: *which attribute to select for partitioning?*

ID3 defines a heuristic called *information gain* to select an attribute for partitioning. Specifically, an attribute with the highest information gain is selected for partitioning the data set. Conceptually, information gain of an attribute specifies the reduction in the *entropy* of the data set after partitioning the data set based on the values of the selected attribute (The notion of entropy, which was proposed by Shannon, is taken from the domain of information theory.) The rationale behind selecting information gain as a measure to decide the attribute was to prefer shorter trees over longer ones. Shorter trees reduce the matching time when the test data is matched against the decision tree. Shorter trees are thus more efficient in matching than the longer trees. Notice that in the tree from Figure 19, we did not branch on **Temperature** if the **Outlook** is *Overcast*. This was done because once we know that the **Outlook** is *Overcast*, the entropy of the data set is 0 (This is because, as per the training data, the value of the output attribute is *No* in all cases.)

[12]Quinlan later proposed C4.5 [74] as an extension of ID3.

At a high-level, ID3 employs a greedy strategy (based on information gain heuristic) to build shorter decision trees than longer ones. Unfortunately, it means that it makes a locally-optimal decision than a globally-optimal one. Consequently, trees built by ID3 can be sub-optimal.

The `LISC` decision tree construction algorithm is more general than ID3 algorithm. First, ID3 operates on data sets where attributes are discrete-valued (Later, extension was proposed to fit continuous-valued attributes into ID3 by converting a continuous-valued attribute into discrete attribute [74].) Our algorithm, on the other hand, is more general than a classification problem — `LISC` attempts to learn a function: given an input, produce the corresponding output. The question of test selection has overlap with decision trees, but other aspects such as parameters, transformation functions on these parameters, etc do not have an analog there. Second, to the best of our knowledge, no decision tree construction algorithm operates on structured input. Lastly, unlike decision trees, `LISC` constructs finite state automaton by sharing nodes of decision trees. This leads to reducing the size of the automaton as compared to the equivalent decision tree.

Branching criteria: information gain vs discriminating tests. ID3 uses information gain to decide which attribute to use for branching. `LISC`, on the other hand, uses discriminating tests that preserve structural integrity of the output. The difference in the branching policies arises because of the difference in the objectives and the contexts of the two algorithms. `LISC` is used in the context where only certain types of parameter values can appear in the output terms. This is what we call *structural integrity*. To ensure structural integrity, `LISC` branches based on values of non-parameter positions. Once all non-parameter positions are explored, it then branches based on parameter positions having the same types of values.

`LISC` cannot use ID3's information gain policy as a branching factor. There are multiple reasons why this is the case. First, none of the existing decision tree construction algorithms such as ID3 and C4.5 operate on data sets where values are terms or variables. `LISC` operates in the context where this is the case. Second, the syntactic integrity check can be seen as dictating the selection criteria for branching. If ID3's choices conflict with those required for the syntactic integrity check, then that could produce ill-formed terms. Third, it has been shown that the information gain heuristic does not work well when building decision trees for terms [89, 81]. Specifically, previous work that used traditional decision trees have experienced massive increase in size in many cases while operating on terms

[81].

Over-fitting and inductive bias. Over-fitting, a well-known problem in machine learning, arises when the learned hypothesis does not generalize well to future test data. The problem arises when the learned hypothesis fits the training data more closely than it should. Inductive bias in machine learning is the set of assumptions which, along with the training data, justify the policy used by the learner to classify test data. In other words, inductive bias specifies how a particular algorithm generalizes so that it is applicable to future data (in other words, it avoids over-fitting.) In all decision tree building algorithms, some form of inductive bias is required to avoid over-fitting.

ID3's inductive bias can be specified as: *shorter trees are preferred over longer trees*[13]*, and trees that place high information gain attributes closer to the root are preferred over those that do not* [62]. Illustration of this policy can be seen in the tree that we discussed earlier. Specifically, the tree built outputs **Playtennis** $=$ *No* immediately when **Outlook** $=$ *Overcast*. It does not explore any other trees with **Outlook** $=$ *Overcast*. By avoiding such exploration, ID3 makes the decision tree generalize to future data. For instance, the particular tree from the example generalizes to **Temperature** $=$ *Normal* when **Outlook** $=$ *Overcast*.

LISC inductive bias, on the other hand, can be specified as: *trees that preserve the syntactic integrity of the output are preferred over those that do not. Among the trees that preserve syntactic integrity, trees that place non-parameter attributes (*non-parameter positions*) closer to the root are preferred over those that do not*. There is a very clear difference between the policies of LISC and ID3. This difference stems from the fact that LISC attempts to address the problem of translating input instruction into output instruction (by preserving syntactic integrity). Building shorter trees is thus not the branching factor for LISC. Because LISC needs to preserve syntactic integrity, it cannot avoid over-fitting when it is branching based on non-parameter values. But when it outputs the translation, it avoids over-fitting by using interval constraints (or variables) instead of using set constraints. For instance, let us say the training data contains a rule that applies for an immediate between $\{1,3\}$. Then, LISC will generalize this rule to interval $[1,3]$ and thus will make it applicable for immediate 2 also. LISC can also simply consider this rule to be applicable for any immediate value between $[-\infty,\infty]$. LISC can do this by relying on the observation that, usu-

[13]Interestingly, the soundness of ID3's favor for shorter trees over longer ones is an unsolved debate. It goes back to 1320, when William of Occam first discussed this question. It is known as *Occam's razor* in machine learning.

ally, the mapping function (hypothesis) used by immediate values is the identity function. The effectiveness of `LISC` policy in terms of over-fitting is experimentally demonstrated.

3.7.3 Finite state transducers and grammatical inference

Finite state transducers (FST) [63] has been one of the fundamental concepts in computer science. Conceptually, an FST is an extension of an FSA (finite-state automata) which produces output strings in addition to accepting input. Formally, a finite state transducer is a finite state machine with two tapes: an input tape and an output tape. This contrasts with an FSA, which has a single tape. FST has been used in a number of areas such as natural language processing, speech processing, image processing, and machine translation. The initial definition of FST operates on strings, but later on, extensions were proposed to FST that accept and produce trees. Transducers which accept trees are called as tree transducers [77, 86, 30]. Tree transducers are more general than FSTs because they allow input and output to be structured. Consequently, tree transducers have found a number of important applications which need to process structured data (e.g., syntax-directed translation, tree rewriting, etc).

`LISC` can also be considered as an application of tree transducers to the problem of model extraction from the code generator logs. Specifically, assembly and IR instructions in the code generator logs are structured, and one can find mappings between parts of inputs and outputs. Furthermore, we found that an algorithm called OSTIA (Onward Subsequential Transducer Inference Algorithm) [68, 69] has already been proposed in the literature for learning string transducers from a training set. OSTIA builds a prefix-tree representation (idea same as *mcp*) of all input/output pairs from training data. Additionally, it has the ability to move output strings towards root of the tree as much as possible (our notion of eagerly emitting output is same as this idea), which it calls as *onward* tree representation.

Although OSTIA can be considered as a fundamental algorithm from which our algorithm is derived, we found that there are key differences between `LISC` and OSTIA. First of all, to the best of our knowledge, application of tree transducers to learn assembly-to-IR translations from code generator logs is a novel idea. Second, the classical definition of FST relies on labelling the automata edges with input symbols. On the other hand, `LISC` relies on a number of different types of tests as discriminating tests. Lastly, OSTIA builds upon the notion of *sequential* transducers, where the order of input and output prefixes is preserved. Conceptually, *sequential* transducers visit an input in sequence, and produce an output in the same order in which input is visited. Such transducers thus inherently restrict which parts of an input are used for deciding the branching conditions in case of discriminating tests. While there is a notion of *subsequential* transducers to eliminate this restriction, we believe that LISC can be thought of as a generalized *subsequential* transducer which considers multiple criteria for branching.

Some of the techniques used in solving grammatical inference problem [51, 78] have interesting correlations with the techniques developed in LISC. Grammatical inference problem is the problem of learning a context-free grammar for the unknown language from the finite set of positive (and possibly negative) examples in the language. A typical approach used in solving this problem is to first build a prefix-tree acceptor which exactly accepts a set of given positive examples, and then to successively merge the states in the tree to produce an automata that accepts more general language than the one accepted by the tree. The problem of learning the grammar is often defined in terms of the problem of finding which states to merge. One of the common approaches used in state merging is to rely on negative examples if they are available. Specifically, if the merging of some states leads to acceptance of any negative example by the automata then such a merging is rejected. While the overall idea of building a prefix-tree and then merging its states can be thought of as similar to employed by LISC, the approach used by LISC is more general. Specifically, LISC builds a tree transducer which is much more general than the tree automata required in the grammatical inference problem. Moreover, prefix-tree acceptor naturally restricts the order of inspected symbols from the input to the sequential order. LISC, on the other hand, does not restrict itself to the sequential order but rather uses non-sequential order. The non-sequential order allows LISC to produce smaller automata (and to emit output much faster) than those produced by using sequential order.

3.7.4 Learning using assemblers and compilers

Although research work [28, 44] do not extract instruction-set semantic models using code generators, their overall approaches have many interesting correlations with LISC.

Derive [44] is an approach to learn machine encodings of assembly instructions by using an assembler as a black-box. Specifically, instead of writing the assembly to machine instruction encodings manually, Derive relies on the observation that such encodings are already specified in the system assembler, and that one can learn them from the assembler by feeding different operand permutations of assembly instructions and analyzing the generated machine instruction bytes. In comparison to LISC, Derive can be thought of as an approach to extract assembly-to-machine instruction encoding models which can be later used in other tools like disassemblers, debuggers, binary rewriters, etc. Instead of obtaining the training date as in LISC, Derive demands the user to supply it with a list of assembly instruction names and operand types. Using such a description, it then generates all possible permutations of register names, few combinations of immediate and other types of operands. Training data used by LISC can thus be thought of as similar to the permutations produced by Derive. In order to learn an encoding of a particular operand, Derive relies on a simple idea of keeping all other operands constant and generating all (or few in case of immediates) permutations of the operand of interest. In terms of learning the mapping functions between input and output operands, it follows an approach similar to LISC: the encoding of a particular operand does not depend on other operands, and they mostly follow an identity function (with immediates undergoing few additional simple transformations such as multiply or divide by a constant). Given that Derive attempts to extract the model of an assembler, it needs to deal with issues such as operand size, endianness, etc, which LISC does not need to deal with. Although, Derive has some interesting correlations, it has one important difference with LISC. Specifically, LISC builds a tree transducer to represent the extracted model, while Derive simply generates a list of C code snippets which implement the encoding for different assembly instructions.

While Derive learns the assembly-to-machine instruction encoding models, work by Collberg [28] attempts to extract target architecture specifications used by the code generators automatically. Collberg's hypothesis is that by using a native C code compiler to compile snippets of C code and by observing the assembly and machine code outputs, one can learn enough of architecture details to construct architecture specifications for the code generators. Such an approach can avoid manual efforts needed in modeling architecture specifications for new architectures and thus enable easy porting of compiler backends. In order to extract the instruction semantics from a set of assembly instructions, Collberg's high-level idea is to use "reverse interpretation". In particular, "reverse interpretation" tries to synthesize instruction semantics by using a set of basic primitives such as operations for arithmetic, logical, shift, load and store. By using very simple basic C programs (such as for $a = b + c$), reverse interpretation can synthesize how to generate assembly code for any other addition operation automatically. The set of primitives used to extract the instruction semantics can be thought of as the mapping function in LISC. Since Collberg needs to extract the instruction semantics, primitives for load and store needs to be provided. Besides these correlations, most other details are different from LISC.

3.8 Summary

In this section, we described a learning-based approach (named `LISC`) to extract instruction-set semantics models by treating a compiler as a black-box. Being a learning-based approach which generalizes from the available training data, `LISC` needs to evaluate the correctness of the extracted models. We discussed how `LISC` makes conservative generalizations to reduce the chances of incorrect learning outcome, and evaluated the accuracy of the extracted models. Finally, we demonstrated that `LISC` can learn the semantic models of new architectures (such as AVR) in a very short amount of time (3 hours), and can considerably reduce the amount of manual effort needed in supporting new architectures. Nonetheless, being a learning-based approach, `LISC` theoretically suffers from the limitations of applicability of extracted models to unseen test data. To address this limitation and to theoretically ensure the completeness of the model, in the next section, we look at a complimentary technique to extract semantic models.

4 EISSEC: EXTRACTING INSTRUCTION SEMANTICS BY SYMBOLIC EXECUTION OF CODE GENERATORS

The key limitation of the LISC approach is that as a learning-based approach, it relies on the completeness of the training data to achieve completeness of extracted semantic models. There could be three limitations of the models produced by LISC:

1. *All target instructions are not covered in the extracted models.*

   This case may arise because the training data used to extract the model does not contain some target instructions. For instance, this may happen because a compiler may not generate some assembly instruction *A* just because there is an alternative more efficient target instruction which is semantically equivalent to *A*.

   It may also be possible that *A* is actually generated by a compiler, but our test inputs did not cover that part of the compiler which generates *A*.

2. *All possible operand combinations of the target instructions are not covered.*

   This case may arise because of some choices that a compiler may make. For instance, even though instruction *A* allows an immediate value as an operand, a compiler may decide to store such immediate value into a register and generate instruction *A* with that register as an operand. Consequently, binaries produced from such a compiler will not contain *A* with an immediate operand.

3. *All possible operand values are not covered.*

   This case arises because there are too many possible values for some operands making it almost infeasible to cover all. For instance, it is infeasible to produce 2^{32} possible instances (on a 32-bit processor) of an assembly instruction containing an immediate value. Moreover, the number of possible instances increases drastically when the instruction contains multiple immediate values.

In terms of the implications of these limitations, the first two limitations lead to incomplete semantic models. Although the LISC completeness evaluation demonstrated that its extracted models can handle assembly instructions from all the binaries on desktop OSes, the completeness number of a learning-based approach is always with respect to a test data set. In other words, it may be possible that binaries present on typical desktop OSes do not cover all of the target architecture instructions and their possible operand combinations.

The last limitation, on the other hand, can compromise the correctness of the extracted models. Specifically, incomplete training data requires LISC to generalize the extracted models to support mapping pairs that are missing from the data. Such generalizations may introduce correctness issues in the extracted models.

A better approach to evaluate the correctness and completeness of LISC extracted models is to generate test data set such that it contains all possible target instructions and their operand combinations. This is the goal of our approach called EISSEC (stands for Extracting Instruction Semantics by Symbolic Execution of Code generators, and pronounced as Isaac).

To put simply, EISSEC is a technique to find out all target instructions and their operand combinations that are supported by a code generator. In addition, it also precisely yet generically captures how various operand values of instructions are treated by the target architectures. By generically, we mean EISSEC will not lead to the problem of generating 2^{32} instances of an immediate operand. Instead, it generalizes the effect of 2^{32} instances. It thus helps in addressing the last limitation of LISC.

In terms of the first two limitations, EISSEC extracts all instructions supported by a code generator and, as we saw in LISC, it is possible that a code generator does not support some target instructions. Thus, the list of assembly instructions and their operands obtained by EISSEC may be incomplete. The key advantage of EISSEC is that it can obtain all assembly instructions and their operand combinations which are supported by a code generator, but were not present in the training and test data of LISC because the parts of the code generator which produce them were not covered by LISC. *In other words,* EISSEC *enables us to achieve complete coverage of a code generator.*

Because our goal is to cover all paths inside a code generator and produce its input (IR) to output (assembly) mapping, the problem that we are trying to solve can be classified as *code generator function extraction*. More formally, the problem of function extraction can be defined as follows: given a system S, find function $f : i \to a$ such that if i is an input to S, then S would produce o as output. In other words, the problem of function extraction is about extracting the function (or input to output relation) supported by system S. In case of our problem, S is a code generator.

In this section, we describe how EISSEC extracts the mapping function supported by a code generator. It does this by relying on a technique called *symbolic execution*. Before getting to the details of EISSEC, we first describe the required background on symbolic execution.

4.1 Background: Symbolic Execution

Software testing plays a central role in software development and maintenance. One of the important challenges in software testing is to generate test cases that maximize program coverage. Given the complexity of modern software (such as browsers, file editing tools, etc,), there is a general consensus in the software community that manual testing is unlikely to achieve adequate program coverage. Symbolic execution is one of the many techniques that can be used for generating test cases that achieve higher coverage of programs.

Symbolic execution [49] is a program analysis technique that executes programs with symbolic rather than concrete inputs. Because programs are executed with symbolic inputs, program data flow now propagates symbolic values from inputs to outputs. Program operations also operate on symbolic values. Program variables, thus, contain symbolic expressions. The effect of executing a program symbolically is that the output of the program is expressed as a function of its input. In other words, symbolic execution helps us uncover the function implemented by the program. During execution, symbolic executions also maintains a path condition which encodes constraints on the inputs that reach a particular program point. These path conditions are updated for every branch point in the program. Test cases that cover all program paths can then be generated by solving the collected path conditions (constraints) using constraint solvers.

Symbolic execution maintains a symbolic state, which maps variables to symbolic expressions, and a symbolic path constraint *PC* over symbolic expressions. *PC* accumulates constraints on the inputs that trigger the execution to follow the associated path. At every conditional statement `if (e) S1 else S2`, *PC* is updated with conditions on the inputs to choose between the alternative paths. A new path condition PC' is created and initialized to $PC \wedge \neg\delta(e)$ in an "else" branch and *PC* is updated to $PC \wedge \delta(e)$ in a "then" branch, where $\delta(e)$ denotes the symbolic predicate obtained by evaluating e in symbolic state δ. Note that, unlike concrete execution, both branches can be taken, resulting in two execution paths. If path condition becomes unsatisfiable, symbolic execution along that path is terminated. Satisfiability is checked with a constraint solver.

Once we have traversed all paths inside a program and have collected *PC* for every path, then these *PC* can be solved using a constraint solver. Solving a path condition for a path produces a mapping between input and output represented by that path. Thus, by solving the path conditions for all paths, we can extract the input to output function supported by the program.

As mentioned earlier, in the software testing domain, symbolic execution is used to generate high coverage test inputs for a program [19, 21, 82, 39]. This is done by querying a constraint solver upon path termination and asking it for an input value that satisfies the path condition for that path. Note that, though there may be more than one set of input values which follow the same program path, for ensuring program coverage, obtaining one input value is enough. Function extraction, on the other hand, needs to know all the input values that satisfy a path condition, and the corresponding output produced by each of those values. Thus, the problem of generating high-coverage test inputs for a program is a sub-class of the function extraction problem addressed in this dissertation.

4.1.1 A simple code generator and its symbolic execution

We now illustrate how symbolic execution of a code generator can be performed in order to extract its function (or the semantic model). To describe a simple code generator, consider a simple hypothetical instruction set for a processor which supports (a) only four registers `r1` to `r4`, (b) only `mov` and `add` instructions, (3) a constraint that a valid source operand can be a register, a memory address (with memory is accessed indirectly using registers only), or an immediate integer; a valid destination operand can be only a register or a memory location, and for `mov`, both source and destination operands cannot be memory at the same time. The grammar for this instruction set and a mapping table used by a code generator for such a processor is shown in Figure 20. The mapping table uses RTL as IR.

Note that the mapping process enumerates some aspects of the instruction set, (e.g., the name of the instruction and the operand type). Other aspects are not enumerated (e.g., the actual registers or integer constants). Instead, the mapping uses expressions to refer to components of the RTL, denoted in the example using the notation `%N`.

A code generator implementing this mapping table is shown in Figure 21. To simplify the illustration of our technique, we used an implementation in a declarative language. Note that the code generator takes an RTL instruction as the input and produces an assembly instruction as the output. For the simple code generator, assembly instructions are represented as strings.

Because RTL has a number of subfields, it can be represented using nested expressions. To simplify understanding, we represent RTLs as first-order terms (same as in `LISC`). Input `rtl` thus can thus be initially bound to the term `(set D S)`. Note that the code generator only accepts RTLs with their operator (first subfield) set to `set`. `D` and `S` are symbolic variables for destination and source of `rtl` respectively (`set` is an assignment from `S` to `D`.) Once we bind `D` and `S`, symbolic state τ will be updated with the bindings. When symbolic

$$\begin{array}{rcl}
\mathit{Instruction} & ::= & \texttt{mov}\ S, D \mid \texttt{add}\ S, D \\
S & ::= & \texttt{r}N \mid (*\texttt{r}N) \mid \$\mathit{Int} \\
D & ::= & \texttt{r}N \mid (*\texttt{r}N) \\
N & ::= & 1..4
\end{array}$$

`mov r%1, r%2`	$\longrightarrow$	`(set (reg %2)(reg %1))`
`mov (*r%1), r%2`	$\longrightarrow$	`(set (reg %2)(mem (reg %1)))`
`mov $%1, r%2`	$\longrightarrow$	`(set (reg %2)(const_int %1))`
`mov r%1, (*r%2)`	$\longrightarrow$	`(set (mem (reg %2))(reg %1))`
`add r%1, r%2`	$\longrightarrow$	`(set (reg %2)(plus (reg %2)(reg %1)))`
`add (*r%1), r%2`	$\longrightarrow$	`(set (reg %2)(plus (reg %2)(mem (reg %1))))`
`add $%1, r%2`	$\longrightarrow$	`(set (reg %2)(plus (reg %2)(const_int %1)))`
`add r%1, (*r%2)`	$\longrightarrow$	`(set (mem (reg %2))(plus (mem (reg %2))(reg %1)))`

Figure 20: **Instruction set and mapping table for a simple code generator**

execution reaches the **match** statement on line 5, there are two possibilities: `S` is a `plus` rtl expression or it is not. It might happen that `S` is bound to some rtl expression in the symbolic state τ, and in such a case either the "**if**" or the "**else**" branch would be taken. Let us say that `S` is bound to `(O, _)` in τ. To decide which branch to take, the symbolic execution system queries a constraint solver on the constraint `O == plus`. If the constraint solver returns *true* then it means that `O` is bound to the value `plus` before line 5, and in that case only the "**if**" branch can be taken. A *false* return value means that `O` is bound to some value other than `plus` before line 5, and in that case only the "**else**" branch will be taken. A return value of *satisfiable* means that `O` is not bound to any value — in other words, it is unbound — and hence the condition `O == plus` can be satisfied by assigning `O` with `plus` in the "**if**" branch and by setting `O` $\neq$ `plus` in the "**else**" branch. Thus in such a case, both the "**if**" and the "**else**" branches would be taken. When the symbolic execution system needs to follow both the "**if**" and the "**else**" branches, it updates state τ with the conditions on `O` which hold true in that branch: `O = plus` in **if** branch, and `O` $\neq$ `plus` in **else** branch. Program execution is effectively divided into two at such a branch point. Also, *PC* is updated with `O = plus` in **if** branch and `O` $\neq$ `plus` in **else** branch. With this explanation, it is easier to see how the tree presented in Figure 22 represents a

```
let recog (rtl) : asm =
  match rtl with
    (SET D S) −>
      (match S with
        (PLUS _) −> (∗ add ∗)
              let (PLUS _ S1) = S in
              let (O1 D1) = D in
              let (O2 S11) = S1 in
              if O1 = reg then
                if O2 = reg then
                  ‘‘add rS11, rD1’’
                else if O2 = mem then
                  let (reg S111) = S11 in
                  ‘‘add ( ∗S111), rD1’’
                else if O2 = const_int then
                  ‘‘add $S11, rD1’’
              else if O1 = mem && O2 = reg then
                let (reg D11) = D1 in
                ‘‘add rS11, ( ∗rD11)’’
        | _ −> (∗ mov ∗)
              let (O S1) = S in
              let (O1 D1) = D in
              if O1 = reg then
                if O = reg then
                  ‘‘mov rS1, rD1’’
              else if O = mem then
                let (reg S11) = S1 in
                ‘‘mov ( ∗rS11), rD1’’
              else if O = const_int then
                ‘‘mov $S1, rD1’’
            else if O1 = mem && O = reg then
              let (reg D11) = D1 in
              ‘‘mov rS1, ( ∗rD11)’’
      )
    | _ −> raise error
```

Figure 21: **OCaml code for a simple code generator**

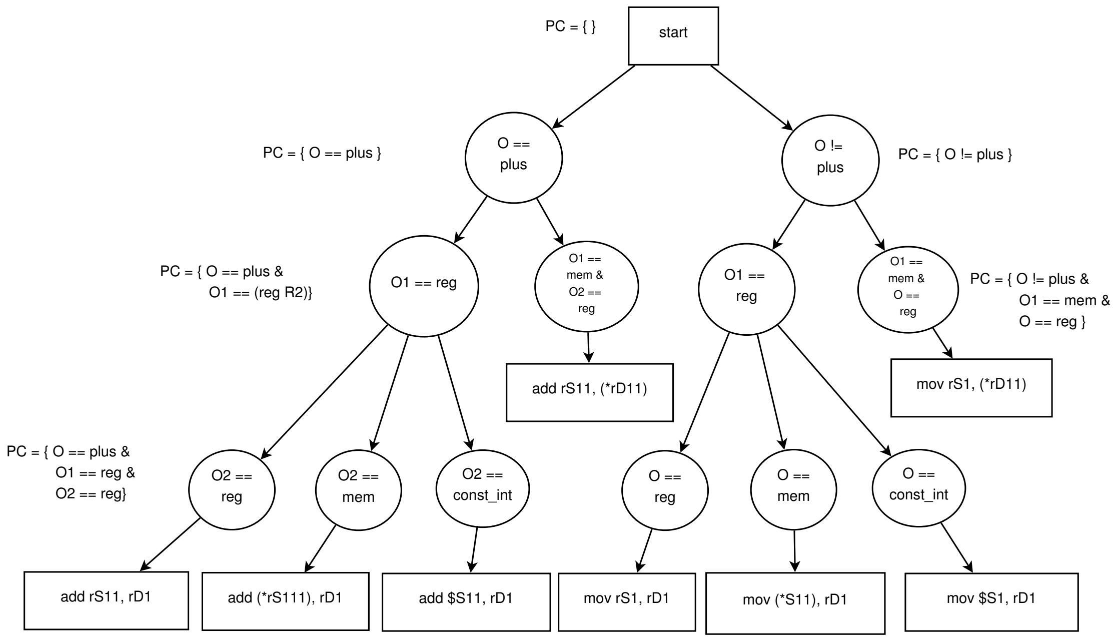


Figure 22: **Symbolic execution tree of the simple code generator**

symbolic execution of a simple code generator's OCaml code. The root node of such a tree is the entry point of the code generator, and the leaves correspond to assembly instructions supported by the code generator. A path from the root to a particular leaf node represents an RTL to assembly mapping pair generated by the code generator. Non-leaf nodes, on the other hand, correspond to branch points in the program and represent the conditions on the input that must be satisfied in order to generate the assembly instructions at the leaves. For instance, in order to generate the assembly instruction "`add r%1, r%2`", the input RTL must be such that: `O = plus && O1 = reg && O2 = reg`. The combination of all these constraints is actually a *PC* for the path with "`add rS11, rD1`" as a leaf node. Satisfying all the constraints in the *PC* generates input rtl `R` which produces "`add rS11, rD1`" as the assembly instruction.

Note that this is where the difference between function extraction and test case generation shows up clearly. To be precise, there could be more than one RTLs (i.e., R_1, R_2, ..) that produce "`add rS11, rD1`". For instance, there are 16 RTLs (corresponding to 4 possible values of `rS11` and 4 possible values of `rD1`) that map to "`add rS11, rD1`". For the problem of test case generation, it is sufficient to produce one RTL (but may not be all) that produces the assembly instruction. This is because all 16 RTLs represent the same path in the code generator, and thus even a single RTL is enough to cover that path. The problem of function extraction, on the other hand, by definition demands that the relation between all 16 RTLs and "`add rS11, rD1`" instruction is obtained. More specifically, note that, similar to test case generation, if we only select single RTL, then we would be extracting an incomplete mapping function (or incomplete semantic model). That is why for function extraction, the constraint solver needs to return to us *all* the possible values that satisfy *PC*. Once the program execution tree is generated, *PC*s for different paths are solved by using a constraint solver. All the solutions provided by the constraint solver for a particular *PC* are the input assignments which map to the output represented by the path for *PC*. Solving all such *PC*s and obtaining all of their solutions gives us the mapping table used by the given code generator.

4.1.2 Concolic execution of the simple code generator

The symbolic execution technique described above is called *classical symbolic execution*, and it was introduced by King in his 1976 paper [49]. Unfortunately, applying classical symbolic execution to extract a function from modern complex software is a very challenging problem. The first challenge is the exponential increase in the number of program paths. Notice that the number of paths explored in the program tree of Figure 22 increases exponentially as more branch points are visited. The second challenge is that all code that can be reached from the program whose function we want to extract needs to be executed symbolically. In the context of code generators, the second limitation proves critical. Code generators are complex pieces of software with thousands of lines of code and, more importantly, they may interact with other components of a compiler (e.g., instruction selector, register allocator, etc). Because we are only interested in the function implemented by the code generator, we only want to perform symbolic execution of a part of a compiler. This is where the idea of *concolic execution* [82] comes into the picture.

Concolic execution is an optimization over symbolic execution that allows us to execute only a part of a program symbolically. To achieve this, it follows a simple idea: restrict symbolic execution to only interesting paths and visit the remaining paths concretely. Concolic execution stands for a mix of symbolic and concrete execution. Conceptually, it operates at a middle ground between symbolic and concrete executions. Specifically, instead of making all program inputs symbolic, only the program inputs of interest are made symbolic, while the remaining inputs are made concrete (i.e., concrete values are assigned to these inputs.) The reason concolic execution reduces the number of program paths visited symbolically is because whenever all the inputs involved in a branch condi-

```
1  let recog (rtl, rtl_id) : asm =
2    if (isRTLInCache(rtl_id) == 1)
3    then getRTL[rtl_id]
4    else
5      match rtl with
6          (SET D S) −>
7          (match S with
8                ...
```

Figure 23: **Modified OCaml code for the simple code generator**

tion are concrete, the condition is evaluated concretely, and only one of the two possible branches is taken. On the other hand, whenever at least one of the inputs involved in a branch condition is symbolic, *PC* is updated appropriately as explained earlier, and both branches are explored. Note that with concolic execution, we can effectively prune one of the two possible subtrees at a branch point. This pruning effectively controls the explosion in the number of program paths.

Concolic execution of the simple code generator can be performed as follows. To demonstrate an example of the need for concolic execution, let us modify the simple code generator. The snippet of the modified code is shown in Figure 23. The modified code takes one more argument `rtl_id` (which represents a unique id for every `rtl`) as input. Then, the code uses the input `rtl_id` to check if we have already translated the input RTL by calling a function `isRTLInCache`. Assume that the function `isRTLInCache` uses a hash table implementation from some library for implementing the cache, and that `rtl_id` of `-1` is never found in the cache. It then returns `1` if the `rtl_id` is found in the cache (in which case the code calls `getRTL` to return the cached RTL), `0` otherwise (in which case it explores the decision tree to translate the input RTL. The code to explore the decision tree is same as that of the original simple code generator (hence not shown).

Given this code snippet, it is clear that we are not interested in exploring the code of `isRTLInCache` and `getRTL` symbolically. So the question is how to handle this piece of code? What we do in this case is to simply assign a concrete value of -1 to `rtl_id`, but assign a symbolic value of `(O, X, (C, A, B))` to `rtl`. This way, we always execute `isRTLInCache` concretely. Because `isRTLInCache(-1)` always returns `0`, the execution will always go into the **else** branch at line 4. This is the code that we are only interested in executing symbolically. In other words, we are pruning **if** branch at line 2 in the program execution tree, thus reducing the number of program paths. The program execution tree

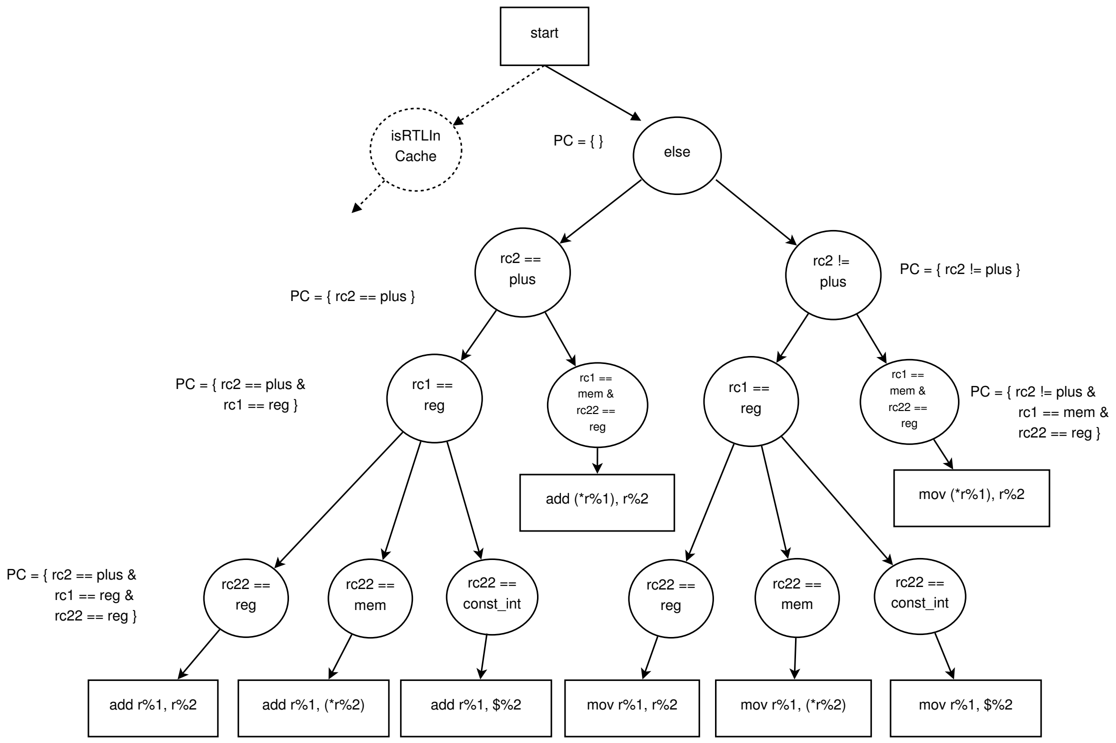


Figure 24: **Concolic execution tree of the simple code generator**

of the simple code generator's concolic execution will look like Figure 24. The pruned subtree is shown with dotted lines.

4.2 Design

EISSEC is a concolic execution system for GCC's code generator. Given the limitations of classical symbolic execution discussed earlier, the complexity of GCC[14], and the fact that we are only interested in extracting a function of the code generator, we decided to employ concolic execution instead of symbolic execution. By executing a code generator concolically EISSEC can thus obtain all the RTL-to-assembly mapping pairs (semantic model) supported by the code generator.

One of the significant complications introduced by concolic execution is the existence of both symbolic and concrete state and the scenarios where there are assignments from one state to another. To give a simple example, a symbolic variable could be assigned a

[14]There are plenty of components in code generator which perform functionalities other than mapping RTL to assembly, and extraction of RTL-to-assembly pairs does not demand executing these components symbolically.

concrete value, and a concrete variable could be assigned a symbolic value. The former is a much easier scenario to deal with, but the latter is the case of "explosion" — by explosion, we mean in such case one has to effectively enumerate all possible values that can be taken by the symbolic variable and assign all of those values to the concrete variable. Moreover, we need to ensure that any natively executed code will not access symbolic variable state.

In this section, we discuss the issues produced by concolic execution of GCC's code generator and how `EISSEC` design handles them. In general, our strategy is to keep as much of GCC's state as concrete, and only mark global variables, functions, etc, that are needed to extract the code generator function as symbolic. Another important aspect of `EISSEC` is how it addresses the function extraction problem. There exists a number of popular symbolic execution systems such as KLEE [19], DART [39], EXE [21], CUTE [82], etc. But these systems are used in the context of test case generation, and not in the context of function extraction. Although these systems do not target the function extraction problem, there are many parallels between them and `EISSEC` in terms of design of concolic execution systems.

4.2.1 Input program

Symbolic execution systems can take input programs in different forms — some take programs in a compiled binary form, while others take program source code. There are systems like KLEE [19] that take programs in a byte-code form (program source code is compiled to byte-code using a compiler.) Systems that perform symbolic execution of a program executable have an advantage of being able to perform symbolic execution of any program (even COTS binaries). Since in our case, the source code of GCC's code generators is available, `EISSEC` operates on the source code. `EISSEC` transforms the C source of the code generator using a source-to-source transformation approach, and the transformed source code is then compiled and executed to perform concolic execution of the code generator.

4.2.2 Overall flow

Our concolic execution engine is implemented as a source-to-source transformation of GCC's code[15] (which is written in C) using CIL-1.4.0 [66]. Because we can use native execution as well, not all of GCC's code needs to be transformed. Specifically, transforma-

[15]We used GCC-4.6.4 — the most recent version of GCC at the time we began our project — for our purpose.

tion starts at the function `recog` that generates target assembly instructions from the input RTL instructions by matching an RTL snippet against an MD rule. Any function called by `recog` with possibly symbolic arguments will also be transformed to support concolic execution. This may not always be desired. To override default choices in terms of transforming (or not transforming) certain functions, we rely on annotations.

Before describing the transformation, it is necessary to understand how and when concolic execution comes into play. After performing all the necessary initializations, we introduce a call to `recog` by setting the RTL code that needs to be translated as symbolic. All other objects in GCC's memory are concrete at this point. Concolic execution will explore all possible paths in `recog` and functions invoked from it. Each such path will ultimately terminate with a successful return from `recog` or a failure. In the former case, assembly code is printed into a string-valued variable returned by `recog`.

Recording the mapping requires obtaining the symbolic values of the argument and output of `recog` corresponding to a single execution path, and storing the pair. Most previous implementations of symbolic and concolic execution have used SAT/SMT solvers as their constraint solvers. Unfortunately, these solvers are optimized to find a single instantiation that satisfies a formula. In our case, we cannot use a single instantiation beacuse that would capture only one of the possible translations that may be performed on a program path. Constraint solvers, such as those included in constraint logic programming languages, are better engineered to produce all possible instantiations. That is why we use constraint logic programming based solvers for our purpose.

4.2.3 Source-to-source transformation for concolic execution.

Because concolic execution systems allow some of the program inputs to be symbolic, the first design challenge for `EISSEC` is how to represent symbolic values in the system. Program variables, in addition to concrete values, can now contain symbolic values. Furthermore, because variables can now have symbolic values, program statements which operate on symbolic values need to be handled. Symbolic execution systems transform such statements so that the effect of executing a statement with symbolic inputs captures the effect of executing the statement with concrete inputs. For example, for $x = y + 1$, if symbolic value of y is Y, then the transformation of the statement would be such that the symbolic value of x would be $Y + 1$. `EISSEC` transforms all of the statement types supported in C, such as assignments, conditionals, function calls, etc.

We describe our source-to-source transformation using an abstract language that captures the essential features of C:

```
P ::= S
S ::= A | I(V,S,S) | W(V,S) | S;S
A ::= R | V = R | *V = V
R ::= O | O op O | op O | f(V,...,V)
O ::= V | C
```

Here, `P` stands for program, `S` for statement, `A` for assignment statement, `I` for if-then-else, and `W` for a while statement, `L` and `R` for lhs and rhs of assignments, `V` for a variable (not a memory location), and `C` for a constant.

The transformation is specified using a function **T** that takes two input parameters: a C-program statement `S` and an abstract environment $\mathscr{E}$. While the abstract environment could hold any type of information about variables, in our case $\mathscr{E}$ says whether a variable is definitely symbolic ($V_{sym} = true$), possibly symbolic (represented as $V_{sym} = \top$), or definitely not symbolic ($V_{sym} = false$). The output of **T** is a pair ($\mathtt{S}'$, $\mathscr{E}'$), where $\mathtt{S}'$ is the transformed version of the input statement `S`, and $\mathscr{E}'$ is the new abstract environment.

Rules governing symbolic vs non-symbolic variables. Note that we rely on both compile-time approximation of whether a variable contains a symbolic value and an accurate run-time information. The former is used to avoid transformation of large sections of code that may never manipulate symbolic state, while the latter is used to maximize concrete execution at runtime. The compile-time symbolic type of a variable with respect to its abstract environment $\mathscr{E}$ is represented using function $cType_{\mathscr{E}}$, while the runtime type is represented using function $rType_{\mathscr{E}}$. Possible values of for compile-time types are $\{conc, sym, \top\}$, whereas possible values for runtime type are $\{conc, sym\}$.

The rules governing variable type need to be such that they ensure the integrity of symbolic and concrete states. Because during concolic execution, both symbolic and concrete states exist simultaneously, it is necessary that these states do not interfere with each other. To be precise, it is necessary to ensure that accesses of all symbolic variables in the program source code are identified correctly and are modified to access symbolic state. Additionally, it is also necessary that all concretely executed program statements do not modify symbolic state. The fact that the GCC-4.6.4 code generator is written in C makes this challenging. Specifically, pointers in C make this issue complex, as they make reasoning of locations that are accessed difficult. Fortunately, features of GCC's code generator source code make addressing this challenge a bit easier.

In order to ensure integrity of symbolic state, we need to ensure that all accesses that clobber a symbolic state are detected by our system. First, pointer dereferences pose a challenge for this requirement because pointer arithmetic and aliases make reasoning about pointer dereferences hard. Symbolic pointers pose an even more significant issue. In particular, dereference of a symbolic pointer demands enumeration of all concrete addresses that could be taken by that pointer at the time of dereference and dereference of all those memory locations (KLEE [19] follows this approach.) This makes the system very inefficient, and introduces complications (ensuring correct and accurate update to memory without corrupting symbolic state)[16]. That is why we allow symbolic pointers in very limited contexts. Second, because global objects are visible from "everywhere" in the program, and because we are performing concolic execution, the requirement of ensuring integrity also means that we need to limit which objects are considered symbolic. GCC has many global objects, and most of them are not accessed in the code generator. Such objects can be safely considered concrete. We thus annotate global objects which are accessed by the code generator and consider only them as symbolic.

Based on the above discussion, the following rules govern if and when a variable can be symbolic or concrete: (1) if a variable is annotated as being symbolic (which we do for selected global variables only), or if a variable's address is not taken then it is considered symbolic, (3) variables of pointer types are considered non-symbolic unless they are annotated otherwise (in our case, only pointers of type RTL were annotated as symbolic), (4) variables of type array, structure and union are considered symbolic if they satisfy either of the above requirements, and they are accessed using an offset which is a compile-time constant (e.g., `a[2]` is fine, but not `p = &a[0]+2; *p=..`) — in other words, accessing them using pointer arithmetic is not allowed[17]. Fortunately, in case of GCC, most of the code uses pointers in accordance with our rule, but there were a few places where symbolic arrays were accessed using pointer arithmetic, and we rewrote such cases to use constant offsets.

Fortunately, GCC's code generator conforms to most of these restriction, but in the following case, we rewrote the code to enforce these constraints. To be precise, the case of

[16]While an alternative approach of treating memory objects symbolically [82] can work in such a case, given the large number of GCC's in-memory objects, we do not want to treat them symbolically.

[17]This does not mean that we refuse a program which uses pointer arithmetic. In such case, we treat pointer arithmetic concretely.

Array of `rtx` accessed using a pointer needed a rewriting. Operands of an RTL instruction are stored in an array (declared as `rtx operands[]`) of `rtx` pointers. GCC's code was accessing the elements of `operands` using pointer arithmetic, such as `rtx* o = operands; ..; *(o+2) = ...` As per rule 4 above, this was not allowed. So we rewrote this access as `operands[2] = ...` Note that, although `operands[2]` can also be thought of as a syntactic sugar over pointer arithmetic and dereference syntax, the expression `operands[2]` allows reasoning about the element of the array that is being accessed. This allows generating appropriate constraints easily. There were 12 places in the code generator where this change was necessary.

Once program variables are marked as *sym* or *conc* as per the above rules, we use standard data-flow analysis to compute symbolic types of the other program variables. Specifically, the following rules were used.

- First, if output (V_O) of a program statement is marked as definitely symbolic/non-symbolic (i.e, either $cType_{\mathscr{E}}(V_O) = conc$ or $cType_{\mathscr{E}}(V_O) = sym$), then we do not need to propagate any symbolic type information from input to output.

- Otherwise, if all input variables of a program statement are definitely concrete, then the output is marked as concrete ($cType_{\mathscr{E}}(V_O) = conc$).

- On the other hand, if all input variables of a program statement are definitely symbolic, then the output is marked as symbolic ($cType_{\mathscr{E}}(V_O) = sym$).

- Otherwise, $cType_{\mathscr{E}}(V_O)$ of output is set to $\top$.

These data-flow analysis rules are applied at function granularity by starting with the types of formal parameters and using them as input for the type propagation rules. $\mathscr{E}$ for the function is updated when the type of new variables is derived, and it is used to transform program statements.

Before we talk about the transformation of each program statement from the abstract language, we will look at the high-level idea behind the transformation, shown in Figure 25. The figure shows how we transform a statement `S` depending on the compile-time type or runtime type of inputs (V_I) and output (V_O). **T** is the transformation function which takes `S` and $\mathscr{E}$ as input and emits the transformation of `S` and the statement to update the abstract environment $\mathscr{E}$ as an output.

- $cType(V_O) = conc$: the high-level idea captured in this case is that if the compile-time symbolic type of the output is *conc* (i.e., the variable cannot hold a symbolic

Conditions on V_O	Conditions on V_I	**T**
$cType_{\mathscr{E}}(V_O) = conc$	$\forall V_I,\ cType_{\mathscr{E}}(V_I) = conc$	`S` $\mathscr{E}[rType_{\mathscr{E}}(V_O) \leftarrow conc]$

	$\exists V_I,\ cType_{\mathscr{E}}(V_I) \neq conc$	**if** $\forall V_I,\ rType_{\mathscr{E}}(V_I) = conc$
		  `S`
		else
		  enumerate
		$\mathscr{E}[rType_{\mathscr{E}}(V_O) \leftarrow conc]$

$cType_{\mathscr{E}}(V_O) = sym$	$\forall V_I, cType_{\mathscr{E}}(V_I) = \top$	$addCons(V_o = \mathtt{S}(V_{I_1},..,V_{I_n}))$ $\mathscr{E}[rType_{\mathscr{E}}(V_O) \leftarrow sym]$

$cType_{\mathscr{E}}(V_O) = \top$	$\forall V_I, cType_{\mathscr{E}}(V_I) = \top$	**if** $\forall V_I,\ rType_{\mathscr{E}}(V_I) = conc$
		  `S`
		  $\mathscr{E}[rType_{\mathscr{E}}(V_O) \leftarrow conc]$

		else $addCons(V_o = \mathtt{S}(V_{I_1},..,V_{I_n}))$ $\mathscr{E}[rType_{\mathscr{E}}(V_O) \leftarrow sym]$

Figure 25: **Transformation logic for** V_O **=** $\mathtt{S}(V_{I_1},..,V_{I_n})$

value), then we cannot execute S symbolically. This is because there is a constraint on the symbolic type of output. In such case, if all inputs are concrete then we can execute S concretely. Note that the runtime check (**if**) has to be emitted before statement S to ensure that all inputs are concrete at runtime. If this runtime check fails, then we enumerate all possible concrete values for symbolic inputs, and execute S for each of them. We will specify how to perform such enumeration for each of the program statement from our abstract language shortly. But the intuitive description of enumeration is given after discussing all the cases.

- $cType(V_O) = sym$: when the compile-time type of output is symbolic, then irrespective of compile-time type of inputs, we generate a constraint which captures relation between inputs and output. The generated constraint is then sent to constraint solver. *addCons* is thus a function used to send a constraint to a constraint solver.

- $cType(V_O) = \top$: when we do not know the compile-time symbolic type of output, then depending on the run-time symbolic type of all inputs, we will either execute the statement concretely (and set run-time symbolic type of output *conc*), or generate a constraint (and set the run-time symbolic type of output to *sym*). Note that the case of compile-time symbolic type of all inputs being *conc* will not arise for this case, as our data-flow analysis in such a would mark the output as *conc* at compile-time.

Enumeration (symbolic to concrete conversion). Whenever there is an operation involving at least one symbolic argument, we must either have a symbolic version of the operation, or we must enumerate and try all possible values of the symbolic argument. Additionally, whenever a symbolic value is assigned to a concrete variable, we must enumerate all possible values that could be taken by that symbolic variable at that program point, and assign each value to the concrete variable one by one. This leads to a significant increase in the number of program paths, but we do not anticipate such cases to arise frequently. We make concrete variables symbolic as often as possible. Additionally, to reduce the number of program paths that we generate by enumeration, we have devised a number of optimizations which we discuss in Section 4.4. Optimizations which reduce the number of program paths are crucial to complete the concolic execution of the code generator.

Figure 26 shows the program statement that will be transformed (in column `S`), the logic behind its transformation (in column **T**), and the updated value of the runtime variable type. Column **T** contains the actual transformation of `S`, while the conditions checked at compile-time to produce the transformation are shown in column $\mathscr{C}$. Following the high-level logic for statement transformation discussed earlier, we do not show cases for which we execute statement concretely (case of `S` being produced in column **T**). Instead, in this figure, we only show cases where the statements are transformed.

Transformations.

- `V` = `O1 op O2`: this case applies for statements containing binary expressions. It can also be thought of as a general version of unary expressions (so we do not show transformation of unary statements.) The idea behind transformation of this statement is same as the high-level idea. Only the case of enumeration deserves some explanation. Specifically, if the compile-time type of output is *conc*, and the run-time type of any of the inputs is *sym*, then we have to enumerate all possible concrete values that can be taken by the symbolic value and use those concrete values to execute the statement concretely. This is the idea captured by the runtime check inserted for this case. For a symbolic execution system, this is the most undesirable situation because enumeration leads to creation of numerous symbolic processes.

S	$\mathscr{C}$	**T**

V = O1 op O2	cType(V) = sym $\|$ cType(V) = $\top$	*addCons*(V = O1 op O2)
		if $\forall O_i$, rType$_{\mathscr{E}}(O_i)$ = conc
		  $\mathscr{E}$[rType(V) $\leftarrow$ conc]
		else $\mathscr{E}$[rType(V) $\leftarrow$ sym]

	cType(V) = conc	**if** rType$_{\mathscr{E}}$(O1) = sym $\wedge$
		  foreach o1 $\in$ conc(O1)
		    foreach o2 $\in$ conc(O2)
		      (V = o1 op o2)
		elif rType$_{\mathscr{E}}$(O1) = sym
		  foreach o1 $\in$ conc(O1)
		      (V = o1 op O2)
		elif rType$_{\mathscr{E}}$(O2) = sym
		  foreach o2 $\in$ conc(O2)
		      addCons(V = O1 op o2)
		else *addCons*(V = O1 op O2)
		$\mathscr{E}$[rType(V) $\leftarrow$ conc]

*V1 = V2	// No condition
	checked as V1
	must be concrete
		if rType$_{\mathscr{E}}$(V2) = sym
		  (**foreach** v2 $\in$ conc(V2)
		    fork_and_continue();
		    *V1 = v2;)
		else (*V1=V2);
		$\mathscr{E}$[rType(V2) $\leftarrow$ conc]

f(V1,...,Vn)	`f_cons is defined`	**if** $\exists i$, rType$_{\mathscr{E}}$(Vi) = sym
		      f_cons(V1,...,Vn, V)

	`f_trans exists`	**if** $\exists i$, rType$_{\mathscr{E}}$(V$_i$) = sym & rType$_{\mathscr{E}}(V_{P_i})$ = conc
		    (**foreach** $v_i \in_i$
		      fork();
		      $V_i = v_i$;
		      f(v1, .., vn))
		else (*f_trans*(V1, ..., Vn, V1$_{meta}$, ..., Vn$_{meta}$))

		f(V1,...,Vn)
S1; S2		(S1n, $\mathscr{E}_1$) = **T**(S1, $\mathscr{E}$) (S2n, $\mathscr{E}_2$) = **T**(S2, $\mathscr{E}_1$) (S1n; S2n;, $\mathscr{E}_2$) $\mathscr{E} = \mathscr{E}_2$

I(V, S1, S2)		**if** rType$_{\mathscr{E}}$(V) = sym
		  let ret = *addCons*(V = true);
		    if ret = *SAT*
		      if *fork*() = *child*
		        addCons(V = true);
		        T(S1, $\mathscr{E}$);
		      else
		        addCons(V = false);
		        T(S2, $\mathscr{E}$);)
		    else
		      addCons(V = false);
		      T(S2, $\mathscr{E}$);
		else I(V, S1, S2);

Figure 26: **Transformation function for different program statements**

- $*$`V1` $=$ `V2`: this case applies to pointer-dereferences, and since pointer being dereferenced is concrete at compile-time, we simply generate a runtime check for enumeration (as in the previous case). In case of enumeration, since `V2` was symbolic, we need to update its runtime type to concrete.

  Dereferences of symbolic pointers introduces an interesting challenge. Specifically, since the pointer value is symbolic once must concretize the value in order to dereference it. This leads to the issue of enumeration discussed earlier. Fortunately, in GCC's code generator, there are no direct pointer dereferences. Instead, pointers to compound data types are dereferenced to access their fields. For symbolic `rtx` pointers, `rtx` being an object, all dereferences were to access the fields of the object. GCC uses special functions, called *accessor functions* to access fields of `rtx`. For instance, assuming that `rtx` were to have a field named `foo`, then GCC would have a function named `GET_FOO(x)` which would be defined as `GET_FOO(x) = x->foo`. In other words, accessor functions define an interface to access fields of RTL instruction, and thus hide the representation of RTL object. We exploit this observation to handle field references of symbolic `rtx`. Specifically, for each accessor function (around 50 in total), we manually write a new symbolic accessor function which would model the semantics of GCC's accessor functions on symbolic RTL instruction. For instance, for `GET_FOO`, we would generate `GET_FOO_CONS(x)` which would generate a constraint that `x` is of type `rtx` and return the symbolic value of `x->foo`. Total code for symbolic accessor functions constitute around 2000 lines of C code. We need to manually write symbolic versions of accessor functions because we want to represent `rtx` type specially in our constraint system. Specifically, we treat `rtx` as a compound type and hide its details (such as fields, their ordering, etc) from the rest of the code. If we transform accessor functions as any other functions, then we will not be able to control the symbolic representation of `rtx` type.

  Note that the approach of allowing accesses to compound objects only via constant compile time offsets allows us to handle references to elements of compound symbolic types like arrays, structures, etc. For instance, the field dereferenced in case of array would be the element of the array being accessed. For these compound types, in addition to `GET` function, we also define `SET` function. Note that this also allows us to handle non-`rtx` type pointer dereferences also. By encapsulating accesses to compound symbolic types, we can change the underlying symbolic representations of these types easily.

- `f(V₁,...,Vₙ)`: since we expect most (if not all) global variables to be concrete, symbolic values will flow into local variables primarily from function parameters (same rules applied to local variables are applied to formal parameters to determine which of them are symbolic.) In particular, a call $f(x,y)$ will need to be transformed into $f\_trans(x,y,x_{meta},y_{meta})$. x_{meta} and y_{meta} allows us to pass meta-information about symbolic actual arguments from caller to callee, and it is used to assign symbolic actual arguments in the function prologue. We also need to transform f's body to produce f_trans's body. f_trans is produced by transforming all statements in its body. Note that in the figure V_{P_i} represents the formal parameter in the function prototype corresponding to actual argument V_i.

  Note that f_trans is different than f_cons because f_cons represents the transformed version of an accessor function f, while f_trans represents the transformed version of any other function f defined in the code generator.

  Note that original function f will continue to exist, in case it is called from untransformed code. But we do not call f from transformed code even if the arguments are all concrete. This allows us to support cases where a global variable (accessed within f) contains a symbolic value. This is important because if we were to call untransformed f in case all of its arguments are concrete, we would update the symbolic global variable concretely, and would miss constraints on that variable.

  Whenever a function call is to be transformed, we check if the function being called is a standard library (such as `libc`) function (e.g., `malloc`). In case it is a standard library function, we do not transform the call, and keep the original call as it is. If the function is not a standard library function and we have already transformed the definition of such a function then `transformed(f)` returns true, and in such case, we emit the code shown next to it. Thus, in short, we do not transform any functions outside of GCC's code generator source code.

  The transformed named for GCC's accessor functions are obtained by appending `_CONS` to the accessor's original name. That is why we first check if the function being transformed is an accessor function and if we have its symbolic version. In case it is, then we invoke symbolic version of the function in place of the original call.

  One another type of function which we manually transform is the one which relates to printing of symbolic operands of assembly instructions. Note that the code for printing assembly instructions mostly consist of `libc` functions such as `write` or `fputs`. If we want to print symbolic operands using these functions, then we have to transform these functions as well. This adds unnecessary complexity to the system. Instead, we write wrappers (similar to accessor functions for `rtx`) and handle the printing of symbolic operands in them. Wrappers and accessor functions in total constitute around 1.2K lines of code.

As a last option in the transformation of function calls, if the function is neither transformed nor an accessor function, then we execute that function call concretely. This is captured in the last case.

- `S1;S2`: this case represents a sequence of two statements. The way we transform this sequence is by first transforming `S1` and then feeding the output $\mathscr{E}_1$ of $\texttt{S}_1$ to transform $\texttt{S}_2$. The output $\mathscr{E}_2$ is then the output $\mathscr{E}$ of the sequence.

- `I(V,S1,S2)`: if-then-else statement represents an important case in statement transformation because it is where the symbolic processes will fork their execution into two. First of all, if the condition (of *if* statement) is concrete, then we can simply execute the statement concretely. In such case, symbolic process will not fork its execution.

  On the other hand, if the condition is a symbolic expression, then we may need to fork the execution. To determine if we need to fork, we first check if the condition is satisfiable (*SAT*) by sending the constraint to the constraint solver. If the condition is unsatisfiable, then we can only take *else* branch. An example of an unsatisfiable constraint is $V = 10$, when V is already bound to 5 by the statement $V = 5$. In the else branch, we add the constraint that V is $false$ as that is what the semantics is. On the other hand, if the condition is satisfiable (e.g., condition $V < 5$ is satisfiable, if V is not bound to any value), then we need to fork the execution. Conceptually, this is similar to Unix `fork()` semantics. After fork, child continues with the *if* branch, while parent continues with the *else* branch.

  Treatment of *if* statement can also be thought of as similar to generating one or two concrete values from V. Specifically, similar to symbolic-to-concrete conversion, we would enumerate V by assigning it either of *none*, $true$, $false$ or both.

```
asm recog_trans(rtl, rtl_meta) {
    RTL_CHILD_CONS(rtl, 1, rc1); // RTL accessor function
    RTL_CHILD_CONS(rtl, 2, rc2);
    // ask constraint solver if the constraint is satisfiable
    if (try(rc2 = (plus,_,_))) { // add
      if (try(rc1 = (reg,_))) {
        RTL_CHILD_CONS(rc2, 2, rc22);
        if (try(rc22 = (reg,_))) {
           // End of program path. Record the I/O mapping and
           // terminate execution here.
           recordMapping(rtl, "add r%1, r%2");
        } else if (rc22 = (mem,_)) {
           // In else, we add negation of if condition.
           addCons(rc22 != (reg,_));
           recordMapping(rtl, "add r%1, (*%r2)");
        } else if (rc22 = (const_int,_)) {
           addCons(rc22 != (mem,_));
           recordMapping(rtl, "add r%1, $%2");
        } else {
           addCons(rc22 != (const_int,_));
           recordUnreachablePath(rtl);
        }
        backTrack();
      } else if (try(rc1 = (mem,_))) {
           if (try(rc22 = (reg,_))) {
             addCons(rc1, != (reg,_));
             recordMapping(rtl, "add (*r%1), r%2");
             backTrack();
           } else addCons(rc1 != (mem,_));
      }
      // Inputs lead program execution to failure cases.
      recordUnreachablePath(rtl);
    } else { // mov
      addCons(rc2 != (plus,_,_));
      // transformation for mov
      ...
     }
   }
}
// To save the space, the original function is not shown here.
// But our transformation preserves the original function.
```

Figure 27: **Source-to-source transformation of the simple code generator**

Example. Source-to-source transformation of the simple code generator from the previous section performed using **T** is shown in Figure 27. Notice that the function `translate_rtl` is transformed into `translate_rtl_trans` with `rtl_meta` being passed as an additional parameter. `rtl_meta` is the part of runtime metadata that helps in identifying if a given variable is symbolic or concrete. Calls to RTL accessor functions such as `RTL_CHILD` are transformed so that appropriate constraints are generated in them. For `RTL_CHILD`, a constraint that would be generated would capture the relationship that `rc1` is the first child of `rtl`. Conditions of the **if** statement are transformed into a call to a constraint solver with conditions transformed into constraints. `try()` is a way to ask a constraint solver if the constraints that it has been passed are satisfiable. `try()` behaves exactly like `fork()` – whenever constraints are satisfiable, it creates two program execution flows: one goes into the **if** branch and the other goes into the **else** branch. In the **else** branch, we first add a negation of the **if** condition as a constraint that must hold. `addCons` is a way to add constraints into the constraint system and also to ensure that the constraint addition does not fail. If `addCons` fails, then it terminates the execution flow along that path and returns to the latest branch point visited in the flow. If `addCons` succeeds, as should happen in most cases, the execution flow continues to the next statement. Return statements from functions are transformed into calls to `recordMapping` to record the input to output mapping. This is because the return statements of `translate_rtl_trans` are an ends of program paths. That is why we invoke `backTrack` after calling `recordMapping`. For other functions, `return` is a way to return to their callers. In such cases, we do not call `recordMapping` and `backTrack`, but instead set the metadata of the return value and return the actual return value. In the example, `translate_rtl_trans` is a top-level function and the transformation generates `recordMapping` and `backTrack` for it. All unreachable program points are transformed into a call to `recordUnreachablePath`, which records the information that the particular program point is unreachable, and then terminates the current execution flow. Execution of this transformed program, which amounts to concolic execution of the simple code generator, yields the mapping table in Figure 20.

4.2.4 C language features and challenges for `EISSEC`

C is a very well-known language for the power it gives to the programmers by having features like pointers, typecasting, etc. All such features pose challenges for `EISSEC`. In this section, we discuss the challenges of performing symbolic execution of GCC's code generator.

Pointers. Pointers in C introduce some complications for our symbolic execution system. This is because, unlike C, in constraint programming languages, there is no concept of a reference to an object. Moreover, treating program variables of pointer type as variables of basic types and assigning them symbolic variables leads to further complications. C allows pointers to be used in dereferencing as well as arithmetic. To support both semantics of pointer types, the type system of our constraint solver must allow all unsafe type casts that are allowed in C. This can lead to runtime inconsistencies. Moreover, dereferencing of symbolic pointers demands assigning all possible valid memory addresses to the symbolic pointer (i.e., symbolic-to-concrete conversion) and then performing the dereference on the concrete pointer. Given the large number of concrete memory addresses, enumeration of symbolic pointers is too inefficient and introduces considerable complications in the system.

To avoid runtime inconsistencies altogether, our system supports pointers in very limited contexts. Specifically, pointers of `rtx` type are allowed to be symbolic in our constraint system. Additionally, pointers to compound types, such as arrays or structures, that are used for accessing field offsets which are compile-time constant expressions are allowed to be symbolic. Pointers of all other types are untransformed, and hence treated as concrete. Fortunately, GCC's code generator hardly uses pointers, and in cases where it does, it uses pointers of type `rtx`. Additionally, `rtx` pointers are not used in pointer arithmetic and are used in a type-safe manner. As a result, representing `rtx` pointers by a symbolic object, and modeling type-safe dereferences of such pointers is quite straight-forward for `EISSEC`. As mentioned earlier, our source-to-source transformation pass handles pointer dereferences via accessor functions which refer to semantics of the operation and generate the corresponding constraints.

Note that representing `NULL` `rtx` pointers and allowing comparison to other `rtx` pointers demands special treatment. This is because `NULL` is generally defined as `0`, and because `rtx` is a term type in the constraint system, comparison of an integer with a term would fail in the constraint solver. To solve this problem, we define a special `rtx` object (with `rtx` code of -1; valid `rtx` objects have positive values of code) in our constraint system, and treat that object to be semantically equivalent to a `NULL` `rtx` pointer.

To give an example, Figure 28 shows a piece of C code, and the comments after the statements show the corresponding constraints that are generated for such code.

```

/* Definition of rtx type */
 struct rtl {
    enum rtx_code code;
    enum machine_mode mode;
 rtx childs[]; // variable size array
};
typedef struct rtl* rtx;
..

/* actual code */
rtx r = new_rtx(); // R = rtx(C, M, CH) /* Prolog functor */
if (r == NULL) // C == −1? /* Valid RTL has C >= 0. */
 .. /* So NULL rtx has C = −1. */

if (r−>code == REGISTER) // C == REGISTER?
 ..
else if (r−>code == MEMORY) // C == MEMORY?
 ..

r−>mode = 0; // M = 0
..
```

Figure 28: **C Source code with use of `rtx` pointers and constraints generated for it**

Type casting. Type casting introduces runtime inconsistencies for our constraint solver. Comparison of `NULL` with `rtx` pointer is one such example. The fundamental problem is that by faithfully following the source type, the constraint solver assigns types to the symbolic variables. If the source type is changed abruptly, then the constraint solver throws a runtime exception about the type change of a symbolic variable. Our constraint solver is written in Prolog and supports type-safe casting and, fortunately, we found that GCC's code generator conforms to the type-safe use of C types as well. We therefore do not have to handle type casting specially.

Unions. GCC's code generator frequently uses unions. To model C unions in our type system, we treat them as tagged unions (i.e., same as structures in C). In other words, the semantics of sharing of C's unions is not modeled in our constraint system. This is because we expect the code generator to use unions in a type-safe manner. In our experiments, we found that GCC's code generator meets our expectation.

Other C features. Converting C's bounded-range arithmetic into our constraint system's pure integer constraints is sound only when there are no overflows. If an overflow reflects a programming bug or, more commonly, an assumption on legitimate inputs, its consequences are not serious — our transformation function simply does not capture input ranges that lead to overflow. But if the overflow is intended and expected, then it poses a problem. However, we found that this is rare in GCC's code generator. Nonetheless, we found one case where the overflow was intended, and our constraint system generated incorrect constraints in such case. To solve this problem, we transform the code automatically so that our constraint system generates correct constraints.

The code below shows how GCC defines a macro `IN_RANGE`. The macro returns true if `VALUE` is within the range of `[LOWER, UPPER]`.

```
IN_RANGE(VALUE, LOWER, UPPER) =
   ((unsigned int) (VALUE) - (unsigned int) (LOWER) <=
    (unsigned int) (UPPER) - (unsigned int) (LOWER))
```

GCC relies on C's conversion of a negative result into a positive, and thus it does not need to check explicitly for a case when `VALUE` is in the range `[0, LOWER]`. Unfortunately, when we have an `IN_RANGE` check as a condition on *if* statement, our system negates this check in *else* branch incorrectly. Specifically, the negation is defined as follows:

```
     ((unsigned int) (VALUE) - (unsigned int) (LOWER) >
   (unsigned int) (UPPER) - (unsigned int) (LOWER))
```

This check ignores the possible values from `[0, LOWER]`, which could satisfy the negation. To solve this problem, arithmetic expressions in C which could potentially trigger overflows are transformed to use the semantics of unbounded precision. Specifically, for the source statement `a = b + c`, where `a`, `b` and `c` are unsigned 32-bit integers, we transform them to:

```
a = (b + c) > MAXUNSIGNED ? a = b + c - MAXUNSIGNED - 1 :  b + c
```

where `MAXUNSIGNED` is `(unsigned)(-1)`. Such explicit modeling of unbounded range arithmetic helps address the challenge for `EISSEC`.

4.3 Implementation

EISSEC's implementation is divided into multiple components.

4.3.1 Source-to-source transformation

Source-to-source transformation is implemented as a plugin to CIL [66], a popular open-source source-to-source transformation system. The implementation of the plugin consists of around 2600 lines of OCaml code. The plugin utilizes some of the CIL features such as a simplification pass to generate 3-address code from C code to simplify the transformation. The plugin also handles other challenges such as avoiding the transformations of system code included from header files. This is done by getting a list of directories which are included in the compiler's include search path. We treat files included from any of those directories as a header files and avoid their transformation. The transformer needs support code (written in C) to implement various functions such as `addOpCons` etc. This support code is approximately around 7200 lines of C code. It also includes approximately 3000 lines of C code for RTL accessor functions.

4.3.2 Dynamic single assignment

One challenge to consider while designing a symbolic execution system is how to provide the value semantics demanded by symbolic execution. To be precise, value semantics means that objects are immutable (which is the case, as symbolic variables are immutable.) On the other hand, the C language uses reference semantics where the value of a variable can be updated multiple times. Note that concrete execution updates variables in memory, whereas symbolic execution generates constraints on values in memory. In particular, a constraint on the value of a variable holds only until the variable is assigned again. To capture these value semantics, the symbolic execution system treats each assignment of a variable as if it is an assignment to a distinct symbolic variable.

The approach of using variable's runtime address to generate a unique symbolic name will not work in case of value semantics. We address this challenge by relying on the notion of *dynamic single assignment* [92]. Specifically, we map every symbolic variable to a unique integer (we call it *generation count*, and it is used to generate a symbolic name), and then reassign the variable to a new generation count in case of multiple assignments to the same variable. Similar to SSA (static single assignment), all new variable references will use the most recent generation count. A stack of generation counts is used to keep track of all generation counts. Use of a stack also integrates well with the variable scoping rules in C. Additionally, the map between locations and generation counts for all local variables of a function is built at the function entry and destroyed on function exit.

4.3.3 Undo records

Whenever a symbolic process reaches a branch point and the branch condition is satisfiable, the symbolic process needs to fork its execution into two processes and follow both paths. The ability to follow both paths is the key difference between symbolic and concrete execution which helps symbolic execution achieve better coverage than concrete execution.

There are two key criteria that must be satisfied in such forked symbolic execution: (1) the symbolic state of both symbolic processes after forking should only differ in terms of the branch condition (in the symbolic process that follows the `if` branch, the branch condition must be true, while in the one that follows the `else` branch, it must be false), otherwise, the rest of the symbolic state must be same, and (2) symbolic states of both processes must be completely isolated from each other (by using copy-on-write, they could share the same state) — changes made by one process to its symbolic state must not be visible in the other symbolic process.

Naive `fork()` based solution. The two criteria from the forked symbolic execution fit perfectly in the context of Unix-style `fork()`. Naturally, the use of `fork()` to implement forked symbolic execution is a very logical solution. The EXE [21] symbolic execution system follows this intuition. Unfortunately, the use of `fork()` to implement the notion of forked symbolic execution leads to serious performance degradation. Our experiments revealed that this is not efficient solution (this also falls in line with the view presented in KLEE [19], the follow up work to EXE.) Specifically, the overhead of `fork()` is considerable with respect to the time spent by the forked symbolic process before it is terminated. This is because most of the symbolic processes correspond to error conditions in the code generator, and in such cases these paths are terminated immediately.

Undo records. Given that a `fork()` based solution proved to be inefficient, we decided to develop our own solution based on logging and rollback to satisfy the criteria of forked symbolic execution. Intuitively, the solution schedules the symbolic process for the `if` branch first, and it *records* all the changes made to the symbolic state before its termination. Once the symbolic process for the `if` branch has terminated, it *undoes* all the changes made by that process (to the symbolic and concrete states), and passes that state as a starting state to the symbolic process for the `else` branch. The overall semantics of our solution are same as `fork`() with one important difference: the solution inherently prohibits parallelism — processes for both branches cannot be scheduled in parallel. We found that, given the performance improvement that it gave, the disadvantage in parallelism can be tolerated. We call our solution *undo records*, because it relies on the notion of maintaining a record of the changes made and eventually undoing them.

The undo record solution intuitively needs to satisfy the following criteria: (1) keep track of all the changes being made to both the symbolic and concrete state, and rollback those changes when the program path has terminated (symbolic process has exited), and (2) transfer control back to the latest branch point whose path is being explored. The first criteria is easier to understand, while the second criteria needs some explanation. Note that a program path will most likely consist of multiple branch points. When such a path is terminated, we need to visit the branch points that were visited in the opposite order. These semantics naturally specify that one needs to maintain a stack of branch points.

A naive solution to implement undo records is to snapshot the symbolic process at a branch point and to apply that snapshot when the program path terminates. Because the symbolic metadata is also maintained as a part of symbolic state, such approach can work. Unfortunately, it is too inefficient. Specifically, the snapshot size could be considerable, especially on a 64-bit machine. The drawback of this approach is that it ignores the fact that most program memory is not writable (specifically, program text and read-only data), and thus we do not need to take a snapshot of it. Additionally, with branch points being visited frequently, there would be only a few changes to the states. Thus, keeping track of them should not be inefficient. This observation inspired our undo record solution which uses a combination of snapshotting and recording techniques.

Recording symbolic state changes. All changes made to symbolic state, such as addition, deletion, and modification of symbolic variables, are recorded before the change is applied. Every symbolic state change record consists of the symbolic variable that is being changed, the type of the change, and the value of the symbolic variable before modification. All changes made to the symbolic metadata are also tracked. Thus, dynamic single assignment effects are also undone in the end.

The changes are undone by performing the inverse of the action logged in the record. For instance, if a symbolic variable is deleted as per the record, then we create a variable. In order to undo constraints sent to the constraint solver, we do not need to send it the inverse of the constraints. This is because the constraint solver supports the branching logic for symbolic processes, and recovers its state to the last branch point whenever a path has terminated (backtracking in Prolog).

Recording concrete state changes. In addition to symbolic state changes, all changes made to the concrete state of the symbolic process need to be tracked as well. Concrete state changes include changes made to program registers and memory. Specifically, all changes made to writable (`W`) memory such as stack, heap, and writeable global data need to be tracked and rolled back. Our solution for these is as below:

- **Program registers**

  To handle registers, we use a snapshot technique. Specifically, we use `setjmp` + `longjmp`, where we call `setjmp` at a branch point and store the `jmp_buf` for every branch point. In the end, when the path is terminated, we call `longjmp` using the `jmp_buf` of the latest branch point on the stack. This helps us to take control back to the latest branch point on the stack and to recover the register state correctly.

- **Heap and global memory**

  Unfortunately, `setjmp` + `longjmp` do not recover the state of the program memory. We developed an additional solution for the memory. Our high-level approach to handle memory is to track all changes being made to the memory contents. Thus, we instrument every program statement and record the memory locations that are being modified in the statement, the type of change being made, the size of the access being made, and the contents of the memory locations before the change.

  This solution works in most cases, but `free()` function causes an interesting complication. The problem is that we cannot use `malloc()` to undo effect of `free()` as this does not guarantee that the memory allocator would allocate the memory at the same address that it allocated before `free()`. To solve this problem, we override the `free()` function, and define it as an empty function (that does nothing). Fortunately, GCC's code generator does not perform heap allocations at all, and thus this approach does not lead to any issues because of the memory leaks.

- **Stack**

Unfortunately, the instrumentation-based approach used for the heap and global memory does not work for the stack. This is because stack memory can be updated in ways which are not visible in the source code. Specifically, function calls and returns will modify activation records, and at the source code level, these changes are not visible. Additionally, stack updates are much more frequent as compared to heap or global memory updates. An instrumentation-based solution would prove inefficient. We use a snapshot-based solution for stack. (Accesses made to the stack are distinguished from accesses made to other memory by relying on top and bottom values of stack.)

In more detail, at the branch point, we take a snapshot of the full program stack (from stack top to stack bottom) and store it in the branch point. (We need to take a full snapshot because a path can terminate in the bottom-most activation record, and thus when we go back to such a branch point, all of the activation records would have been deleted. In such case, we would need to recreate all those activation records.) When the program path is terminated, we refer to the saved stack of the latest branch point and use it to overwrite the current stack. The fact that the program stack is generally small (less than 8KB on Linux) makes this approach time-efficient (by trading some space).

We opted for this inefficient, but simple, solution because we found that it works fine in our context. Moreover, a more efficient approach that we designed was leading to a lot of complications and was making our system unstable. Our efficient approach was trying to reduce the amount of stack space saved in each branch point. The idea that we can snapshot the top part of the stack since the previous snapshot can reduce the memory consumed considerably. However, this approach complicates undoing of stack changes considerably. Specifically, the complications arise because there may not be synchronization between visits to the branch points and the creation or deletion of activation records. These complications were making our system buggy. As a result, we decide to use a simple approach instead of this efficient approach.

Our undo record approach is implemented in around 500 lines of C++ code. The space complexity of our approach is proportional to the length of the program path (in terms of branch points), and is not proportional to the number of program paths. Thus, our system can scale to millions of paths.

4.3.4 Constraint solver

Constraint solver plays an important role in the efficiency of a symbolic execution system. Because the constraint solving time dominates the overall runtime of symbolic execution systems, popular symbolic execution systems such as KLEE [19], CUTE [82], etc, have implemented optimizations to simplify the constraints before sending them to a constraint solver or eliminate them altogether. These systems use SAT/SMT solvers such as STP [38] or Z3 [34] as their constraint solvers. SAT solvers solve the SAT problem, whereas SMT solvers are SAT solvers with the support for higher-order theories such as theory of bit-vectors, theory of arrays, etc. With the support of these theories, SMT solvers allow modeling of high-level language constructs efficiently. For instance, constraints generated from C code that use arrays can be efficiently modeled using SMT solvers with the theory of array. That is why SMT solvers are popular among many symbolic execution systems.

Even though SMT solvers are commonly used constraint solvers among symbolic execution systems, in `EISSEC`, we use a finite-domain constraint solver [90] with *constraint logic programming* (CLP) to model the constraints. The reason why we do not use SMT solvers is because SMT solvers are efficient at producing multiple answers to the same query[18]. Specifically, they are not engineered to provide all solutions [42, 95]. CLP, with the support of a finite domain constraint solver, on the other hand, is based on the idea that all solutions need to be enumerated. In `EISSEC`, we need to produce all answers to a query when we convert a symbolic variable to a concrete variable (we called it *concretization* or "explosion").

The implementation of our constraint solver uses `Swiprolog` [96], a popular Prolog engine, with its support for CLPFD [90]. GCC's code generator only generates finite-domain constraints, so CLPFD is enough for our purpose. The solver is implemented in around 700 lines of Prolog and is supported by 1000 lines of C code. It uses the well-known Prolog concept of backtracking (using `fail.`) and other supported features of `Swiprolog` such as association lists, enumerating all solutions to a query using `labeling`.

Although, we could map most of our requirements from the constraint solver into the predicates of CLPFD or Swiprolog, one problem demanded special treatment. Specifically, we found that neither CLPFD nor `Swiprolog` provides predicate(s) to access the set of constraints imposed on the variables. The output mapping rules generated by `EISSEC` are of the form $A \rightarrow I \mid C$, where A is an assembly instruction which maps to an IR instruction

[18]It is thus quite logical that the symbolic execution systems that are used in the context of bug finding (where generating even one answer to a query is enough) can use SMT solvers.

I under a set of conditions C. C represents conditions on the variables from A and I. To give an example, push $\%\mathtt{X} \to (\mathtt{set}(\mathtt{mem}\ (\mathtt{reg}\ \mathtt{esp}))(\mathtt{reg}\ \mathtt{X}))\ |\ X = eax, ebx, ..$ could be a mapping rule for push instruction with a register as an operand. Note that X is a variable, and since it can only contain the valid register names, there are constraints on it. These constraints would be imposed by GCC's code generator, and they would be present in the Swiprolog constraint store. Unfortunately, we did not find any predicates (to the best of our knowledge) to access the constraint store and retrieve the constraints on all the variables in a mapping rule. To solve this problem, we access the constraints using clpfd_attr and access its propagators via fd_props[19] as Swiprolog stores the constraints using attributed variables [91]. In order to capture a complete set of constraints on all the input and output variables of a mapping rule, we traverse the dependence graph of the variables starting from the output variables and reaching all the input variables. The dependence graph can is an undirected graph where the nodes are variables and the edges are constraints between the variables. The goal of the traversal is to print constraints appearing on all the paths between the output and the input variables.

The constraint solver runs as a separate process than the transformed C code and simply acts as a proxy between transformed C code (which generates constraints) and the Swiprolog Prolog engine (which solves them). The C support code provides APIs to the transformed C code to generate constraints (addCons), sends them to the constraint solver, and also maintains statistics for diagnostic purposes. Additionally, it also supports functions which deal with symbolic processes, such as handling of symbolic fork (try()), enumerating all solutions to a query (getNext()), recording input/output mapping (recordMapping()), etc.

4.4 Optimizations

Symbolic to concrete conversion ("enumeration" as we described earlier) leads to a large number of program paths, more than what we would get if there is only one symbolic process per program path. As the total time taken to extract a code generator function is proportional to the number of program paths, enumeration increases the time needed for the function extraction. As a part of improvement, we found a few optimizations that reduce the effect of enumeration on the number of paths.

To understand how optimizations help in reducing the number of program paths, we

[19]These are not standard predicates in Prolog, but rather are internal ones.

```
 1  /* Some definitions */
 2  enum rtx_code = {REG = 0, MEM, CONST_INT, LABEL};
 3  int num_ops[] = {1, 2, 1, 1};
 4  #define HAS_SINGLE_OPERAND(code) (num_ops[code] == 1);
 5
 6  /* actual code */
 7  int getType(enum rtx_code code) {
 8    if (HAS_SINGLE_OPERAND(code)) {
 9      ...
10      if (code == REG)
11        ...
12      else if (code == MEM)
13        ...
14    } else {
15      ...
16    }
17  }
```

Figure 29: **Sample C source code for concolic execution**

first need to understand the problem in more detail. Consider that we want to execute the code in Figure 29 concolically. The code implements a simple function of accepting the `rtx` code and processing it depending on whether the code corresponds to an RTL term with a single operand or multiple operands (this is captured in `num_ops` which is an array indexed by `code`.)

To perform concolic execution of the above code, we make the input `code` symbolic. Thus, when we start, at line 7, we only have one symbolic process. Assuming that we decided to treat `num_ops` array concretely, the index used to access the array must also be concrete. Then, we have a case of symbolic-to-concrete conversion in the code of the `if` statement (at line 8). Instead of one symbolic process, now we have to enumerate all possible values of `code` when we are indexing into `num_ops`. Given that `num_ops` has four elements, we must generate four concrete processes, each for the element of `rtx_code`. To be precise, in the first concrete process, the value of `code` will be `REG`; in the second one, the value will be `MEM`, and so on. Each of these four processes will now continue with the execution of the remaining statements before the program path has ended. Given that the condition of the `if` statement (at line 8) is satisfied by three of the four processes, the three of them will enter the block of statements beginning at line 9. One process, on the other hand, will enter the `else` block beginning at line 14. Given that every `if` statement can produce two processes for a single input process, the total number of processes that will execute the above code will be (3*2*2+1) = 13. Thus, there will be 13 input to output mapping pairs for above code. On the other hand, if there had been no symbolic-to-concrete conversion, then we would have (1*2*2+1) = 5 symbolic processes.

The above example shows that, if a symbolic state is never required to be converted to concrete state, then we do not need to enumerate. Unfortunately, in concolic execution such ideal case is uncommon. Nonetheless, one can make several interesting observations about the process of enumeration, and reduce the cost of enumeration. Specifically, if we can reduce the number of concrete processes from 3 to 2, or may be 1, at line 8, then, we will have fewer than 13 concrete processes, and we will be able to finish the concolic execution of the C code more quickly. Such observations make the basis of our optimizations, which aim at achieving the ideal case either by deferring the conversion or by reducing the effect of symbolic to concrete conversion, and thus reduce the number of explored program paths.

4.4.1 Using range and set constraints

Notice that the reason the enumeration produced four concrete processes above is because the `num_ops` array contains four elements. Thus, these four processes will each have `code` set to `REG`, `MEM`, `CONST_INT`, `LABEL`, resp. Also notice that the condition `num_ops[code] == 1` is satisfied by three of them, while the remaining one does not satisfy the condition. So instead of having three different processes to represent the case when the number of operands is one, we can have one process and set `code` as `REG` | `CONST_INT` | `LABEL`. The idea here is that, instead of setting `code` to a single value, we set it to a *set* of values. Note that such an optimization is sound because we are capturing the same semantics of the three different processes setting `code` individually.

Fortunately, our constraint solver can support setting a symbolic variable to a set by relying on `swipl`'s support for domain constraints. Specifically, in `swipl`, we represent the above set as `code` = `REG` | `CONST_INT` | `LABEL`. Unfortunately, it is easy to see that, to represent set constraints, we need to enumerate all elements of the set. Fortunately, range constraints make this process easier.

Range constraints allow us to represent a set of elements that form a range. For instance, the set {1, 2, 3, 4} can be represented as a range [1, 4]. `swipl` supports this notion, and we exploit it to represent sets compactly. For instance, the set {`REG`, `CONST_INT`, `LABEL`} can be represented compactly as {`REG`} and [`CONST_INT`, `LABEL`]. Note that in this example, because the range is representing a set of two elements, there is a little reduction in the number of processes. However, a set having hundreds of elements that form a range can lead to significant reduction in terms of the number of processes.

By using a combination of set and range constraints, we can reduce the number of program paths in the above example from 13 to 7 (7 = 2*2*2+1). The above example also represents a common case in GCC's code generator, and thus, by relying on set and range constraints, we are able to reduce the number of program paths considerably. We implement this optimization in array accessor functions, and in case of GCC, some of the arrays contain hundreds of elements. By using range constraints, such elements could be represented using few ranges.

4.4.2 Strength reduction

Another way to avoid symbolic-to-concrete conversion is to avoid operations that index into concrete arrays using symbolic values. Instead, if we can represent the semantics of such operations in a way that avoids enumeration, then we can achieve get more speedup. Strength reduction thus aims at replacing an expensive operation by a semantically-equivalent less expensive operation (strength reduction in standard compiler optimization terminology is same.)

In the context of our example, notice that symbolic-to-concrete conversion is required because we are indexing into a concrete array symbolically. We replace the check in C `num_ops[code] == 1` by a less expensive check such that no enumeration is needed. In this case, we rewrite the check as `code == REG || code == CONST_INT || code == LABEL`, avoiding the enumeration. Note that the set constraints discussed earlier also achieve the same effect, but with strength reduction generation of set constraint is more straight-forward.

The savings offered by this optimization are directly proportional to the number of elements that we must enumerate. In the context of the example, we would have only 5 (i.e., 1*2*2+1) paths instead of 13. We have implemented this optimization by hand-rewriting the expensive operations by semantically-equivalent less expensive ones.

4.4.3 Exploiting hardware-level parallelism

Recall that our *undo record* solution takes away the inherent parallelism in exploring both the *if* and the *else* branches of a symbolic conditional in parallel. We found that such restriction was necessary so that EISSEC does not overwhelm the CPU with too many live symbolic processes. But we also found that `EISSEC`, by default, does not exploit multi-core parallelism commonly found in contemporary desktop and server CPUs. To put this observation to use, we developed a parallel execution system within `EISSEC`.

Our parallel execution system relies on the following observations: (1) code generator's RTL-to-assembly translation process operates on a decision tree, and (2) that subtrees of the root node represent considerable portions (divided equally) of the code generator source code. Thus, if we schedule *N* different subtrees of the root node in parallel, then we get speedup of *N* (ideally). Because all subtrees are representing a similar amount of symbolic execution work, parallelism should yield maximum benefit. This optimization can thus be seen as a restricted form of parallelism: we are parallelizing the top-level subtrees of the source code only, but within each tree, there is no parallelism. Because *N* is not considerably more than the number of CPU cores, this optimization does not overwhelm the CPU.

In the context of the example, we schedule `if` and `else` branches of the top-level `if` statement (at line 8) in parallel. If both branches are doing a similar amount of work, this yields parallelism of two.

4.5 Evaluation

The evaluation of `EISSEC` is divided into three parts. In the first part, we evaluate the performance of `EISSEC` in extracting the x86 semantic model. In this part, we also see the impact of different optimizations on `EISSEC` performance. In the second part, we analyze the extracted model in terms of soundness, completeness, and ability to support binaries produced by other compilers. More importantly, we compare the x86 models extracted from `EISSEC` with those of `LISC`. Because `EISSEC` allows us to cover all possible assembly instructions produced by the code generator, testing `LISC` model using `EISSEC` model is an interesting experiment. In the last part, we analyze the effort involved in applying `EISSEC` to extract a model for other architecture (AVR).

The evaluations in this section were performed on an Intel Core i7-3517U ultrabook running at 1.90GHz. The machine is equipped with 4GB of RAM and 4GB of swap space, and runs Linux kernel-3.13.0-53 and 32-bit Ubuntu-14.04 desktop OS. We performed symbolic execution of the x86 code generator from `gcc-4.6.4` and the AVR code generator from `avr-gcc-v4.8.2`.

4.5.1 Model extraction performance

GCC's decision tree. We first discuss the RTL-to-assembly decision tree which is the focus of our symbolic execution. With reference to this tree, we also describe the terms which we will use in our evaluation.

We performed symbolic execution of the GCC-4.6.4 x86 code generator which contains 2244 RTL-to-assembly mapping rules for x86. These rules cover all of the x86 advanced instruction sets such as SSE, AVX, etc. GCC's decision tree, which is one large C function, implements this using approximately 120,000 lines of C code.

The decision tree accepts RTL instruction as input and outputs assembly instruction(s) as output. The nodes in the decision tree represent specific constraints on the fields of RTL, and the leaves represent the assembly instructions. Thus the nodes along the path from the root to a leaf represent a set of constraints that are satisfied by the input RTL that maps to the assembly instruction at the leaf node. (As a side note, GCC generates this decision tree from an architecture-specific machine description file when GCC is compiled for that architecture.) A sample C code for the code generator and the decision tree produced by GCC for it is shown in Figure 30. Two things need to be mentioned: instead of showing the conditions at the node, we have shown them on the edges, and gray nodes represent error conditions.

With reference to this decision tree, we describe our terms below.

- *Positive paths and failure paths.*

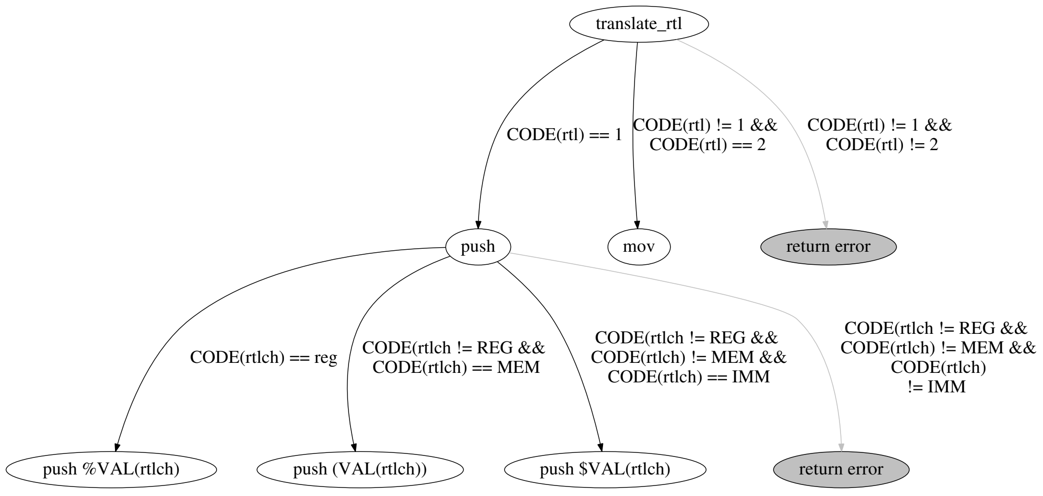
  A path in the decision tree is a path from the root of the tree to any leaf node. Because the input to the tree is an RTL instruction and the output is the corresponding assembly instruction, a path in the decision tree corresponds to a single RTL-to-assembly mapping.

  A positive path in the decision tree corresponds to the case of a valid RTL to assembly mapping. By valid RTL, we mean the RTL that is accepted by the decision tree as correct RTL, and maps to a valid assembly instruction supported by the target architecture.

  Failure paths, on the other hand, represent exactly the opposite of positive paths. To be precise, failure paths represent cases where the input RTL to the code generator is invalid and there will not be assembly instruction(s) generated by the code generator if this RTL is given to the code generator as input. In other words, failure paths never end with the production of a valid RTL; they end with producing an error code.

```
assembly translate_rtl(rtl) {
 if (CODE(rtl) == 1) {
  /* push */
  rtlch = CHILD(rtl);
  if (CODE(rtlch) == REG)
   return "push %VAL(rtlch)";
  elif (CODE(rtlch == MEM))
   return "push (VAL(rtlch))";
  elif (CODE(rtlch == IMM)
   return "push $VAL(rtlch)";
  else
   // Unsupported operand
   return error;
 } else if (CODE(rtl) == 2) {
   /* mov */
   /* operands of mov */
 } else
   // Unsupported instruction
   return error;
}
```

(a) **Sample C code for the decision tree**


translate_rtl
CODE(rtl) == 1
CODE(rtl) != 1 && CODE(rtl) == 2
CODE(rtl) != 1 && CODE(rtl) != 2
push
mov
return error
CODE(rtlch) == reg
CODE(rtlch != REG && CODE(rtlch) == MEM
CODE(rtlch != REG && CODE(rtlch) != MEM && CODE(rtlch) == IMM
CODE(rtlch != REG && CODE(rtlch) != MEM && CODE(rtlch) != IMM
push %VAL(rtlch)
push (VAL(rtlch))
push $VAL(rtlch)
return error

(b) **Decision tree**

Figure 30: **Sample C code and its decision tree**

In the example figure, "`translate_rtl -> push -> return error`" is a failure path. While, "`translate_rtl -> push -> push %VAL(rtlch)`" is a positive path.

- *Length of a path*

  We measure the length of a path in the decision tree by counting the number of nodes in the decision tree along the path. A node in the our decision tree represents a branching point where a condition would decide where execution would continue. Thus, the length of a path in decision tree is the number of conditionals along that path. Because a conjunction of such conditions along a path (i.e., path condition *PC* as it is called in symbolic execution systems) represents the condition on the input, the length of a path is the same as the number of clauses in *PC*.

  For instance, length of "`translate_rtl -> push -> push %VAL(rtlch)`" is 2. (It is not 3 because we count only the interior nodes on the path and not the leaves.)

- *Coverage*

  Coverage is defined as the percentage of leaf nodes visited by `EISSEC` (at least once) out of all the leaf nodes in the decision tree. There exist other coverage criteria such as the number of lines of source code covered, number of branches covered, etc. As our goal is to obtain the complete semantics model for the target architecture, we found our definition of coverage effective in evaluating `EISSEC`. Moreover, unlike test case generation or bug finding, where most symbolic execution systems are used, our problem demands covering all program paths. So coverage criteria based on source code lines do not serve our purpose.

  For instance, in the example tree, if we produced "`push %VAL(rtlch)`" then we have covered 20% of the leaves (note that the "`mov`" node does not have a child.)

Performance of extracting complete x86 model. We evaluate the performance of `EISSEC` in traversing all paths and extracting the complete x86 model.

Detailed results on the evaluation of the symbolic execution of the complete x86 code generator are shown in Figure 31. The first thing to note for these numbers is that they represent the baseline for further comparison. We did not enable any optimizations during this experiment. Thus, in the worst case, we take 13 CPU days to extract the complete semantics model. We make the following observations:

Total Time (in Days)	**Number of Paths (in M)**	**Failure Paths (in M)**	**Positive Paths (in M)**	**Path Length**	**Number of Constraints (in B)**	**Coverage (in %)**	**Virtual Memory (in MB)**
0.00	0	0	0	0	0.00	0	17
0.51	23	18.1	4.5	123	0.26	13	20
0.89	34	28.4	5.8	119	0.48	15	19.7
1.27	46	38.6	7.1	135	0.70	17	20.4
1.65	57	49.9	7.3	146	1.00	17	21
2.04	69	61.2	7.6	131	1.35	19	20.2
3.17	103	94.9	8.5	26	1.58	21	17.6
4.88	137	127.6	9.0	82	1.95	37	18.9
6.09	167	156.3	11.2	92	2.46	49	19.2
7.31	241	223.3	17.4	37	3.16	66	18.1
8.53	266	248.0	17.6	68	3.45	69	19
9.75	287	268.5	18.1	71	3.65	70	18.7
10.42	307	287.5	19.2	74	3.83	79	18.7
11.55	340	316.9	23.5	35	3.91	87	17.8
12.22	360	336.0	24.3	42	4.49	91	18
13.11	386	360.2	25.6	1	5.96	100	17.3

Figure 31: `EISSEC` **performance (without optimizations) to extract complete x86 model**

- *Total paths, positive paths, and failure paths.*

  There are a large number of total paths at various times during the experiment because there exist many failure paths. Our analysis revealed that between 80–94% (on different days) of the total paths traversed by the system ended as failure paths. The reason behind the large number of failure paths is that a positive path is a conjunction of constraints imposed by the nodes along that path. If a conjunction of constraints represent a positive path, then the negation of conjunction can lead to failure paths. A negation of the conjunction with N conditions can be represented in N different ways. (It is not $2^N - 1$ because, in a decision tree, N conditions are checked in a particular sequence. A negation of such a conjunction can be done by negating each condition one by one.) In other words, for a single positive path with N internal nodes along the path, there could be a maximum of N failure paths. In the experiment, the highest path length that we obtained was 146. This means that, in the worst case, there would be 146 failure paths for a single positive path. Fortunately, there were around 4X to 15X failure paths for a single positive path.

Paths per assembly instruction. That just completes half the answer. The question of why there are so many positive paths is yet to be answered. There are two important reasons why there are so many positive paths: (1) a mapping rule produced by `EISSEC` contains assembly instruction along with some operand combination, and (2) the effect of symbolic-to-concrete conversion.

To understand the first reason, realize that the output of a code generator is a complete assembly instruction along with operand combinations, for instance, `push %eax` as against just `push` as a mnemonic. Because GCC prints operands differently, it is intuitive that there would be multiple mapping rules for a single assembly instruction. For instance, for the `push` assembly instruction which accepts different types of operands such as register, memory, immediate operand, etc, there would be multiple different rules. Specifically, these different rules correspond to printing of operands in the following different syntactic notation (in AT&T syntax): `%reg`, `$immediate`, `(%reg)`, `(%reg,%reg)`, `(%reg,%reg,const)`, and `const(%reg,%reg,const)`, etc. Thus, for a single mnemonic `push`, there are multiple mapping rules (positive paths), each corresponding to a particular operand combination. Note that, for mnemonics with multiple operands (e.g., `add`), the possible number of operand combinations are even greater.

While analyzing the number of positive paths, we came across the effects of symbolic-to-concrete conversion. To explain with an example, for a single mapping rule of `push %reg`, we found that there were multiple mapping rules corresponding to different ways in which x86 registers are mapped in RTL. For instance, all 16-bit parts of 32-bit general purpose registers correspond to RTL with one of its fields (mode) set to `HImode`, where all 32-bit registers correspond to RTL with that field set to `SImode`. It is thus possible to see that, by generating multiple mapping rules corresponding to `push %reg`, our symbolic execution system leads to enumeration of different values of the mode field in RTL. Moreover, we also found that, because of symbolic-to-concrete conversion for other fields of RTL (other than mode) which are not related to the size of registers, our system can produce duplicate mapping rules. For instance, it may be possible that for 32-bit registers, our system produced eight different mapping rules (corresponding to eight 32-bit registers in x86) instead of generating a single rule for all 32-bit registers.

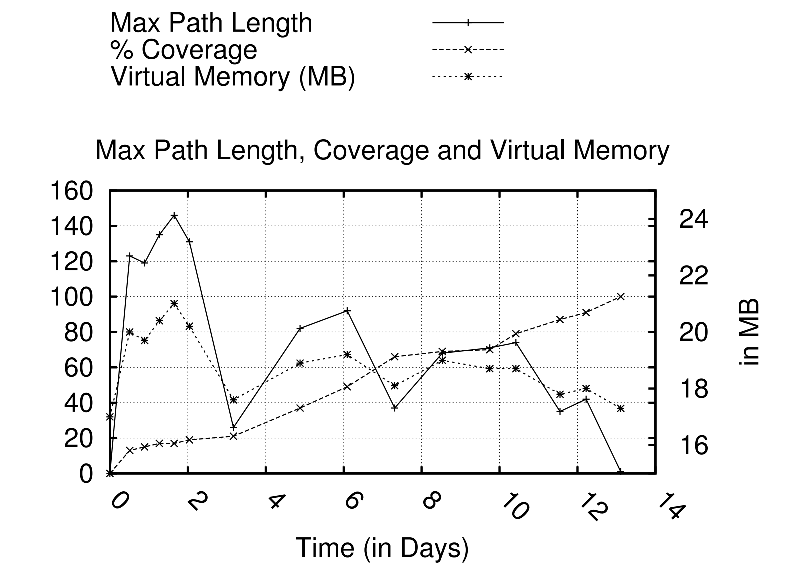
In relation to the number of assembly instructions, the number of positive paths is a large number. We address this issue is two ways. First, we implement an optimization of reducing the impact of symbolic-to-concrete conversion. Second, instead of keeping mapping rules (positive paths) as a list, we build an automata (transducer) from them. Automata helps in finding the common parts across terms and thus reduces the model size considerably.

- *Length of a path, percentage coverage, and memory consumed*

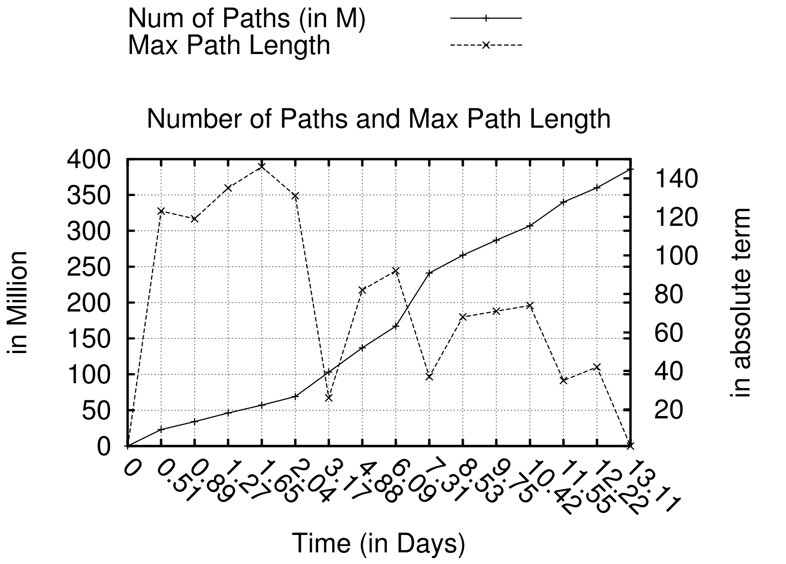
  As expected, the percentage coverage of the x86 code generator reaches 100%. However, it is interesting to observe that, during some parts of the experiment, our system obtained more coverage than some others. Specifically, within the initial 2 days, `EISSEC` covered 20% of the total paths, while in the next 3 days, it covered only around 17% more. This is because of the difference between maximum length of the path in the part of the code generator that was being explored at these two times. Specifically, in the first 2 days, the maximum path length was 146, while for the next 3 days, it was 82. A path with a longer length most likely has a greater number of leaves than a path with a shorter length. The effect of path length on the coverage is also visible towards the end of the experiment, where the improvement in coverage reduced considerably as our system was exploring a relatively small subtree of the decision tree.

  It is also interesting to note that the length of a path reduced drastically (from 131 to 26) on the third day. We found that this was because our system finished exploration of a subtree of the decision tree around that time, and was about to enter the next subtree. Recall that we execute both branches of a condition sequentially. Consequently, all children of the root node are visited in DFS fashion. Such sudden dips in the path length are also visible in Figure 32a. We found that these dips correspond to finishing the exploration of a large subtree and moving to the next one. This is also visible in Figure 32b in which the slope of the lines corresponding to paths is high during the first 7 and half hours of the experiment, and it reduces towards the end. Although we do not parallelize branches of a conditional, we do exploit hardware level parallelism by exploring subtrees of the root node in parallel.

  It is also important to note the amount of virtual memory consumed by our system (including the constraint solver) is small. Specifically, the memory consumed is proportional to the maximum path length. This is exactly the benefit of undo record feature in our system. This is because at every node (branch point) along a path, we



(a) **Relation between maximum path length, coverage and virtual memory**



(b) **Relation between number of paths and the maximum length of the path**

Figure 32: **Relation between number of paths, coverage and virtual memory**

need to store the machine state (CPU and memory) in order to explore another child of that node. Given that the complete exploration of a code generator is an all-path problem, we believe that our decision of using sequential execution kept the demand on memory low. As a consequence, we could complete the exploration of the full code generator on a machine with 4GB of RAM.

Parameters	**Without Optimizations**	**+ Range Constraints**	**+ Strength Reduction**	**+ Top-level Parallelism**
CPU time (in Day)	13.11	9.20	9.04	6.64
Number of Paths (in M)	386.00	270.00	263.88	263.88
Failure Paths (in M)	360.20	259.10	257.56	257.56
Positive Paths (in M)	25.60	10.90	6.32	6.32
Number of Constraints (in B)	5.96	3.71	3.45	3.45

Figure 33: `EISSEC` **performance with optimizations to extract complete x86 model**

`EISSEC` **performance with optimizations.** To understand the effectiveness of different optimizations, we systematically enabled them one by one, and measured the key performance parameters. The results are summarized in Figure 33. The best time to extract the complete x86 model was around 6 days and 14 hours. In the figure, optimizations are enabled in sequence, going from left to right. For any column, all optimizations in the columns on its left are enabled. Note that we do not include other parameters such as coverage, maximum path length, virtual memory, etc, because they will be the same irrespective of the optimizations. We make following observations from the figure.

- *Range constraints*

  Among all three optimizations, range constraints yield the maximum effectiveness (around 42% reduction from the base case) in terms of the reduction in CPU time. We expected greater reduction in runtime from this optimization. Our expectations were high because cases where symbolic-to-concrete conversion occurs operate on sets of 120 to 140 elements. To be precise, one such case was using an RTL code (which is symbolic) to index into an array of 128 elements, and was checking if the value in an array satisfies a condition. In the worst case, symbolic-to-concrete conversion would produce 128 concrete processes for a single symbolic process. By relying on the condition, we were able to represent the semantics using 24 ranges, thus reducing the symbolic-to-concrete conversion factor from 128 to 24. With a factor of 5 reduction, we were expecting to see similar benefit in the result of this optimization. However, our analysis pointed out several symbolic-to-concrete conversion places as culprit. Specifically, to realize the full effect of this optimization, all the places along a path which use symbolic-to-concrete conversion should use this optimization. We found that we are not able to obtain a factor of 5 reduction at some of those places, and consequently, we do not realize the full effect of this optimization.

- *Strength reduction*

  Strength reduction followed similar ideas to range constraint, and with range constraint optimization enabled, we were not expecting significant savings from this optimization. This proved to be true as we got around 1% reduction in CPU time. We still see some reduction in the number of positive paths, because some syntactic structures reduce the effectiveness of range constraints. This is because, in such cases, our system cannot realize that it can use range constraints instead of symbolic-to-concrete conversion. By rewriting these syntactic constructs, we enable strength reduction which in turn enables range constraints.

- *Parallelism*

  We explored all 6 subtrees of the root of the decision tree in parallel. We created one process per subtree and also added appropriate constraints from the previous trees in succeeding trees. To be precise, note that the root of these subtrees actually represents different conditions on the input. When sequentially executing these conditions, the second condition always means that the first condition is not satisfied. In order to capture these semantics correctly, we must negate the conditions at the root of these subtrees and add them to the constraint set of the next subtree.

  With a factor of 6 parallelism, we observed only 36% improvement in CPU time. We found significant differences in the heights of the subtrees. To be precise, four of six subtrees are of same height, which is around one third of the height of the remaining two. That is why the four of the six trees were explored very quickly as compared to the last one. The exploration of the last two trees dominated the runtime. Note that, in the case of this optimization, all other parameters have the same value as that of strength reduction, because these parameters are independent of sequential or parallel execution. In other words, parallel execution only offers the advantage of utilizing more cores to complete the exploration of the decision tree.

  A better approach to parallelism could be explored in the future. Specifically, instead of restricting parallelism only among top-level subtrees, we could employ parallelism at every branch point depending on CPU utilization. To be precise, at every branch point, if the CPU is not fully utilized, then we could explore both branches of a conditional in parallel rather than exploring them sequentially. If the CPU is fully occupied, then we could switch to sequentially exploring the branches of a conditional. We believe that such an approach could yield maximum improvement under the constraint of a given CPU limit.

4.5.2 Soundness, completeness, and model size

Soundness. To check the soundness of the extracted x86 model, we followed the semantic equivalence test from `LISC`. This soundness test did not find any soundness violations other than the two violations (of `call` and `jmp`) reported in the `LISC` evaluation.

An approach of how we obtained assembly and IR pairs for this test needs an explanation. Recall that, in `LISC`, we used binaries from the Ubuntu-14.04 distribution as the test data. In case of `EISSEC`, we followed a different approach. We used `EISSEC` mapping rules themselves to produce concrete assembly and IR pairs. We could do this because the mapping rules, in case of `EISSEC`, precisely capture constraints on input operands that satisfy those rules. By using the constraint solver to find the satisfying assignments for input operands, we can concretize assembly and IR pairs. Being able to produce a set of all valid operand values is one of the advantages of symbolic execution. In contrast, the compiled binaries may contain only a commonly used subset of all operand combinations.

Next, we used the x86 model obtained using `EISSEC` to test the soundness of the x86 model obtained using `LISC`. This is interesting to evaluate because generalizations in `LISC` can lead to soundness issues. Our approach for this evaluation consisted of three steps: (1) generate a set of concrete assembly instructions by using the `EISSEC` model, (2) translate those instructions to IR using `LISC`, and (3) run the semantic equivalence test on the original assembly instruction and the translated IR. With this approach, we did not find any soundness issues in the `LISC` model.

Given that `LISC` cannot infer constraints on the input and output operands precisely, it often can accept operands which it should not accept. For instance, just by relying on the code generator logs, `LISC` cannot infer that `xmm` registers cannot be used as the base register in memory operands. Thus, it would accept instruction such as "`mov %eax,(%xmm0)`". Note that `LISC` accepts this instruction because it would have seen instructions such as "`mov %eax,(%ebx)`" which it generalizes to allow any register combinations. To find out how many such instances are accepted by `LISC`, we used the `EISSEC` extracted model to generate invalid assembly instructions. This was done by negating the constraints on the input and output operands, and generating the assembly instructions that satisfy those negated constraints. We found a number of cases where `LISC` accepted invalid assembly instructions. But note that we did not come across such cases while evaluating `LISC` on Ubuntu-14.04 binaries, because GCC would never produce these invalid assembly instructions. That is why acceptance of invalid assembly instructions does not represent a serious practical problem with LISC.

Completeness. Measuring the completeness of a model extracted using EISSEC is a challenge in itself. Because symbolic execution is expected to produce a "complete" model that is used by the code generator, to measure the completeness of such a model, we must compare it against the architecture manual. But, given that such a comparison is practically difficult to automate (recall that this is the one of the reasons why we decided to use the code generator for model extraction to start with!), we followed the same completeness test as we did in LISC. Specifically, we performed two tests: (1) comparison against all binaries on Ubuntu-14.04, and (2) comparison against LLVM-produced binaries. The first test tells us if we covered all instructions that the code generator could produce, while the second test tells us exactly which instructions are not supported by the particular version of GCC.

On the Ubuntu binary test, we found that we were able to translate 99.66% of the assembly instructions (without any manual effort). In comparison, the LISC x86 model was able to translate 99.49% of the instructions. EISSEC could translate 0.17% more instructions because it did not face the issue of missing operand combinations. It also found that out of 49 missing instructions reported in LISC, 2 were actually not missing (`rcl` and `rcr`). Nonetheless, remaining 47 instructions were not found in the model extracted by EISSEC.

We then evaluated the EISSEC model against LLVM-produced binaries from LISC. We were able to translate 97.45% of the instructions from those binaries (vs 96.03 with LISC). We found that the slight improvement was again contributed by the improvement in missing operand combinations. Unfortunately, we could not translate assembly instructions that had missing mnemonics in LISC the LLVM test. To be precise, those mnemonics were also missing from the EISSEC model.

We then used the EISSEC x86 model to test the completeness of the LISC x86 model. This evaluation is interesting because Ubuntu binaries may not cover all possible operand combinations for the assembly instructions. The EISSEC model, on the other hand, can produce such combinations. In order to perform such evaluation, we used the EISSEC model to generate a list of assembly instructions along with all possible operand combinations known to GCC. Out of around 7000 operand combinations of assembly instructions, we found that around 260 were not accepted by LISC. One such example is `push $immediate`. We found that GCC-produced code generator logs did not contain such instructions. Another case was of a memory operand written in the form of base, index and scale (e.g., `8(%eax,%ebx)`). We found that GCC logs did not contain this operand for many instructions.

Model size. We used the `LISC` transducer construction algorithm on the set of mapping rules obtained by `EISSEC`. Note that the number of mapping rules in case of `EISSEC` is greater (6.32M) than the number of parameterized rules in the case of `LISC` (370K). This was because of the effect of symbolic-to-concrete conversion and also because `EISSEC` mapping rules contain the enumerated mode field. Specifically, in `EISSEC`, the mode of an RTL is explicitly encoded in the rule. Because modes of various registers are different (e.g., the mode of `eax` is different than the mode of `ax`), there are more mapping rules in `EISSEC` than in `LISC`. Note that `LISC` does not treat RTL mode as a parameter. It instead infers the mode of the register from its name.

Even though the mode field for the same assembly instruction might have different values in the `EISSEC` mapping rules, rest of the fields were same, so the transducer construction algorithm exploited the common parts effectively. Nevertheless, in order to handle different values of the mode field, the constructed transducer has more states and edges than `LISC`. Specifically, in comparison to the `LISC` transducer with 133K edges, the `EISSEC` transducer has 797K edges. Construction of the transducer took around 80 minutes.

We believe that these results deliver on the promise of `EISSEC`. Specifically, by being able to cover all paths inside the code generator, we were expecting to get a model that is more complete than `LISC`. And, we believe that the results match our expectations. On the other hand, it is also interesting to find that the model extracted by `LISC` was not too incomplete as compared to that of `EISSEC`. This implies that, by compiling a number of packages with various compilation options, we could explore almost all of the code generator.

4.5.3 Model extraction for AVR

After extracting an x86 model using symbolic execution, we followed the same steps to extract an AVR model from the code generator of `avr-gcc-v4.8.2`.

Complexity of AVR code generator. The AVR architecture has 76 mnemonics, and the code generator used for the symbolic execution contained 98 RTL-to-assembly mapping pairs. The decision tree used by the code generator takes approximately 12K lines of C code.

Total Time (in mins)	**Number of Paths (in K)**	**Failure Paths (in K)**	**Positive Paths (in K)**	**Path Length**	**Number of Constraints (in M)**	**Coverage (in %)**	**Virtual Memory (in MB)**
0	0	0	0	0	0.00	0	3.2
20	11.2	9.1	2.1	17	2.8	13	6
40	36.2	29.2	7	29	5.6	49	8.3
60	56.3	48.5	7.8	21	8.3	71	7.3
87	73.1	63.4	9.7	7	11.2	100	5.8

Figure 34: `EISSEC` **performance (with optimizations) to extract a complete AVR model**

Time to apply `EISSEC` to AVR. To enable symbolic execution of a code generator, `EISSEC` relies on manual modifications to parts of the decision tree. Mostly, these manual modifications are required in the places where the C code of the decision tree does not conform to the restrictions enforced by our source-to-source transformation system. Nonetheless, at several places, we rely on manual modifications to enable optimizations. Although, manual modifications are performed at several places, they demand porting effort in applying `EISSEC` to AVR. In total, applying `EISSEC` to AVR took approximately 7 hours and 45 minutes of manual modification effort. Note that this time does not include the time to extract the AVR model by executing the decision tree symbolically.

The model extraction performance. After several manual modifications to the code generator decision tree, we applied `EISSEC` to transform and execute the modified decision tree symbolically. We also enabled all the optimizations that we enabled for the x86 model extraction. The AVR decision tree has 2 subtrees under the root, so we explored both of them in parallel.

The detailed performance results are shown in Figure 34. The columns in the figure have the same meaning (but different units, e.g., thousands vs millions) as those in the evaluation of x86 model extraction performance. In total, `EISSEC` took 1 hour and 27 minutes to extract the complete model from the AVR code generator. This time is considerably less than the one taken for the complete exploration of the x86 code generator, because AVR is an embedded system architecture with hardly 76 instructions and only 3 modes (8-bit, 16-bit, and 32-bit). Consequently, the number of program paths explored by `EISSEC` are also less than those in the x86 code generator exploration.

Completeness of the extracted model. To evaluate the completeness of the extracted AVR model, we applied it to translate the same set of `coreutils` binaries (`ls`, `cp`, `cat`, `echo`, and `head`) used in the `LISC` test. The model was able to translate all of the assembly instructions from these binaries. Moreover, in terms of the number of mnemonics, it contained 72 out of total 76 mnemonics from the AVR architecture manual. Similar to the results reported in the `LISC` evaluation, we found that 4 mnemonics were not supported by the GCC's code generator (`break`, `nop`, `wdr`, and `sleep`).

4.6 Related Work

The idea of symbolic execution was proposed in the 1976 paper by King [49]. The idea of classical symbolic execution introduced in this paper has two key limitations. First, the limitations of the constraint solver in terms of solving the symbolic formulas can limit the applicability of symbolic execution. For instance, if a branch condition generates a non-linear constraint then a linear constraint solver cannot solve it, and the symbolic execution cannot proceed in such cases [39, 82]. Second, all the program parts that could be called from the program under symbolic execution needs to be symbolically executed. Specifically, that means, all the libraries used by the program needs to be symbolically executed. In fact, if a library used by the program invokes system calls with symbolic arguments, then even the system call handlers needs to be symbolically executed. Given the complexity of modern software (such as their interactions with environment, number of libraries used, etc) and the path explosion problem (the fact that number of program paths grow exponentially in terms of number of static program branches), classical symbolic execution of modern software is practically infeasible. That is why, in the last decade or so, a great deal of research [82, 39, 19, 21, 20, 40, 72, 24] has focused on improving classical symbolic execution so that it can be applied to modern and complex software. Significant improvements in the computational powers of SAT/SMT solvers have also fueled this interest considerably. These symbolic execution techniques are generally called as modern symbolic execution techniques. Given the improvements brought by modern symbolic execution systems, some of these systems are now used actively in software industry for software testing [40, 87, 13].

Modern symbolic execution systems. Modern symbolic execution systems address the limitations of classical symbolic execution by mixing concrete execution along with symbolic execution (Mixed concrete and symbolic execution is called as *concolic* execution [82].) In other words, these systems try their best to perform symbolic execution as long as they can, but if they come to a point where symbolic execution is not possible (such as when constraint solver cannot solve the formula, or a function call or a system call is made and we cannot perform symbolic execution of callee code) then they *concretize* the symbolic variables. *Concretization* essentially consists of assigning a concrete value to a symbolic variable. This is done by asking constraint solver for a possible value that could be taken by the variable. One can then simply select any value out of possible values randomly [39]. The drawback of concretization is that once we concretize a symbolic variable, the execution will follow only one path from the program tree rooted at the point of concretization — if we had continued symbolically, we would have traversed all of the program paths rooted at that point. This is generally referred to as *incompleteness* of concolic execution. Concretization, though it reduces the completeness of symbolic execution, it helps us get past a point where a classical symbolic execution would be stuck. Moreover, concolic execution helps in executing large and uninteresting parts of complex software concretely, and allows us to perform symbolic execution of selective part of the program. This is one of the crucial advantage of concolic execution, which we also exploit in our work. Specifically, a compiler code generator might refer to other parts of the compiler, but we are interested in symbolic execution of the code generator only. Concolic execution, in such case, allows us to treat other "uninteresting" parts of the compiler concretely. But instead of selecting one value randomly at the time of concretization, we choose all possible instantiations of a symbolic variable. We will now talk about common challenges faced by modern symbolic execution systems, and contrast existing solutions for these challenges with ours.

4.6.1 Path explosion problem

Path explosion problem is one of the key reasons why classical symbolic execution cannot be applied to modern complex software. Concolic execution tries to address this challenge by pruning some program paths by treating some parts of the program concretely. Fundamentally, path explosion problem in unsolvable — the number of program paths grow exponentially, but machines have limited physical resources (such as memory). So all of the existing approaches try to address this problem in the context of limited physical resources. These approaches can be classified into following two categories: (1) path prioritization, and (2) path pruning.

Path prioritization tries to prioritize paths which are likely going to yield "interesting" results over others. Techniques which follow this approach [19] use various heuristics for decision making. An example of an heuristic can be a coverage based selection — select a path which is likely going to explore unexplored parts of a program. Intuitively, the approach of these techniques could be thought of as making a best effort in fulfilling the analysis objective (bug finding, etc) without worrying about covering all the program paths.

Path pruning, on the other hand, tries to reduce the number of program paths either by merging them [50, 84, 29] or by pruning redundant paths [16]. Intuitively, multiple concrete executions can be combined together by representing them using a single symbolic execution. Merging multiple such concrete executions can thus reduce the number of program paths considerably. Techniques such as RWset [16], on the other hand, try to prune redundant program paths. One of their criteria to find a redundant path relies on the insight that if a program execution reaches some program point which has been visited before and both the times the symbolic state that was used at the point is same, then program execution will follow same program path from that point onwards. Such redundant program executions can then be killed. Such pruning can reduce the number of program paths also.

`EISSEC` also relies on techniques similar to above discussed techniques to address path explosion problem. `EISSEC`, unfortunately, does not really have a choice in terms of path selection or prioritization. This is because to obtain a complete model of the instruction set semantics, we need to cover all program paths. So `EISSEC` does not really use any path selection heuristic/algorithm. Its path selection can thus be considered as a systematic exploration — we systematically visit all program paths one-by-one by visiting *if* branch of a condition first and then going into *else* branch. Nevertheless, `EISSEC` uses optimizations such as use of range and set constraints and strength reduction to merge multiple paths together. Intuitively, whole idea behind these optimizations is to defer symbolic-to-concrete conversion as much as possible and thus avoid splitting of program paths as much as possible.

4.6.2 Constraint solver inefficiencies

Constraint solver plays a key role in symbolic execution systems because in many programs constraint solving time dominates the overall runtime. Moreover, some programs which cannot be executed symbolically might be producing constraints which bogs down constraint solvers. Consequently, a lot of attention has been paid to improve constraint solver abilities and performance. This has led to production of a number of efficient SMT solvers, such as STP [38] and Z3 [34], used in many popular systems such as KLEE and

Microsoft's SAGE, resp. Given that SMT solvers with the support for various theories such as arrays, match closely with the semantics of C language, most systems use SMT solvers as their decision procedures. Furthermore, given that constraint solving dominates overall runtime, most system employ some optimizations for query processing. Specifically, they try to eliminate queries as much as possible [19], employ counter-example guided cache [19], or perform simple static optimizations on queries [82, 19] before sending them to solvers. In our case, we did not employ any optimizations.

Unfortunately, SAT/SMT solvers are designed to efficiently find only one solution to a given formula. In other words, existing solvers are not really efficient for all-SAT problem (finding *all* solutions to a SAT problem) necessitating modifications to the core of these solvers [42, 95]. Fortunately, in the context of bug finding or software testing where these solvers are mostly used, producing one solution (or a few solutions) to a formula is enough. There is rarely a need for solving all-SAT problem in case of bug finding or software testing.

Unfortunately, model extraction problem addressed by `EISSEC` is an all-SAT problem: in order to ensure a complete semantic model, we need to find all inputs which reach a particular program path. Given that SAT/SMT solvers are not really efficient for all-SAT problem, we decided to use solvers based on constraint logic programming (CLP). CLP languages and their solvers have dealt with the problem of producing all solutions to a formula (given that the problem space is not too large) much more efficiently than SAT/SMT solvers. We discussed the details of our solver in the Implementation section, so we will not repeat that discussion here.

4.6.3 Symbolic execution for function extraction

Given that the symbolic execution allows exploration of all program paths, it can be used in the context of the problems related to verifying program properties. For instance, work presented in [10] uses symbolic execution in order to extract and verify cryptographic protocol models from their C implementations. Although the high-level idea of using symbolic execution to extract a function from C code is similar to `EISSEC`, there are number of differences. Specifically, the complexity of cryptographic code handled in this work is several orders of magnitude less than GCC's code generator. For instance, highest number of C lines symbolically executed by them is around 1000, while GCC's code generator is approximately 120K lines of C code. Moreover, their system confines symbolic execution to main path in the code and can only handle the protocol implementations with no significant branching. In order to reduce the complexity of symbolic execution, they manually build semantic models for commonly used cryptographic function. EISSEC, on other hand, would use concrete execution in such cases. Approach described in [94] uses model extraction of GUI programs from hand-held devices and compares extracted models with the expected models (obtained from specifications). Because of their context, their notion of models is different than ours. To be precise, their work is mainly concerned with extracting models to capture how system responds to user inputs. Consequently, their model is a state machine which captures system transitions on various event inputs. EISSEC definition of model, on the other hand, is simply the mapping between input (RTL) and output (assembly instructions).

Another application of the symbolic execution of a complete program could be in the program comprehension or software maintenance [70, 43]. To be precise, reverse engineering using symbolic execution can help the maintainers in understanding the code which is poorly documented. Preliminary work by Pichler et al [70] describes an approach similar to ours for extracting specifications from the programs. Although, the overall ideas used are similar to ours, there is one important difference. Specifically, this work uses dynamic symbolic execution to perform concrete program execution along with symbolic execution. Whenever symbolic execution comes to its limits, results from concrete execution are used to get past the limits. EISSEC, on the other hand, starts with symbolic execution only. Whenever symbolic-to-concrete conversion occurs, all possible concrete values of a symbolic variable are enumerated systematically. Another difference is that this work, being the preliminary work, does not present any evaluation results. So it is hard to evaluate their system with EISSEC.

4.7 Summary

In this Section, we described our symbolic execution system (called EISSEC) to explore GCC's code generator for model extraction. We had two questions in performing symbolic execution of the code generator: (1) how does the model extracted using symbolic execution compare with that obtained using LISC, and (2) whether we can address the all-path problem in the context of GCC's code generator. As compared to most of the existing symbolic execution systems that limit their exploration to few paths, our problem demands covering all the paths of a code generator. In order to solve this problem, we described several design choices (such as undo record), and we believe that the evaluation results demonstrated the effectiveness of our choices. In terms of comparison with the model extracted using LISC, we found interesting results. Specifically, we were able to cover more operand combinations and instructions than LISC. We believe that these results speak about the effectiveness of our symbolic execution system. Moreover, they also satisfy our expectations with which we started EISSEC.

5 ArCheck: Checking correctness of extracted semantics models

In the previous sections, we discussed our approach for extracting instruction set semantics from compiler code generators. But before we can use the extracted semantic models in some applications (e.g., to build binary translation system), we need to ensure that they are both correct and complete. The models may have correctness issues because compilers, given their complexity, may contain bugs. Bugs in code generators may result in generating incorrect assembly instruction(s) for some IR instructions. Such inconsistencies can introduce correctness issues in the applications built using these models. That is why it is necessary to remove these inconsistencies before they can be used in any application. To address this problem, in this section, we develop an approach, called `ArCheck`, to check the correctness of the models extracted from the code generators. Because completeness of the models has been discussed and evaluated earlier (in `LISC` and `EISSEC`), we will not discuss it here.

`ArCheck` is applicable to a number of problems. Although we demonstrate our approach on the models extracted from compiler code generators, the techniques that we develop in this section can be used to test the semantic models extracted from any source. Thus, `ArCheck` has very general applicability. Specifically, below are two of the possible applications of `ArCheck`.

- Semantic models are used by a number of software systems such as binary translators, virtual machines, system emulators, etc. Most of these systems play vital roles in software security. That is why it is important to ensure that these systems are tested thoroughly. `ArCheck` can be used to address this problem and to test the semantic models used by these systems.

- Another interesting application of `ArCheck` is to test compiler code generators themselves. Because models are extracted from compiler code generators, testing of these models can also test code generators. Recent research [97] has demonstrated that modern compilers such as GCC and LLVM contain many bugs. And these bugs highlight the limitations of existing compiler testing techniques. `ArCheck` thus makes an important contribution to the state-of-the-art in compiler testing.

5.1 Models and Inconsistencies

Given the general applicability, we will first discuss our technique to test the models extracted from any general source, and then will apply it for the models extracted from code generators.

5.1.1 Types of inconsistencies in semantic models

Recall that a semantic model captures the mapping between target assembly instructions and their semantics. Given this, the following are the possible inconsistencies in semantic models.

- **I1**: *Abstraction of semantics.* This kind of inconsistency occurs when semantics of an assembly instruction is captured correctly but loosely. For instance, for x86 `add` instruction which modifies only the carry flag (`CF`) and the overflow flag (`OF`) bits of `EFLAGS`, semantics of `add` might say that `EFLAGS` is modified, without detailing which bits are modified and how they are modified. This occurs frequently in compilers.

- **I2**: *Omission of semantics.* This kind of inconsistency occurs when some of the semantics of an assembly instruction are omitted while modeling that instruction. This is a correctness issue in the model because instruction semantics are not modeled correctly. An example of omission of semantics is forgetting to mention that x86 `add` instruction may modify the `EFLAGS` register.

- **I3**: *Over-provisioning of semantics.* This kind of inconsistency occurs when captured semantics of an instruction is more than the actual semantics. For instance, when semantics for the x86 `mov` instruction contains clobber of `EFLAGS` even when `mov` does not clobber it.

Note that the above definition of inconsistencies is with respect to modeling the semantics of assembly instructions in terms of IR. In this section, we talk about inconsistencies in modeling semantics of assembly instructions in the compiler IR. We take this view because this is how architecture specifications are usually written. Specifically, specification writers encode the semantics of the target machine instructions in terms of compiler IR instructions. Although we check modeling of the assembly instructions in IR for inconsistencies, the approach developed in this section is generic and can be applied to check IR-to-assembly modeling inconsistencies as well.

5.1.2 How to detect inconsistencies in semantic models

Conceptually, to detect inconsistencies we check that for every instruction, its semantics are "correct". Let us assume the existence of an oracle which would tell us the "correct" semantics of every assembly instruction. To detect inconsistencies for every assembly instruction in the model, we obtain its semantics from the oracle, and check if it same as the semantics present in the model. Architecture manuals or actual hardware CPUs can function as an oracle. (For this report, we assume that the hardware manuals or the processors are "correct.") Unfortunately, architecture manuals are not in an easily-parsable form, thus it is hard to obtain semantics from them automatically. CPUs, on the other hand, offer a solution to automate the process of semantic extraction. We rely on the CPUs to function as the oracle for this study.

Can we perform semantic equivalence checking of the abstract rules themselves? Use of hardware to obtain the semantics of assembly instructions would require concretization of the abstract mapping rules from the semantic models. This is because we cannot execute abstract assembly instructions on the hardware CPU.

Symbolic equivalence checking [75] (or *symbolic simulation*) is a commonly used approach to prove equivalence of two programs. It is a popular technique in the electronic design verification field to check the equivalence of RTL or gate-level descriptions in HDL. The idea behind symbolic equivalence checking is to obtain symbolic formulas that capture the semantics of the programs to be checked, and then to prove that those formulas are equivalent by using constraint solvers or theorem provers. Symbolic formulas are obtained by executing programs *symbolically* (i.e., by making program inputs symbolic, instead of executing concretely). Symbolic input values propagate during program execution from source to destination. Furthermore, whenever control flow branches in a program, all branches are explored unless any of them is a provably infeasible path. Symbolic traversal of program paths ends when program execution ends. The equivalence of formulas for each program path for two programs is then checked using a constraint solver.

Unfortunately, in our case, symbolic equivalence checking cannot be used to prove the equivalence of abstract IR-to-assembly mapping rules. This is because equivalence checking demands representations of both IR and assembly instructions, such that they are amenable to symbolic execution. The representation of IR instructions satisfies this criteria, but unfortunately, that of assembly instructions do not. Consequently, if we want to apply symbolic equivalence checking to assembly instructions, then we must encode semantics of each assembly instruction in some traversal-friendly representation manually. As discussed in section 1, this is exactly what we want to avoid, given the fact that such manual encoding is often the root of many modeling errors.

Had it been the case that there were two different semantic models M and M', such that $M = \{\langle A, I_{I1} \rangle\}$, where I_{I1} is an IR instruction used to represent the semantics of an assembly instruction A in some intermediate language, and $M' = \{\langle A, I_{I2} \rangle\}$, where I_{I2} is an IR instruction for the same assembly instruction in some other intermediate language, then we could apply symbolic equivalence checking to both IRs for the same assembly instruction. But, unfortunately, when we have only one model, we cannot perform symbolic equivalence checking on abstract rules.

How to convert abstract mapping rule into concrete one? Recall that the difference between abstract and concrete mapping rules is that the operands of concrete mapping rules are concrete. So, conceptually, to convert abstract mapping rules into concrete rules, we must produce concrete operands in place of abstract operands. The process of concretization will generate multiple concrete rules for a single abstract rule. So the question then is if we should generate all of the possible concrete rules, or we can generate only a few. This question translates into even more fundamental question: *how many concrete mapping rules do we need to test using semantic equivalence checking to obtain same level of confidence that we would have obtained if we had tested abstract rules directly?* Note that, in the worst case, we would need to test all possible concrete rules to test a single abstract rule. This is undesirable, because the number of concrete rules would grow combinatorially in the number of operands. For instance, when an instruction has a single operand which can be a register, and the target architecture of the instruction has 6 registers, then there would be 6 concrete rules. On the other hand, if there are 2 operands of type register, then there would be a maximum of 36 concrete rules. We can systematically eliminate some concrete rules. Intuitively, ways to eliminate the rules would arise from understanding how we can constrain the operands of abstract mapping rules.

- *Constraints defining invalid operands and their combinations.* One of the easiest ways to eliminate concrete rules is to rely on constraints that restrict the types of operands the instructions can have. For instance, the x86 `mov` instruction accepts two operands, both of which cannot be memory operands simultaneously. Such instruction-specific operand constraints are encoded in the applications that use semantic models. For instance, the architecture specifications used by the code generators of modern compilers contain such constraints for assembly instructions. An approach to enforce such constraints is to generate a concrete rule and use these applications to check if that the rule is valid. This approach eliminates the need of enforcing such constraints ourselves.

- *Constraints specifying functionally similar operands and their combinations.* Once rules containing invalid operands are eliminated, the rest of the rules are all valid rules. To eliminate some of the valid rules, we rely on functional-similarity of assembly instructions containing the same mnemonic but different operand combinations. Specifically, by functional-similarity we mean that most of the operand combinations of an assembly mnemonic generally perform similar functions. For instance, the x86 `mov %eax,%ebx` and `mov %eax,%ecx` perform the same function of moving values between two registers. We can select one representative for all of the functionally-similar combinations. For instance, if we generate a concrete rule for the `eax` and `ebx` operand combination, then we can eliminate the `eax` and `ecx` combination. Relying on this observation, for every architecture, we define operand combinations which perform the same function for abstract rules.

A generic approach to generate concrete rules from abstract rules could be: (1) for every abstract rule, find out how many operands it takes. Let us denote this number by K, (2) using N different sets of functionally-similar operands, generate N^K different operand combinations (note that N^K is reasonably low in practice), (3) feed N^K different instances of concrete rules to the code generator and check which of them are valid.

5.1.3 Detecting inconsistencies in compilers' semantic models

To detect inconsistencies in the compiler semantic models, we first formalize the semantics of assembly and IR instructions. Both IR and assembly instructions operate on processor state. Hence their semantics can be formalized in terms of the changes they make to the processor state. Note that an IR's view of processor state can be more limited than the assembly-level view. For instance, the processor state may capture detailed information about condition codes and other details of a processor's internal state, which may not be of interest to the code generator. A natural approach, therefore, is to expose only a subset of the assembly-level state at the IR-level. This discussion leads to the following definition that formalizes a notion of correspondence between processor states at the IR- and assembly-level.

Definition 4 (State at assembly- and IR-level) *The state $\mathbf{S}_A$ at the assembly level is given by an assignment of values to a set of variables V_A representing the processor's user-space accessible registers and memory. Each variable $v \in V_A$ takes values from an appropriate domain.*

The state $\mathbf{S}_I$ at the IR-level is given by an assignment of values to a set of variables V_I representing the processor's state as viewed by the code generator. We assume that $V_I \subseteq V_A$. The set of permitted values for a variable v at the IR level will include all of the values permissible at the assembly level, plus a special value called $\top$ that captures the idea that the value is unknown or undefined. $\mathbf{S}_I$ is said to be valid *for an IR snippet I if for every variable v read by I (excluding those variables that are first updated and then read by I), $\mathbf{S}_I(v) \neq \top$.*

Definition 5 (Semantics of an assembly and IR instruction) *The semantics of an instruction at the IR (or assembly) level can be understood in terms of how it modifies the processor state. We use the notation $I : \mathbf{S}' \longrightarrow \mathbf{S}''$ to denote that the execution of I in state $\mathbf{S}'$ leads to a new state $\mathbf{S}''$.*

Use of actual hardware as the oracle introduces a complication in the problem. Semantics of an assembly instruction obtained from hardware would be in the form of a hardware state, and we would need to produce IR semantics in terms of hardware state for comparison. An easier approach to obtain IR semantics in terms of hardware state would be to develop a "hardware" for IR where we can "execute" the IR and obtain its semantics. Once both semantics — the semantics of an assembly instruction and its IR — are obtained in terms of hardware state, comparing them would be easier.

Definition 6 (Processor state correspondance) *States $\mathbf{S}_A$ and $\mathbf{S}_I$ are said to correspond, denoted $\mathbf{S}_A \sim \mathbf{S}_I$, if:*

$$\forall v \in V_A \; (\mathbf{S}_I(v) = \mathbf{S}_A(v)) \vee (\mathbf{S}_I(v) = \top)$$

Alternatively, one can say that $\mathbf{S}_I$ *is a conservative approximation of* $\mathbf{S}_A$*:* either they agree on the value of a state variable, or $\mathbf{S}_I$ leaves it unspecified. The latter choice is made by the developers of machine descriptions, who choose not to model the exact value of a state variable, but simply state that an instruction "clobbers" it. (This imprecision is deliberate, as it makes it possible to develop machine descriptions that work across variants of a processor. They also provide a way out when the instruction set description is ambiguous or unclear about the final value of a state variable.)

While precision of the model can be an important property for some applications of the semantic models, correctness of the model is a fundamental property of the model without which the applications of the models would need to worry about soundness and correctness. Moreover, if we consider precision for state correspondence then too many inconsistencies would be reported. That is why for compiler semantic models, we do not consider **I2** as an inconsistency. Nonetheless, by tweaking the definition of state correspondence, it is quite easy to change this decision.

I1 and **I3**, on the other hand, must be considered for state correspondence because they represent cases in which semantic equivalence of an assembly instruction and its modeling in IR — one of the important guarantees given by code generators, as per our assumptions — is violated.

Definition 7 (Soundness of IR) *I is said to be a <u>sound</u> abstraction of the semantics of A, denoted $I \sim A$, if the following condition holds for every state $\mathbf{S_I}$ that is valid for I and all states $\mathbf{S_A} \sim \mathbf{S_I}$:*

$$\big((I : \mathbf{S_I} \longrightarrow \mathbf{S'_I}) \ \wedge \ (A : \mathbf{S_A} \longrightarrow \mathbf{S'_A})\big) \Rightarrow \mathbf{S'_A} \sim \mathbf{S'_I}$$

Mapping rule $\langle I, A \rangle$ is said to be sound, if $I \sim A$.

From the definition of state correspondence, it is easy to see that if I is *not sound* for A then there will be at least one variable v that will have conflicting values in $\mathbf{S'_A}$ and $\mathbf{S'_I}$. This means that a subsequent instruction I' that relies on v will likely diverge from the behavior of A' even when I' and A' are equivalent in every way. In other words, a code generator that emits A for I will generate incorrect code unless it ensures that none of the following instructions rely on v's value. Doing so complicates the code generator logic, because the mappings from IR to assembly now become context-specific. More important, such an approach seems to defeat the purpose of architecture specifications used in compilers such as GCC and LLVM: the purpose of these specifications is so that the compiler does not have to reason about the semantic equivalence of IR and assembly, yet this step requires that very same task to be performed! For this reason, we believe that code generators will translate I into A when I is *sound* for A.

Definition 8 (Testing extracted model for inconsistencies) *Let I be an IR snippet and A be the assembly code present in the extracted model. The problem of testing the extracted model for inconsistencies is to test every such $\langle I, A \rangle$ pair to determine if $I \sim A$.*

The problem of testing extracted model for inconsistencies is also same as the problem of testing code generators for correctness. Formally, it can be defined as below.

Definition 9 (Checking correctness of compiler code generators) *Let I be an IR snippet, and A be the assembly instruction produced by a code generator $\mathcal{G}$ for I. The problem of checking correctness of $\mathcal{G}$ is same as the problem of testing model extracted from $\mathcal{G}$ for inconsistencies.*

Unfortunately, given that a state is defined by the contents of registers and memory locations (Definition 7), a number of possible start states is enormous. Thus, verifying the semantic equivalence of mapping rules by exhaustive enumeration of start states is practically infeasible. Thus, instead of *verifying* the semantic equivalence of the mapping rules, we *test* their equivalence under some specific start states with the aim of uncovering as many semantic differences as possible. An important challenge faced by our approach is the design of start state generation strategy — *a strategy that we are looking for must be such that with the smallest number of start states, maximum of the semantic differences should be uncovered.*

Checking correctness of mapping rules also helps in finding semantic equivalence bugs in the code generator architecture specifications. That is why we call our approach `ArCheck` (Architecture Specification Checking). In other words, we check if the architecture specifications used by the code generators *conform* with the specifications used by the actual hardware. An important contribution of our approach is that it is architecture-neutral and very practical: given the challenging problem of *verifying* modern compilers, we develop an approach which can be easily applied by code generator developers at the time of development to detect all the correctness issues in the architecture specifications. We believe that such checking will in turn help developers in discovering many compiler bugs ahead of time, which, otherwise, could lead to compiler crashes, or even worse, generation of wrong code.

Any software testing technique must address two fundamental problems in testing: *test generation* and *result verification*. For result verification, as we mentioned before, we will use hardware CPU as the test oracle. Test case generation is one of the fundamental challenges that any software testing approach must address. To handle this challenge, we develop a coverage testing strategy to expose most of the semantic differences in the extracted mapping rules. Generating test cases to check semantics of assembly instructions is a very hard problem because it requires one to know the exact semantics of each and every instruction. Consequently, developing an automated test generation strategy, and that too which considers assembly instructions for multiple architectures, is a very challenging problem. We exploit an important observation to tackle this problem. *IR instructions often exposes the semantics of assembly instructions in detail. Consequently, if we develop a test case generation strategy by leveraging instructions in architecture-neutral languages, then such strategy can be effectively automated and can also be applied to multiple architectures.*

5.2 Our Approach

Our high-level approach, called `ArCheck`, to check correctness of semantic models extracted from GCC's code generator is depicted in Figure 35. We describe main steps of `ArCheck` below.

- **Instantiation.**

  First step of `ArCheck` is to instantiate abstract rules obtained from the extracted semantics model. Since we have already described a generic approach to obtain concrete rules from abstract rules, we will not repeat it here. Instead, we will describe how one can apply that approach to x86 below.

  An important challenge in instantiation is how to define functionally similar operands and their combinations. This requires one to be familiar with the target architecture of the extracted model and functions performed by different operand combinations. For x86-32, we have defined following functionally similar categories:

  - **general-purpose registers:** `eax`, `ebx`, `ecx`, `edx`, `esi`, and `edi`,
  - **stack-related registers:** `esp` and `ebp`,
  - **Constant integers:** `[INT32_MIN, INT32_MAX]`
  - **Constant memory addresses:** `[0, 4GB]`
  - **Memory operands:** `ConstInt(Reg)`, `ConstInt(Reg,Reg,ConstInt)`, etc

  Note that we have not specified details of some other categories such as constant floats above. Nonetheless, above description essentially capture the the notion of functionally similar categories. In all there are around 6 different categories of functionally similar x86 operands. For an abstract rule containing `mov %1,%2`, we

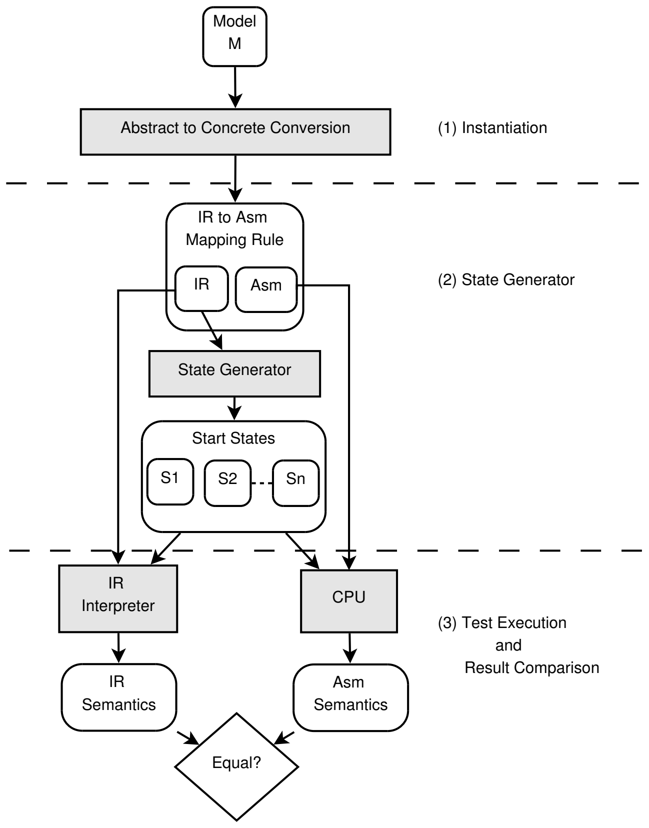


Figure 35: **Design of** `ArCheck`

would generate 2^6 different concrete rules by systematically selecting both operands from each of the 6 different categories such that duplicate operand combinations are avoided. Note that not all of these 2^6 combinations are valid, e.g., `mov $100,$200` is an invalid combination. We will realize this after feeding such `mov` instruction to the code generator. Also note that we have made a category of memory operands, because putting them in one category eliminates a possible source of explosion in the number of operand combinations.

Another challenge is related to how many instances of an abstract rule should we test. Intuitively, higher the ratio of abstract to concrete rules higher should be the level of confidence in the soundness of the extracted models. So we thought about three different selection points: one instance, multiple instances, and all-possible instances. All-possible instances option is same as exhaustive enumeration of all operand combinations of an abstract rule. To reflect these three options, we have developed three different modes for generating different number of concrete rules from abstract rules:

  - *single-instance mode* (**SI**): in this mode, one valid operand combination out of N^K is selected while instantiating abstract rules.
  - *multiple-instances mode* (**MI**): in this mode, multiple valid operand combinations (such as 8 — the idea is to pick a number between 1 and N^K for the target architecture) are selected while instantiating abstract rules.
  - *restricted exhaustive enumeration mode* (**REE**): Since testing all-possible N^K operand combinations of an abstract rule may not be practically feasible, we develop a restricted exhaustive enumeration mode, where the ratio of number of abstract to concrete rules is restricted to a sufficiently high number (closer to N^K but still such that testing is practically feasible).

- **Generating start states.** Once abstract rules are instantiated, the next step in the process consists of generating the start states to be used for the semantic equivalence test. `ArCheck` relies on white-box analysis of IR instructions to generate start states for a given mapping rule.

- **Test execution and result comparison.** Last step in `ArCheck` consists of using the generated start states to obtain semantics of an IR and the corresponding assembly instruction from a given concrete mapping rule. To obtain the semantics of an IR instruction, we have developed an IR interpreter which accepts the IR instruction and the start states generated by the previous step as input. It then interprets that IR instruction after setting the interpreter in the given start states. The difference between end states of the interpreter and its start states is the semantics of that IR instruction. To obtain the semantics of an assembly instruction, we have developed a hardware-based oracle. Once the semantics of both an IR and the assembly instruction are obtained, we compare them for equivalence.

We will now describe state generation, and test execution and result execution step in detail.

5.2.1 Start state generation

Generating start states for checking the correctness of mapping rules from the extracted model is one of the important challenges for our approach. This is because the goal of `ArCheck` is to achieve all of the following objectives: (1) maximize the possibility of finding semantic differences in the mapping rules, (2) minimize the number of start states generated for a given mapping rule, and (3) develop a strategy such that it can be easily applied to test models of other architectures. Note that these objectives actually outline the requirements of a test generation strategy for the problem of checking the correctness of mapping rules: given the challenges of verifying modern compilers and seemingly infeasible approach of exhaustive enumeration, how can one develop a practical approach for achieving maximum confidence in the correctness of extracted models (and in code generators, in general)?

What are "interesting" outcomes and "interesting" inputs. We believe that first two objectives mentioned above naturally lead to the following requirements: (a) generated test inputs must "represent" most of the remaining inputs from the input space, and (b) the number of generated test inputs must be reasonably small subset of the input space. By "represent" we mean that the semantics of an instruction for the representative inputs should be same as that for the non-representative inputs. Requirement (b) says that such representative inputs must be a reasonably small subset so that testing of a single mapping rule is practically feasible.

An ideal test strategy that can satisfy both the requirements would partition the input space according to the inputs for which an IR or the assembly instruction exhibits same semantics, and then select one representative test input from each partition. We believe that such strategy maximizes the confidence in the soundness of mapping rules because the selected test inputs would explore the semantics of an instruction thoroughly. That is why we call them "interesting" test inputs.

Unfortunately, generating "interesting" test inputs for each and every mapping pair is a practically infeasible task. Note that the notion of "interesting" inputs assumes that we already know what is the semantics of an instruction for each and every input value. But obtaining such knowledge is practically infeasible because (1) it would require us to exhaustively explore all input space and obtain instruction semantics for each input, and (2) semantics of an instruction depends on values of its operands. For instance, for assembly instruction from Figure 5, if `eax` contains -2, then output of `add` is 0 and, on x86, zero flag will be set. On the other hand, if instruction was `add $1, %eax`, and `eax` contained -2, then output would have been `-1`. The point here is that same inputs to different assembly instructions produce different semantics. Thus, if we want to generate "interesting" inputs, then we need to generate them for each and every mapping pair. That is why we do not follow this approach.

Instead, we make an important observation that one could enumerate different output values corresponding to different semantics for an instruction, and then obtain inputs which would produce those output values. Since number of different semantics for an instruction is small, such enumeration seems feasible. Additionally, such approach would avoid problems in generating "interesting" inputs, and at the same time, satisfy all of our requirements. For instance, for both `add $1, %eax` and `add $2, %eax`, one of the desired outputs will be 0 because value 0 is treated specially by most architectures. Once we fix our desired outputs, we can find out which inputs lead to those outputs. That way we do not need to perform exhaustive enumeration at all, and we still obtain "interesting" inputs. A set of desired outputs is what we call "interesting" outcomes. Specifying such "interesting" outcomes and deriving "interesting" inputs from them is the core of `ArCheck`.

We will concretize our discussion about "interesting" outcomes now. Taking Figure 5 as an example, the `plus` operator in the IR performs a signed 32-bit addition. And various possible outcomes are: (1) a positive value in the range of [1, INT32_MAX], (2) 0, and (3) a negative value in the range of [INT32_MIN, -1]. They are divided in such categories to capture the different behaviors the CPU might have for different outcomes. For example, for `add` instruction of x86, when the sum is 0, zero flag is set. But it is not changed in other cases. In addition, the signedness and the size are also factors for considering the categories, since they are relevant for possible overflows during addition. Therefore, "interesting" inputs to test soundness of a mapping pair would be those which generate outcomes belonging to every category at least once. In this example, One can simply select boundary values of the above two ranges and 0, i.e., {INT32_MIN, -1, 0, INT32_MAX} as interesting outcomes.

With above discussion, it is quite easy to see why random testing would not be a good strategy for our purpose: random testing does not have any notion of partitioning of space of possible inputs and outcomes. Furthermore, as the number of partitions of the input space increases, its confidence in the soundness of mapping rules decreases.

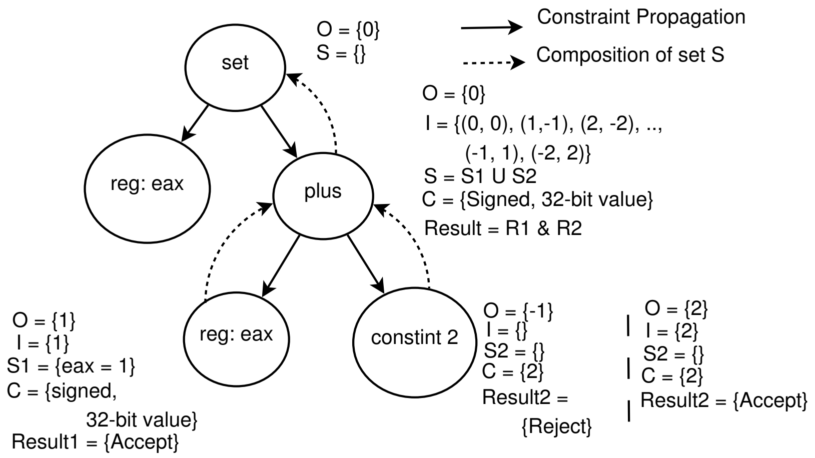


Figure 36: **Constraint generation and propagation for start state generation**

5.2.2 Obtaining "interesting" test inputs: constraint generation, propagation and solving

Unfortunately, designing a generic strategy for generating "interesting" test inputs for a mapping rule is challenging problem. This is because such inputs depend a number of factors such as types of source and destination (e.g., 1-byte vs 4-byte), the way in which input data is used (e.g., is the value in the operand used as a number or as a memory address for indirect reference), etc. Additionally, since we are going to use the obtained test inputs in order to obtain semantics of an assembly instruction, it is necessary that the constraints imposed by the physical machine state are considered while generating start states. For instance, it is better if `esp` and `ebp` point to `R+W` pages rather than inaccessible pages. While it is not absolutely necessary that such constraints are satisfied, we consider them because invalid memory access errors do not represent interesting semantics for us. This is because compiler IRs do not have a notion of physical memory layout, consequently, they do not have a notion of invalid memory access.

Given the discussion about possible constraints on the inputs, it is easy to see that the test case generation problem that we are trying to solve can be formulated as a constraint satisfaction problem. "Interesting" test inputs are then simply solutions of such constraint satisfaction problem. A way to formulate constraint satisfaction problem from the white-box analysis of IR instruction for `add eax,$2` is shown in Figure 36. Notice that in the Figure IR instructions are represented as trees. The way in which we obtain start states from the IR tree in the Figure is as follows. Set *O* represents the set of outputs of the IR instruction. In the figure, we set it to 0, i.e, we are trying to solve the following constraint satisfaction problem: "find out a 4-byte signed integer value for `eax` which when added with 2 would produce the output of 0." S is a set to represent the state initializations which satisfies the constraints. For the example in the Figure, S contains an assignment of `eax`. Once we decide set O for a given mapping pair, the analysis starts by propagating set O of the root of IR tree to the set O of the first right child. This is done because the operation performed by RTL is actually the root of right subtree. That is why O of `plus` is same as that of `set`. `plus` node then looks at constraints C that can be enforced on I, the set of inputs which produce the expected output. For this example, it enforces the constraint that the addition must be signed 32-bit. Solving this constraint produces all pairs of signed 32-bit integers which add to 0 in set I. `plus` node then takes every pair of I, and propagates its first element to its left child and second element to right child. This is done till all children of a node accept the assignment of values. For $(\texttt{1}, -\texttt{1})$, `reg : eax` node accepts the assignment because it satisfies its constraints, but `const_int 2` node does not accept the assignment of -1 because it does not satisfy its constraint. Analysis then goes on with this process till it reaches $(-\texttt{2}, \texttt{2})$ for which both children of `plus` accept the assignment. Once a satisfying assignment is found, analysis stops exploring any further input pairs. This is because we are interested in only one representative input for any given partition of the input space. Note that all pairs of I which generate same element of O are in one partition. Once `reg : eax` accepts the assignment, S is updated with the value for `eax`. Since `const_int 2` does not correspond to any part of the state, it does not update S. Set S for `plus` is then obtained by unifying S of all of its children for a satisfying assignment. Set S is thus formed in the bottom-up fashion, while the constraints propagate in top-down fashion. S for `set` thus contains an element `eax` $: -\texttt{2}$.

An algorithm to generate "interesting" inputs for IR to assembly mapping rules is given in Figure 37. This algorithm starts with a call to `generate_start_states` function with *ir* as the IR instruction of a mapping rule, and it ends by returning set S, the set of start states to test a given mapping pair. Using function `get_outputs` and by passing it the top-level IR operator *op* and its type, it obtains a set of expected outputs for the given mapping pair. For instance, for RTL 32-bit signed `plus` operator, it would obtain $\{\texttt{INT32\_MIN}, 0, 1, \texttt{INT32\_MAX}\}$ as the expected outputs.

Note that `get_outputs` is manually specified, and its implementation will typically be different for each IR operator. For this reason, we have not shown it in the figure. Since the number of operators in the IR is relatively small, the effort involved in this step is relatively small. Despite being manual, our approach does provide a systematic way to divide outputs into equivalence classes, while enabling us to leverage human knowledge

```
generate_start_states ($ir$):
    $op$ = $TopOp(ir)$
    $O$ = get_outputs($op$, $Type(op)$)
    **foreach** $o_i$ **in** $O$ **do**
        // get start state which would produce output $o_i$
        // and add it to $S$, which is set of all start states
        $S$.add(get_inputs_for_output($ir$, $o_i$))
    **done**
    **return** $S$

get_inputs_for_output ($ir$, $o$):
    $op$ = $TopOp(ir)$
    $I$ = get_inputs($op$, $o$, $Type(op)$)
    $H$ = $Children(ir)$
    $s = \{\}$
    **if** $H = \emptyset$ **then** // leaf node
        // does input match constraints
        **if** check_cons ($o$, $ir$) **then**
            $s$.add($o$)
            **return** $s$
        **fi**
    **fi**
    **foreach** $(i_1, \ldots, i_n)$ **in** I **do**
        **foreach** $ir_j$ **in** $H$ **do**
            // propagate constraints to children
            $s_j$ = get_inputs_for_output ($ir_j$, $i_j$)
        **done**
        **if** none of $s_1, \ldots, s_n$ are empty **then**
            // all children have valid input assignments,
            $s$ = merge($\{s_1, \ldots, s_n\}$) //now combine them.
            **return** $s$
        **fi**
    **done**
    **return** $\{\}$ // if no satisfying assignments found
```

Figure 37: **An algorithm for obtaining "interesting" inputs from an IR instruction of a mapping rule**

and insight to minimize the number of such classes.

Once a set of expected outputs is obtained, function `get_inputs_for_output` is called to obtain a set of inputs for each of the expected outputs. In order to find inputs which produce the specified output, `get_inputs_for_output` first calls `get_inputs` by passing it the operator, the expected output value, and the type. For the "`add`" example, one of the calls would be $\texttt{get\_inputs}(\texttt{plus}, 0, \texttt{Int32})$. Note that the set C discussed in the Figure 36 can be seen as a collection of $\{\texttt{plus}, \texttt{Int32}\}$ here. Note that `get_inputs` returns a set I of all 32-bit integers, which, when added, produce the expected output. I is a set of tuples whose arity matches that of the IR operator. For instance, for `plus`, tuples will have 2 elements, while for unary negation, tuples will have only 1 element.

Note that function `get_inputs` might return an empty set in case it could not find an inputs which satisfy combination of op, o, and $Type(op)$. In such case, constraint propagation cannot proceed, and we return immediately.

On the other hand, if I is non-empty, then constraint propagation starts as an element of set I would now become expected output of the children of ir. In order to propagate constraints, it obtains set (H) of all children of given IR instruction. H would be empty for leaf nodes of IR, in which case, it simply checks if the constraints imposed by that IR node are satisfied by the input to that node. For instance, for $(\texttt{mem}\ (\texttt{reg eax}))$ RTL expression, it would check if the input value is a valid memory address which points to an accessible page. Note that an input of a leaf node would be same as the o, that is why we simply check using value of o. Given the small number of IR expressions which impose constraints at the leaf node, we have simply enumerated these constraints manually. Note that if a constraint is matched even for one input value, the algorithm updates state s and returns it. This is because we are only interested in finding a single representative. If constraints are not satisfied by the input value, then it continues to find the next input value which matches those constraints. When H is non-empty, input values from tuple $(i_1, \ldots, i_n)$ are passed to respective children of the input IR. If all children accept the assignment, then the algorithm returns the obtained state s as the output, otherwise it continues to next i. If no satisfying assignments are found for all children, then we return empty set.

5.2.3 Test execution and result comparison

For IR execution system, `ArCheck`, relies on an interpreter for that IR. Compiler such as LLVM already provides an interpreter for its IR, but others such as GCC do not. Since semantics of compiler IRs is defined precisely, it is straight-forward to develop an interpreter if there does not exist one already.

Our high-level approach to obtain semantics of an assembly instruction is to execute the test instruction under a user-level process monitoring framework. Such a framework satisfies all the requirements needed from an assembly execution system for this problem. First, it can execute the test instruction in a separate isolated environment and monitor the execution. Second, by relying on process tracing features offered by most OSes, it can inspect and modify register/memory contents for test execution. Third, it can gracefully handle exceptions or signals generated by the test instruction. The framework first assembles a test program containing the test instruction, creates an isolated environment for the execution of the assembled code, and initializes the environment with the specified start state. Isolated environment ensures that all effects of a test execution are confined. The difference between start and end states of the environment at the end of the execution would be semantics of the test instruction.

For memory, semantics that we are interested in is being able to tell older and newer contents of all the memory locations whose contents have been modified as a result of test execution. A simple approach to satisfy the desired memory semantics would be to snapshot memory of the isolated environment just before and after the execution of a test instruction and to compare the snapshots. This approach may seem too inefficient because, in the worst case, it could require us to compare snapshots for whole virtual address space (4GB on a 32-bit machine) of a process. But we observed that the virtual memory layouts of our assembly execution system and isolated environments are very sparse. Given this, we decided to use this simple approach for our purpose.

Nonetheless, we made an important observation which helped us optimize simple approach considerably. Specifically, we observed that the difference between start and end state of memory needs to be calculated only if memory is updated by the test instruction. In other words, if memory is only read by the test instruction, then we do not need to compare memory snapshots at all. To put this observation to use, `ArCheck` marks memory as read-only. With this setup, the framework can receive memory access fault in two cases: when test instruction accesses invalid memory location, and when the test instruction performs write access. We distinguish between these two cases by making memory read-write and re-executing the test instruction. If the framework receives one more fault, then it is because of first case now, and it is dealt as an exception case. If the framework does not receive any fault, then it is the second case, and the framework makes a note that memory snapshots must be compared in such situation. Note that as an additional optimization, the framework also notes the pages that will be modified and only considers those pages in comparing memory snapshots. Furthermore, if the framework does not receive memory fault in the first place, then it means that either the instruction does not access memory at all, or it performs read access. In both situations, we do not need to compare snapshots at all. Consequently, in both these situations, our system does not even snapshot memory at the end of the execution.

Once end states of assembly and IR executions are obtained, we follow Definition 6 and compare them.

5.3 Implementation

Prototype implementation of our approach targets GCC's x86 code generator and runs on 32-bit Linux. The approach, however, is more general, and can easily be applied to other compilers and architectures.

5.3.1 Obtaining concrete rules from abstract mapping rules

Implementation of instantiation step needs some architecture-specific knowledge such as different types of operands and their combinations. But once such knowledge is encoded in the system, rest of the implementation of instantiation (feeding concrete rule to GCC and checking validity of concrete rules) is architecture-neutral. While **SI** and **MI** mode relies on architecture-specific knowledge, **REE** mode is implemented by randomly selecting a sufficiently high number (architecture-specific and configurable) of operand combinations per mapping rule.

5.3.2 Obtaining start states for a mapping rule

The algorithm for start state generation is implemented in 1000 lines of C code and it uses a constraint solver written in 500 lines of Prolog. Expressiveness of constraint language essentially comes from the expressiveness of IR — we simply map semantics of IR operators to a sequence of constraints in Prolog.

Most of the components of state generation are architecture-neutral, but we do need an architecture-specific component to impose architecture-specific constraints such as those related to memory layouts.

5.3.3 Obtaining RTL semantics

Since there does not exist an interpreter for GCC's RTL, as a part of our prototype, we have developed an RTL interpreter by referring to GCC's RTL specification. Implementation of the interpreter took around 3K lines, and is mostly architecture-neutral. Only architecture-specificity is in initializing interpreter's state from the input start state. This initialization code for x86 is about 50 lines of C++ code.

Errors in the IR interpreter can result in false positives or false negatives in `ArCheck`. To address this challenge, we have tested our RTL interpreter by using it to interpret RTL programs obtained from source code of `coreutils` package. Furthermore, whenever `ArCheck` reports a semantic difference, we manually verify that the RTL semantics does not match that of the assembly instruction. So far in our testing, we have not encountered any false positives or negatives.

5.3.4 Obtaining assembly semantics

Implementation of our user-level process monitoring framework relies on parent-child relationship and `ptrace()` interface of Linux. Test execution starts with the framework assembling a test program containing the test instruction and then creating a child process (using `fork()`) for executing the test program. The test program consists of a test instruction wrapped with two trap instructions. Trap instructions allow the parent (i.e., the framework) to intercept child's execution (by writing a trap handler) just before and after the execution of the test instruction, to set the start state, and to capture the end state of the execution. After `fork()`, child calls `mmap()` to map the binary encoding of the assembled test program and jumps to its beginning, while parent calls `wait()` and blocks itself. Once trap handler performs necessary actions such as setting or capturing the state, parent continues child's execution by sending `PTRACE_CONT` request to child. Along with trap, parent also handles other signals that might be raised by child by registering signal handlers for them. Although, cases leading to signals are not interesting for our purpose, parent handles them to exit gracefully.

To access child's execution state, the monitoring framework relies on the `ptrace()` interface provided by the Linux kernel. In particular, using `PTRACE_SETREGS` request parent process can set the registers used by the child process, while using `PTRACE_GETREGS` it can get their values. This is needed to initialize hardware state from input start state and also to save the end state of child's execution. We will discuss the way in which the framework handles memory shortly.

Overall flow of getting the semantics of an assembly instruction using our framework is as follows.

1. To start, parent takes both an assembly instruction to be executed and a start state as input. The start state is obtained from the algorithm discussed earlier in the report.

2. Parent then composes a test program, by simply wrapping the test assembly instruction by two trap instructions. Test programs thus consist of sequence of three assembly instructions. We will discuss the role of trap instructions shortly. Parent then assembles the test program using `as` assembler and produces a binary file containing the shell code (byte encoding) for the test program using `objcopy`. To handle exceptional cases during child's execution, parent then sets signal handlers for most of the common signals such as `SIGSEGV`, `SIGFPE`, `SIGTRAP`, etc. Finally, parent calls `fork()` to create a child process and passes the file containing the shell code of the test program to the child. Parent then calls `wait()` and blocks for the child to execute the test program.

3. Upon starting its execution, child process first calls `mmap()` to map the shell code of the test program as an executable page, and then jumps to the first instruction from the test program.

4. First instruction of the test program is a trap instruction. When child executes this instructions, OS halts the child process and notifies parent immediately via `SIGTRAP`. Upon receiving first `SIGTRAP`, parent notes that the child is about to execute the test instruction, and so it initializes child's register state using `PTRACE_SETREGS`. Use of trap thus allows parent to initialize child's hardware state just before the execution of the test instruction. After hardware state is initialized successfully, parent resumes child's execution by sending it `SIGCONT` signal, and again blocks itself by calling `wait()`.

5. Child then executes the test instruction. Execution of the test instruction can lead to any exceptional situations, which are all handled by the parent (if such situations lead to generation of a signal.) In most of the exceptional cases, such as execution of illegal instruction or unaligned memory access, it is not safe to continue with the child execution. Consequently, parent simply records the diagnostic information about

the situation and then kills the child process by sending `SIGKILL` signal. If child execution does not lead to any exceptional situations, then parent is not interrupted at all.

6. If execution of the test instruction does not lead to any exceptional situations, the child continues with the execution of the subsequent instruction which is our inserted trap instruction. When it is executed by the child, OS again sends `SIGTRAP` to the parent. This time though, parent reads register values via `PTRACE_GETREGS` and saves them as the end state. After reading the register values, it kills the child process as we do not need to continue with the child execution.

Our user-level process monitoring framework is implemented in 2000 lines of C code and 50 lines of shell script C code has some architecture-specific features, such as the use of trap instruction, but is mostly architecture-neutral. The shell script uses GNU assembler `gas`, `objdump`, and `objcopy` to encode a test program containing a given assembly instruction wrapped in two traps and obtain its binary encoding.

5.3.5 Handling memory

Since taking snapshots of process's memory using `ptrace()` is too expensive, we rely on accessing `/proc/<pid>/mem`, a file representing virtual memory of a process. Our snapshots have formats very similar to that of core files. Start states of memory also use the same format as that of snapshots. According to the start state, desired values are written to specified memory locations before executing a test program.

As compared to the assembly execution environment, our RTL interpreter deals with memory differently. Specifically, all load and store operations performed by the test RTL instruction are transformed into file I/O operations on the file representing start state of the memory. This is done to isolate memory of the RTL instruction from that of the interpreter, which has its own memory layout that is different than that of the RTL instruction. We could have also represented memory for an RTL instruction as a byte array, but such an approach would demand mapping between virtual memory addresses used by the RTL instruction to the corresponding addresses inside byte array. This would unnecessarily complicate the implementation. On the other hand, since we already had a file representing start state of the memory, transforming RTL instruction's memory accesses into file IO was an easier approach. Lastly, details of all load and store operations (exact address of

	SI	**MI**	**REE**
# of Mapping Rules	140	1132	150K
# of Test Cases	1056	5762	421,090
% of Useful Test Cases in `ArCheck`	92	85	79
% of Useful Test Cases in **Random testing**	64	59	52
Time To Run Test Cases in `ArCheck`	5 min 7 sec	31 min	1 day 6 hrs
Time To Run Test Cases in **Random testing**	4 min 10 sec	24 min	1 day 1 hrs

Figure 38: **Analysis of mapping rules obtained from GCC's x86 code generator logs and test cases generated for them**

Description		`ArCheck`			**Random Testing**		
		SI	**MI**	**REE**	**SI**	**MI**	**REE**
Classification of Semantic Differences	**D1**	25	26	28	3	10	11
	D2	4	4	6	2	3	1
	D3	1	1	1	0	0	1
	D4	1	3	4	0	1	1
	Total	**31**	**34**	**39**	**5**	**14**	**14**
# of New Bugs Found		1	1	1	0	0	1
# of Existing Bugs Found		15			7		

Figure 39: **Statistics of evaluating mapping rules obtained from GCC's x86 code generator**

the access, the contents) performed by IR are recorded and are eventually used to obtain memory semantics of an IR instruction.

5.3.6 Test execution and result comparison

Based on the observation that there is no dependency between the logged mapping rules, we have built a test execution system which can run multiple test cases in parallel. The degree of parallelism can be changed using a command line parameter. Test execution and result comparison are both implemented in C, and use about 700 lines of code.

5.4 Evaluation

We evaluated the effectiveness of our approach by testing models extracted from x86 code generator of GCC-4.5.1. Our current prototype supports only general-purpose and SSE x86 instructions. (Supporting the rest of the instruction set requires further engineering work on the RTL interpreter.) We conducted our experiments on 32-bit Linux running on a quad-core Intel Core i7 processor.

For comparison purposes, we implemented a random start state generation strategy. Note that a random strategy can be used to generate any number of start states, but to simplify comparisons, we generated the same number of start states as `ArCheck`.

The extracted model that we tested had 140 abstract rules corresponding to 140 unique mnemonics. These 140 mnemonics covered roughly 23% of total 32-bit x86 instructions. Breaking down these 140 mnemonics, we found that they covered 80 of around 200 general-purpose and 60 of around 70 SSE instructions. Of the remaining 120 general-purpose instructions, around 78 were system-related and I/O instructions, 12 were control-transfer[20] instructions, and 30 were not covered in the extracted model. Using 140 abstract rules, `ArCheck` generated 140 concrete rules in single-instance mode (**SI**), 1132 concrete rules in multiple-instances mode (**MI**), and 150K concrete rules in restricted exhaustive enumeration mode (**REE**). So that is abstract to concrete rule multiplier of 1, around 8, and around 100 for these three different modes respectively.

Following observations can be made from these results:

- Average number of generated test cases in three modes differs. `ArCheck` generated an average of 8, 5, and 3 test cases per instruction in **SI**, **MI**, **REE** modes respectively. The average number for **REE** mode is smaller because instructions, such as "`mov`" for x86, generate higher number of operand combinations but relatively lesser number of test cases than many other instructions.

- Numbers of useful test cases generated by `ArCheck` are significantly higher than those generated by random state generation approach. We call a test case "useful" if `ArCheck` completes a test run for it without raising exceptions. For instance, when `eax` is 0, `mov 0(%eax),%esp` leads to a null-pointer dereferencing exception.

  Using randomly generated start states, almost every memory related instruction ended up leading to an invalid memory access. `ArCheck` also produced a few useless test cases for some of the SSE instructions because the constraints involving floating point instructions are more complex than those on integer operations.

[20]We do not handle control-transfer instructions because they present a complication in restricting control-flow and regaining control back to the test framework. Nonetheless, we are definitely considering them in our future work.

Figure 39 compares ArCheck with random testing in terms of different categories of instances in which semantics of IR and assembly instructions do not match. Note that for this evaluation, we checked for both soundness and semantic equivalence. Unlike soundness check, semantic equivalence is a strict check: it demands that the semantics of an assembly instruction is strictly modeled by an IR instruction. We enforced both these checks because we wanted to understand the type and the number of instances which belong to both categories. Note that the numbers reported in the Figure 39 are for semantic equivalence check. We will discuss which of these differences are soundness violations shortly.

Following observations can be made from these results:

- The semantic differences found by all three modes of ArCheck are considerably more than those found by random testing. Note that we have grouped semantic differences by abstract rules, and have counted maximum of 1 difference per abstract rule. We did this in order to eliminate duplicate counting of same semantic difference for different instances of same abstract rule. So 31 differences that we found using **SI** mode of ArCheck means that 31 x86 abstract rules from the model were found having at least one semantic difference.

- Comparing results from **SI** and **MI** mode with those from **REE** mode, we can see that the first two modes found fairly comparable number of semantic differences in significantly less time. This suggests that most of the semantic differences manifest for most operand combinations. The results are quite different for random testing, where detection is a lot less reliable.

  Though **SI** and **MI** mode did well in terms of finding semantic differences, **REE** mode found the most in all tests. This suggests that testing a mapping pair with multiple operand combinations might help in finding more differences. On the other hand, **REE** mode also took considerably longer to finish its run. We found that one of the major reasons for the amount of time taken is the explosion caused by the immediate values. To eliminate this source of explosion, but also to exploit the advantage of instruction mode over others, in the future, we can think of a mode where the multiplier between abstract and concrete rules is restricted to value less than 100 but close to 100. We speculate that such a mode should be able to finish its test run lot faster than that of the **REE** mode, but at the same time, perform equally better as **REE** mode.

```
movzwl 8(%esp), %eax
(set (reg:HI ax)
 (mem:HI (plus:SI (reg:SI 7) (const_int 8))))
```

Figure 40: **Example of `movzwl` instruction and its RTL**

```
shrdl $16, %ebx, %eax
[(set (reg:SI ax)
 (ior:SI
  (ashiftrt:SI (reg:SI ax) (const_int 16))
  (ashift:SI (reg:SI bx)
   (minus:QI (const_int 32) (const_int 16)))))
(clobber (reg EFLAGS))])
```

Figure 41: **Example of `shrdl` instruction and its RTL**

- Unlike `ArCheck`, which detected the new bug in all three modes, only the **REE** mode of random testing detected this bug. We will discuss this point shortly.

Figure 39 shows that **REE** mode of `ArCheck` found the largest number (39) of semantic differences. We have classified these differences into 4 different categories:

- **D1:** *Imprecise modeling of* `EFLAGS`. This kind of difference arises frequently in GCC's x86 code generator when an RTL instruction does not capture precise bits of `EFLAGS` that are modified by the corresponding assembly instruction. For instance, the example of `add $2, %eax` discussed earlier falls into this category. This type of difference does not represent a soundness issue in the code generator because `(clobber (reg EFLAGS))` is actually an over-approximation of the instruction semantics.

- **D2:** *Incorrect value in the destination operand.* This kind of difference arises when an RTL instruction does not perform some operation, such as zeroing or sign-extending a value, that is performed by an assembly instruction. For instance, for `movzwl` instruction in the Figure 40, RTL simply moves lower 2 bytes of the source into destination, but fails to zero out upper 2 bytes of the destination. This kind of difference is a soundness violation.

- **D3:** *Incorrect operation in RTL.* This kind of difference arises when an RTL instruction performs different operation than the corresponding assembly instruction. This kind of difference represents a soundness violation. For instance, for `shrdl`

```
mull %ebx
```

```
[(set (reg:SI dx)
  (truncate:SI
    (lshiftrt:DI
      (mult:DI
        (zero_extend:DI (reg:SI ax))
        (zero_extend:DI (reg:SI bx)))
      (const_int 32))))
 (clobber (reg:SI ax))(clobber (reg EFLAGS))]
```

Figure 42: **Example of `mull` instruction and its RTL**

instruction in the Figure 41, RTL uses arithmetic shift operator (`ashiftrt`), whereas assembly instruction performs a logical shift (`lshiftrt`).

More specifically, semantics of `shrdl` instruction as per Intel manual is: "The instruction shifts the first operand (`eax`) to the right the number of bits specified by the third operand (count operand). The second operand (`ebx`) provides bits to shift in from the left (starting with the most significant bit of the destination operand)." The way RTL models this is by inclusive-or of arithmetically right-shifted destination and left-shifted source operand. Soundness issue shows up when the destination contains a negative value. Since arithmetically right-shifted destination will have top bits set to 1. Inclusive-or with such a value will then generate result with its top bits set to 1 instead of moving contents of source into the top bits of the destination. We detected this issue when we set, `eax` = 0xb72f60d0, `ebx` = 0xbfcbd2c8. Above `shrdl` instruction in that case produced 0x**d2c8**b72f in `eax`. But the corresponding RTL produced 0x**ffff**b72f in `eax`.

We reported this difference to GCC's bug reporting system. GCC developers have acknowledged that this is a bug, and have fixed it promptly. Details of our bug report can be found at [6].

- **D4:** *Update to a destination not specified.* This kind of difference arises when GCC uses some implicit assumptions not mentioned in the RTL specification. For instance, `mull %ebx` instruction shown in Figure 42 modifies register pair [`edx:eax`], where the top 4-bytes of the product are stored in `edx`, and lower 4-bytes of the product are stored in `eax`. But RTL for `mull` stores lower 4-bytes of the result in `edx`, and says `eax` is clobbered.

We found that GCC uses this instruction only when it is computing a modulo of a number (in other words, there is an implicit assumption that this instruction should only be used when the product cannot be more than 4-bytes long.) Since RTL for `mull` instruction captures over-approximation of the semantics, it does not represent a soundness violation. Nonetheless, use of implicit assumptions can possibly lead to soundness violations. Moreover, We believe that programming practices that rely on implicit assumptions lead to a number of latent bugs which are not uncovered easily. To achieve the objective of minimizing the number of bugs in compilers, one should strictly avoid such programming practices.

5.4.1 Detecting known soundness issues

As further evidence of the ability of `ArCheck` to identify code generator bugs, we used it on older version of GCC with known bugs in machine descriptions. (These were the bugs that have previously been reported to the GCC team and fixed.) Overall, we obtained a list of 15 soundness issues reported against x86 code generator in the last 4 years (July 2011 to December 2014). We consider an issue as a soundness related if semantics of assembly and IR instruction in the issue do not match. (This requires human knowledge about the target architecture, so we had to manually analyze the bug reports.) We specifically tested the mapping pairs involved in the issues, and verified that `ArCheck` is able to detect all the issues. Some of these issues are serious, e.g., missing update to `EFLAGS`, changing order of source and destination, not following RTL specification accurately, etc.

We will now discuss some of these bugs and the way we detected[21] them.

- The bug reported in [7] is about failing to model possible updates to `EFLAGS` by an execution of `sbbl` instruction. `ArCheck` detected this bug because the start state generator initialized start states to unset bits of `EFLAGS` (because as per RTL semantics `EFLAGS` neither affected the end result, nor `EFLAGS` were clobbered by it), but the result produced by the assembly execution system had those bits of `EFLAGS` set. This bug belongs to category **D2**. Out of 15 bugs we collected, 3 were of this type.

- One older bug [4] (older than 4 years) detected by `ArCheck` was about an incorrect modeling of `movsd` SSE2 instruction. This instruction operates on two `XMM` registers

[21]Though our implementation does not support all of x86 instructions, to detect these bugs, we added support for the assembly and RTL instructions from the buggy mapping rules. Specifically, we had to add support to detect 6 of the 15 issues.

and moves the lower 64-bits (double precision floating point number) of the source register to the lower 64-bits of the destination, and preserves the upper 64-bits of the destination. The issue was an incorrect use of RTL's `vec_merge` operator because of which exactly opposite semantics (assign the upper 64-bits of the source to the upper 64-bits of the destination, and preserve the lower 64-bits of the destination) was modeled. We detected this bug because `ArCheck` initialized the source and the destination `XMM` register with different values (because instructions which move values from a source to a destination have no "interesting outputs", so the state generator initializes the source and the destination with different byte patterns such as all bits set to 0 in the source and all bits set to 1 in the destination or vice versa[22]), and the result produced by the RTL interpreter was different than the one produced by the assembly execution system. This bug is a representative of the category **D3**. The list of 15 bugs had a couple of this type.

- An interesting type of bugs detected by `ArCheck` is syntactic bugs and syntactic errors which lead to soundness violations. For instance, the bug reported in [1] is about an incorrect order of operands in `bextr` assembly instruction generated by GCC's x86 code generator. Specifically, `bextr` instruction takes 3 operands of which first and third can only be registers, while the middle one can be a register or memory operand. Modelling of this instruction in x86 machine description was incorrect — it allowed first operand to be either a register or memory. We detected this bug when we attempted to assemble the assembly instruction using `as`. Although, this particular bug is not a soundness violation, and `ArCheck` is not designed to detect these type of bugs, some syntactic errors can lead to soundness violations. For instance, the bug reported in [9] is about missed square brackets around an immediate constant which changed the semantics of an instruction from moving a value from a memory location to moving an immediate value. Although the particular bug in [9] is a x86-64 bit bug, it is easy to see that it is indeed a soundness violation. The list of 15 bugs had 1 bug of this type.

Finally, we believe that the presence of these bugs in the bug reports indicates that the errors were not caught by the end-to-end testing used in GCC, thus establishing the need for dedicated frameworks such as `ArCheck`.

[22]Rationale behind such assignment is being able to detect incorrect move of a single bit.

Given the experimental results of `ArCheck`, an important question is: *what should be the ratio of abstract-to-concrete mapping rules so that we achieve very high confidence in the soundness of extracted semantics model?* Note that, theoretically, to achieve very high confidence, one should test every abstract rule from the model with all possible operand combinations. On the other hand, experimentally, we found that **REE** mode took considerably more time but found very little additional differences than those found by **MI** mode. So we think that ratio close to that of **MI** mode is good enough to achieve high-level of confidence in the soundness of the extracted models.

5.5 Related Work

Compiler testing, verification, and bug finding. Compiler testing has been an active area of research for several decades. One of the earlier works, which described use of machine descriptions for compiler testing, is [79]. It describes an interesting approach to compiler testing by checking equivalence of source program and its object code by translating both to a common intermediate language. Translation of object code to intermediate language is done by relying on a set of procedures which encode the semantics of low-level instructions in terms of intermediate language instructions.

More recently, by randomly generating valid C programs, the CSmith system [97] performed a systematic end-to-end testing of modern compilers such as GCC and LLVM and found a number of bugs in them. Along with CSmith, other works such as [54, 83, 59, 99] have applied the technique of automatically generating valid C/C++ programs to test compilers. A fundamental difference between `ArCheck` and all these works is the targeted testing of code generators performed by `ArCheck`. In contrast, systems such as CSmith are targeted at testing all components of the compiler. Unfortunately, end-to-end testing of compilers — a very common practice used by modern compilers — does not help in testing individual components of the compiler thoroughly. This is corroborated by findings of CSmith as well. Tools such as `ArCheck` are thus complementary, and serve to provide much more in-depth testing of individual components of the compiler.

Compiler verification has also been a prominent area for compiler research with techniques such as *certified compiler* [53, 52] and *translation validation* [65, **?**, 71] being developed. CompCert [53, 52] has been a popular compiler verification work with promising results. However, scaling formal verification to production compilers such as GCC remains a challenge. While recent work has made significant progress in tackling components of production compilers, e.g., mem2reg optimization in LLVM [100], scaling these to components of the size of code generator represents a significant challenge. Testing-based approaches such as `ArCheck` thus represent an important complementary approach that can work on industry-strength compilers.

Given that verification of a complete compiler is difficult, Translation Validation tries to check the correctness of each of the compiler passes. The correctness of a pass is checked by comparing the semantics of the program being compiled and its semantics after each pass. Such checking also helps in pinpointing the compilation errors and thus helps with compiler debugging and testing. A common approach in Translation Validation is to first check the equivalence of control flow graph of input program and that after each pass. Once both graphs are checked for equality, symbolic evaluation along with constraint solver is used to check the correctness of input program and the output of each pass. Unfortunately, Translation Validation of a code generator is infeasible since the output of a code generator is a sequence of assembly instructions. And to prove the equivalence of input source program with the list of assembly instructions, one needs to convert the assembly instructions in some intermediate language. And this demands manual modeling which is exactly the source of semantic bugs in code generators.

Whereas `ArCheck` is focused on the correctness of IR to assembly translations, the work of Fernández and Ramsey [**?**] targets the correctness of assembly to machine-code translations. They utilize a language called SLED (Specification Language for Encoding and Decoding) to specify instruction encodings at a high-level, and to associate assembly and machine code instructions. These specifications can then be used to generate machine code from assembly, or vice-versa. The main focus of their work is that of correctness checking of the mappings specified in SLED. This is accomplished using a combination of static checks and by comparing SLED-based translations with the translations produced by another well-tested independent tool such as the system assembler.

Approaches for test case generation. Generating better test inputs by improving test case generation strategies has been an area of active research. One can broadly classify existing testing strategies into black-box and white-box. Black-box testing strategies such as fuzz testing (random testing) [61] and grammar-based fuzz testing [58, 56], can also be applied for testing code generators. A drawback, however, is that it is difficult to ensure that all "relevant" and/or "interesting" input values have been tested. In contrast, `ArCheck`has been explicitly designed to leverage the semantics of IR, and intuition and insight of human experts, to generate relevant/interesting test cases.

White-box testing strategies, such as symbolic execution [49, 19, 39], on the other hand, generate more "interesting" inputs because they treat the system-under-test as a white-box. Symbolic execution, in particular, seems best suited for our problem since it is a coverage testing technique; checking soundness of code generator mapping rules is a coverage testing problem. Moreover, symbolic execution has recently received a lot of research attention, and it has led to improvements in the symbolic execution systems. Unfortunately, it still suffers from the problem of explosion in the number of branches while handling complex code. Fortunately, IR instructions that we want to analyze are simple enough that such problems will not arise. Nonetheless, given the simplicity of white-box static analysis of IR instructions, we preferred it over symbolic execution for building robust tools that can operate on production compilers.

5.6 Summary

In this section, we developed a new, efficient and practical approach for systematic testing of semantic models extracted from code generators of modern compilers. Our approach can also be considered as a way to check the correctness of compiler code generators. One of the key contributions of our work is the development of an architecture-neutral testing technique. Another important benefit of our approach is that it treats the compiler/code-generator as a black-box, and hence can be easily applied to any compiler. A third major benefit is that it not only detects bugs, but also makes it easy to locate/diagnose them.

Our evaluation showed that `ArCheck` can identify a significant number of bugs and inconsistencies in architecture specifications used by GCC's code generator. Specifically, we used `ArCheck` on approximately 140 unique x86 instructions and identified potential inconsistencies in 39 of them. Although a majority of these are not soundness related, we believe that approximately 7 are, including one that has already been fixed by GCC developers. Moreover, we verified that `ArCheck`is able to detect 15 other known bugs (soundness issues) in the previous versions of GCC. Some of these bugs are serious, and their presence in the bug reports indicates that the errors were not caught by the end-to-end testing used in GCC, thus establishing the need for dedicated frameworks such as `ArCheck`. These results demonstrate the utility of tools such as `ArCheck`, and suggest that similar tools should be integrated into the test cycle of today's compilers.

6 Building assembly-to-IR translators automatically

Recall from our discussion in Section 1 that the existing binary analysis, translation, and instrumentation systems such as Valgrind and QEMU manually build their assembly-to-IR translators. Another common approach is to develop architecture-neutral translators and drive them using the manually-written architecture specifications. Because `LISC` and `EISSEC` help us in extracting semantic models automatically, in this section, we demonstrate an application of such models by using them to build assembly-to-IR translators automatically.

Before we describe how to use the extracted models for building the translators, we will describe a typical design of binary analysis, translation, and instrumentation systems.

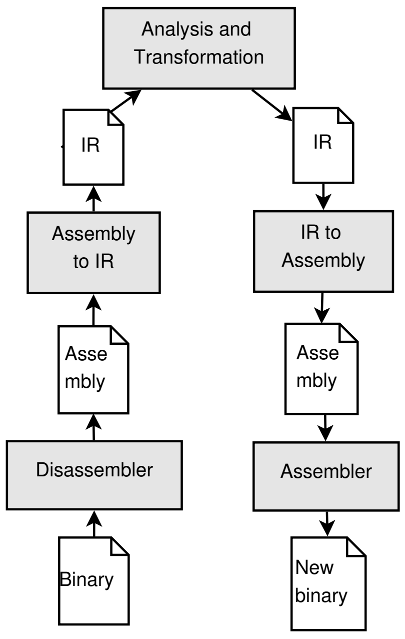


Figure 43: **Process of architecture-independent binary analysis, binary translation, and binary instrumentation using IR**

Figure 43 shows a typical design of binary translation and binary instrumentation systems. The process of binary translation or instrumentation starts with accepting an input binary and ends with generating a new binary (In binary translation, a new binary may not be generated. Instead, the generated IR may be interpreted.) Binary analysis systems, on the other hand, do not generate new binaries. They instead analyze the binaries (by analyzing the IR), produce appropriate analysis results, and terminate the process at that point. These systems, in other words, do not have the second part of the framework which translates IR into a new binary. Below, we describe the various components of these systems.

1. *Disassembler.*

   Program binaries, before they are analyzed or transformed, need to be disassembled. Disassemblers handle the low-level architectural details, such as object file formats (e.g., COFF [60], ELF [88], etc), encoding of machine instructions, etc, from the input binaries and effectively hide all such details from the higher-level components in the system. An output of a disassembler is a list of assembly instructions corresponding to the machine instructions from the input binaries[23].

2. *Assembly-to-IR translator.*

   Once the binaries are disassembled, the next step is to translate the list of assembly instructions into the list of IR instructions. The rationale behind such translation is architecture-neutrality, which enables the analysis systems to[24] support binaries from different architectures.

   A naive approach to implement an assembly-to-IR translator is to manually encode the assembly-to-IR translation rules in the translator code directly. Unfortunately, this approach makes the translator architecture-specific. A better approach is to develop an architecture-neutral translator, which refers to the architecture-specific translation rules. Architecture-specific rules, in such a case, can be written in the form of an architecture specification. Such a design of the translator, which refers to the manually-written architecture specifications, is common among the developers of binary analysis, binary translation and binary instrumentation systems [25, 67, 15, 35, 11, 27, 14, 48]. Instead of relying on the manually-written specifications, we develop an approach to use the extracted models as specifications.

[23]An accurate disassembly of program binaries is in itself a research problem with a number of solutions such as BinCFI [98] being proposed. We will not discuss the disassembly problem and its solution here, but will assume that there is a disassembler which provides us with an accurate disassembly of binaries.

[24]The IR instructions produced by an assembly-to-IR translator may not be in a form which can be directly used for analysis and instrumentation. The generated IR instructions will mostly likely be semantically close to the assembly instructions. Because the IR instructions are semantically close to the assembly instructions and do not capture high-level program semantics (such as variables and types), they may be of less value to the architecture-neutral binary analysis and instrumentation tools. To address this issue, low-level analysis passes (not shown in the figure) are run on the IR instructions. Because program analysis to extract high-level program semantics is a separate research topic, we do not talk about these analysis in this dissertation.

3. *Analysis and transformation.* In this step, any desired program analysis and transformation passes work on the semantically-rich program representation (IR) of the input binary. An essential advantage of building an architecture-neutral system is that the program analyses or transformations can be architecture-neutral and thus applicable to any architecture. This saves the effort of implementing the analyses or transformations for every architecture.

4. *IR-to-assembly translator + assembler.* This is the last step in which the list of IR instructions is translated into a list of assembly instructions, and a new binary is generated by assembling those assembly instructions. Commonly this is done by relying on the specifications used by the assembly-to-IR translators [27]. The functionality of this stage is similar to that of a combination of a compiler backend and an assembler.

We now describe our approach to use the extracted models for building an assembly-to-IR translator automatically.

6.1 Our Approach to Assembly to IR Translation

An input for the assembly-to-IR translation is the list of assembly instructions obtained by disassembling the input binary. One can obtain such a list using a disassembler (e.g., `objdump`).

6.1.1 Challenge

To translate a list of assembly instructions into a list of IR instructions, an assembly-to-IR translator needs to solve an interesting algorithmic challenge. The challenge stems because a sequence of assembly instructions may be translated to a single IR instruction. This leads to the question of how many assembly instructions should be grouped together and translated into IR. For instance, if the input assembly list contains four assembly instructions, $\{a_1, a_2, a_3, a_4\}$, then there exists following different ways of partitioning them, where each partition is translated into a single IR instruction:

1. All partitions of size 1:
    - (1) $\{a_1\}$, $\{a_2\}$, $\{a_3\}$, $\{a_4\}$,

2. Partitioning with at least one partition of size 2:
   - (2) $\{a_1, a_2\}$, $\{a_3\}$, $\{a_4\}$,
   - (3) $\{a_1\}$, $\{a_2, a_3\}$, $\{a_4\}$,
   - (4) $\{a_1\}$, $\{a_2\}$, $\{a_3, a_4\}$,
   - (5) $\{a_1, a_2\}$, $\{a_3, a_4\}$,
3. Partitioning with at least one partition of size 3:
   - (6) $\{a_1, a_2, a_3\}$, $\{a_4\}$,
   - (7) $\{a_1\}$, $\{a_2, a_3, a_4\}$,
4. Partitioning with at least one partition of size 4:
   - (8) $\{a_1, a_2, a_3, a_4\}$

Thus there are 8 different ways in which one can partition the list of four assembly instructions for translating to IR. Which of these different ways should one use for translation? Note that all of the partitioning schemes may not be valid for a given list of assembly instructions. To be precise, for all 8 ways (mentioned above) to be valid for the translation, all assembly sequences used by them must be present in the extracted semantics model. For instance, for the 7th partitioning to be valid, the extracted model must have two rules: one for the single assembly instruction a_1 and another for three assembly instructions $\{a_2, a_3, a_4\}$. If one of these rules does not exist, then we can safely discard that partitioning. This observation can be applied to eliminate invalid ways of partitioning. Unfortunately, given the exponential number of possible partitioning schemes (valid and invalid), a simple algorithm could run into worst-case of exponential time complexity. Fortunately, given that different partitioning schemes of a given list of assembly instructions represent overlapping partitions, we can develop an efficient dynamic programming algorithm. The overlapping partitions of above example is shown in Figure 44. Note that a path in the tree corresponds to either a valid or invalid partitioning scheme. There could be multiple paths corresponding to the valid schemes, and we just want to find any one path which corresponds to a valid scheme. In the figure, partitions that overlap are shown in the same color. For instance, partitions $\{a_3, a_4\}$, $\{a_3\}$ and $\{a_4\}$ overlap same partitions across multiple paths. If we observe carefully, then all of these overlapping partitions correspond to possible partitioning schemes of assembly sequence $\{a_3, a_4\}$. Once we solve the problem of finding

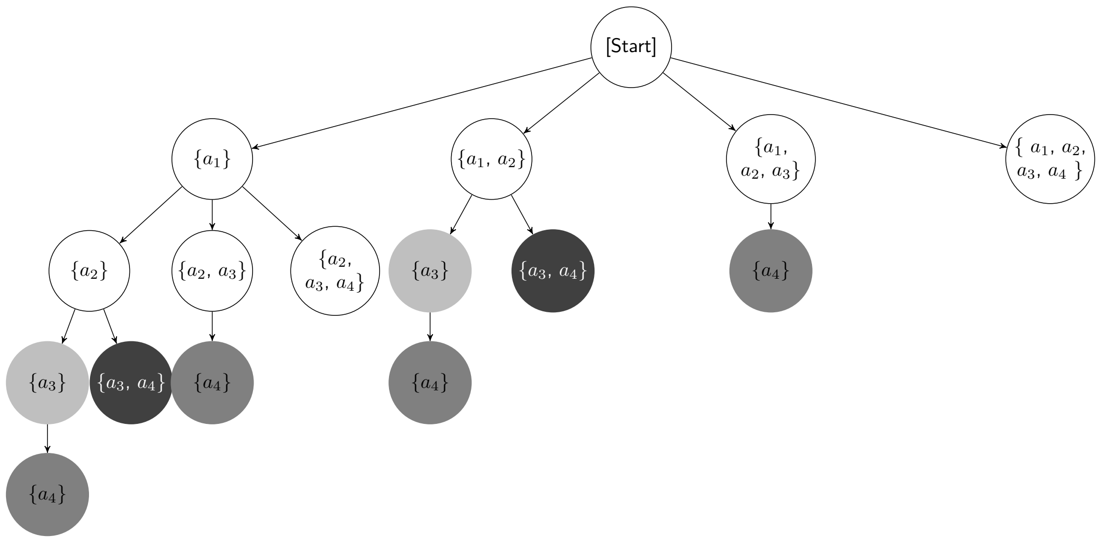


Figure 44: **Tree of various possible partitioning schemes of sequence $\{a_1, a_2, a_3, a_4\}$**

a valid partitioning scheme of $\{a_3, a_4\}$, we do not need to solve it again. We will exploit this overlapping sub-problem property of the assembly-to-IR translation problem in order to develop an efficient dynamic programming algorithm.

In order to develop a dynamic programming algorithm, we will use few notations and then develop a recursive solution to the problem. The recursive solution will essentially capture the overlapping sub-problem nature. We will first denote the list of assembly instructions by A, and the particular element i of A by A_i. Thus, $A_1 = a_1$ in the above example. To represent a partition of A consisting of all elements from i to j, with both A_i and A_j included, we will use notation $P_{i,j}$. Thus, in the above example, $P_{2,4} = \{a_2, a_3, a_4\}$. When i and j are equal, $P_{i,j}$ represents a single element. We will denote the semantic model used for assembly-to-IR translation by $\mathscr{M}$. Using the model to obtain a list of IR instructions for a list of assembly instructions (A) can be represented as $\mathscr{M}[A]$. Note that A here could be a single assembly instruction or a sequence of them. Also, the model may not have the IR instructions corresponding to some value of A. In that case $\mathscr{M}[A] = \phi$. Finally, the notion that there is some valid partitioning scheme for a partition $P_{i,j}$ is represented by solution $S_{i,j}$. Intuitively, in the tree of possible partitioning schemes, $S_{i,j}$ corresponds to one of the all possible valid paths from node A_i to node A_j. If there is no valid partitioning scheme for $P_{i,j}$ then $S_{i,j}$ = []. Concretely, we can define $S_{i,j}$ as:

$$S_{i,j} = \begin{cases} \mathscr{M}[P_{i,j}], & \text{if } i = j \\ \mathscr{M}[P_{i,i}] \ \ \& \ S_{i+1,j} \mid & \\ \mathscr{M}[P_{i,i+1}] \ \& \ S_{i+2,j} \mid & \\ \cdots & \\ \mathscr{M}[P_{i,j-1}] \ \& \ S_{j,j} \mid & \\ \mathscr{M}[P_{i,j}], & \text{Otherwise.} \end{cases}$$

Intuitively, above equation captures the observation that the solution to a list of instructions constitutes any of the following: either a translation for its first element and the solution for the remaining elements in the list, or a translation for the partition obtained from first two elements and the solution for the remaining elements in the list, and continuing this way till we get the translation for whole sequence directly by treating that list as a single partition. Note that the disjunction above represents the fact that there could be multiple paths in the tree of possible partitioning schemes. Also, $S_{i,j}$ when i is equal to j is same as $\mathscr{M}[P_{i,i}]$. To illustrate above equation, for $\{a, b, c, d\}$, $S_{1,4}$ is:

$$\begin{aligned} S_{1,4} \ = \ & \mathscr{M}[\{a\}] \quad \& \ S_{2,4} \mid \\ & \mathscr{M}[\{a,b\}] \ \ \& \ S_{3,4} \mid \\ & \mathscr{M}[\{a,b,c\}] \ \& \ S_{4,4} \mid \\ & \mathscr{M}[\{a,b,c,d\}] \end{aligned}$$

This equation exactly captures the overlapping sub-problem nature of the assembly-to-IR translation problem. This is because in order to find $S_{1,4}$ we need to find $S_{2,4}$, which requires finding $S_{3,4}$. Specifically, for above example,

$$\begin{aligned} S_{2,4} \;=\; & \mathscr{M}[\{b\}] \;\;\&\; S_{3,4} \mid \\ & \mathscr{M}[\{b,c\}] \;\&\; S_{4,4} \mid \\ & \mathscr{M}[\{b,c,d\}] \end{aligned}$$

The dynamic programming algorithm to find the translation of an input assembly list thus solves these equations recursively, storing the solutions to sub-problems to avoid solving them again.

6.1.2 Algorithm

Dynamic programming algorithm for assembly-to-IR translation which incorporates the approach discussed above is shown in Figure 45. Given a list of disassembled assembly instructions (represented by A), the partition $P_{i,j}$ of A identified by the start (i) and end (j) index in A, and the semantic model $\mathscr{M}$, the algorithm produces a list of IR instructions (I) for A. $S_{i,j}$ from the equations above is thus a list of IR instructions I corresponding to $P_{i,j}$. Typically, first invocation of $Translate$ would look like $Translate(A,\ 1,\ |A|,\ \mathscr{M})$, where $|A|$ represents the total number of assembly instructions in A. The algorithm follows our earlier discussion and solves the problems recursively. Solutions to sub-problems is memoized in a table T. $T_{i,j}$ represents a solution (list of IR instructions) corresponding to $P_{i,j}$. Note that the algorithm stores the information when some sequence (or sub-sequence) of assembly instructions does not have a solution. This is done in line 19. Storing this information essentially helps algorithm in avoiding the exploration of sub-partitions for a visited partition. Lines 5–8 check if the solution to the problem already exists, and if it does, then return it directly without proceeding any further. If the solution does not exist, then the algorithm explores all partitions of increasing size until a solution is found. If it finds a solution to any of the partition sizes, then it is returned immediately. @ operator at line 16 is simply used to represent concatenation of an IR list for $P_{i,k}$ and I'.

To demonstrate partitioning decisions made, let us go back to our initial example where $A = \{a_1,\ a_2,\ a_3,\ a_4\}$. Let us say that the solution to $P_{1,4}$ is to partition A such that $\{a_1,\ a_2,\ a_3\}$ is one partition and $\{a_4\}$ is another. The tree of invocations of $Translate$ before arriving at the solution is shown in Figure 46. A node in the tree captures the value of

```
1.  // $A$ is the list of all assembly instructions to be translated to IR.
2.  // $i$ and $j$ define the partition $P_{i,j}$ of $A$ which we need to translate.
3.  // Algorithm returns, $I$, the list of all IR instructions for instructions of $P_{i,j}$.
4.  **algorithm**: *Translate*($A$, $i$, $j$, $\mathscr{M}$):
5.  **if** $T_{i,j} \neq \phi$      // $T$ is a table used to store solutions to sub-problems.
6.  **then**
7.     **return** $T_{i,j}$
8.  **fi**
9.  **for** $k = i$ to $j$
10. **do**
11.    **if** $\mathscr{M}[P_{i,k}] \neq \phi$      // If a mapping pair exists in the model.
12.    **then**
13.        $I'$ = *Translate*($A$, $k+1$, $j$, $\mathscr{M}$)
14.        **if** $I' \neq []$
15.        **then**
16.            $T_{i,j} = \mathscr{M}[P_{i,k}]$ @ $I'$
17.            **return** $T_{i,j}$
18.        **else**
19.            $T_{j+1,k} = I'$
20.        **fi**
21.    **fi**
22.    // May be the current way of splitting assembly list is not correct.
23.    // Let's increase the partition size by 1.
24.    $k = k+1$
25. **done**
26. // Mapping rule for assembly instructions from $P_{i,j}$ is not in $\mathscr{M}$ at all.
27. // We cannot lift this assembly sequence.
28. **return** []
```

Figure 45: **Algorithm for assembly-to-IR translation**

$P_{i,k}$ accepted in various invocations of $Translate$, and the values of $k+1$ and j used in the corresponding invocation. Notice that siblings represent successive iterations of **for** loop, where as a path (towards a leaf) leaf represents recursive invocations made to explore a solution. In the figure, the solution nodes are shown in Gray, where as the nodes which are not explored (because of memoization) are shown in Dark Gray. Also, notice that a sub-partition is visited only once, which allows us to find the worst-case complexity of the algorithm easily.

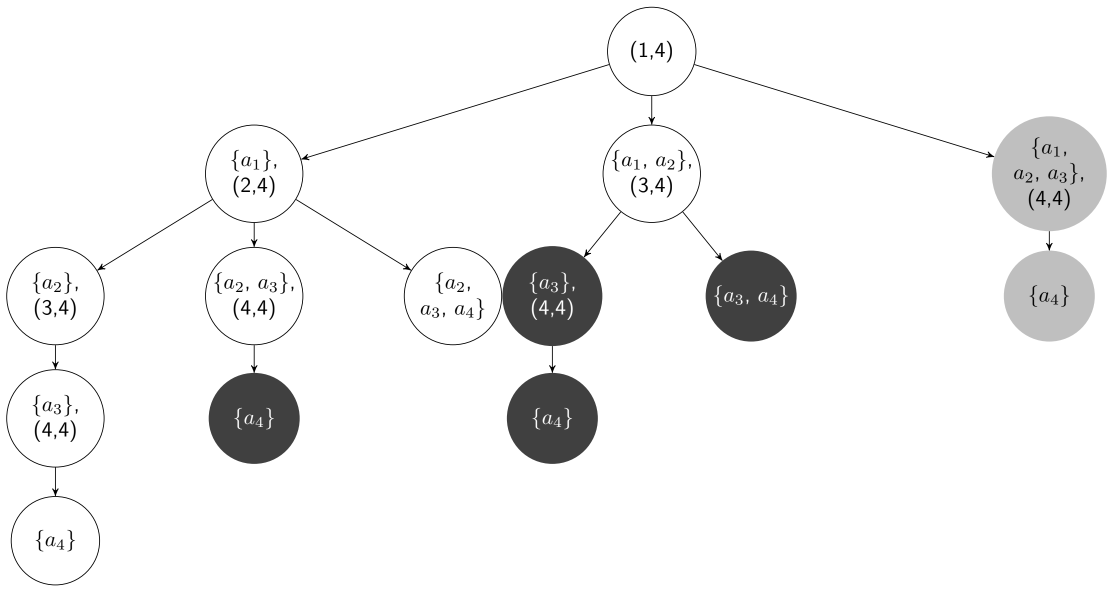


Figure 46: **Decision tree for assembly to IR translation**

Since the algorithm avoids exploring subpartitions of an already visited partition, the worst case complexity of the algorithm is bounded by the total number of partitions of a given list of assembly instructions. For a list of size N, given that partition size can vary from 1 to N, the total number of partitions can be obtained by the formula:

$$1+2+..+N = \frac{N \times (N+1))}{2}$$

The worst case time complexity of the algorithm is thus $O(N^2)$.

While the algorithm described above is a generic one, we found that in practice we do not require full generality. Specifically, instead of allowing a maximum partition size to be equal to $|A|$, we restrict it to 4 for x86, ARM and AVR architectures. This is because we found that on these architectures, a single IR instruction can map to at most 4 assembly instructions. This is implemented by simply modifying the condition of **for** loop as: $k = i$ to $min(i+4, j)$. Because the algorithm that we implemented using constant value for a maximum partition size, its worst-case time complexity is $O(N)$.

6.1.3 Other challenges for Assembly to IR translation

Besides the algorithmic challenge discussed earlier, there exists a few other challenges for the assembly-to-IR translation.

Soundness of using extracted semantics model for assembly-to-IR translation. The first challenge relates to the *soundness* of our approach. Specifically, the question is why is it sound to use the extracted models for assembly-to-IR translation.

Code generators use extracted models in the forward direction to translate IR to assembly, so a natural question is whether it is sound to use them backwards. There are several reasons to believe that they are sound for translating assembly to IR. The first concerns the way in which code generators (architecture specifications) are developed. The developer starts by enumerating instructions in the target architecture, and specifying the semantics of those instructions in IR. The developer view is one that corresponds to an assembly to IR mapping. Second, note that the IR-to-assembly mapping must be sound, or else the code generator will likely generate incorrect code. Therefore, an assembly instruction must perform all of the actions that are included in IR. This suggests that perhaps the assembly instruction could do more than what was asked for by the IR, e.g., change an additional register. If so, assembly-to-IR translation does not capture these extra actions. However, consider the fact that the IR optimizer performs several optimizations such as removal of redundant computations and reordering of code snippets. These would be unsound if we allowed an IR snippet to be replaced by an assembly instruction that modified CPU state in ways beyond what was stated in IR.

In addition to above discussion, we checked the correctness of the models using `ArCheck` semantic equivalence test. Because every assembly-to-IR pair from such models is semantically equivalent, the assembly-to-IR translation will always be sound.

Syntactic differences between the output of a compiler and a disassembler. Another challenge faced by the translator is the different ways of printing the instruction mnemonics. For instance, the GCC compiler does not print the suffix of instruction mnemonics, such as the ‘l’ suffix for 32-bit x86 `mov` instruction. The `objdump` disassembler, on the other hand, prints the suffix. These minute differences create challenges for the translator because the assembly-to-IR translation rules will not match in these cases. To handle such issues, we introduce a post-disassembly step to process the assembly instructions and to rectify all such differences. We have not shown this step in the figure for the architecture-independent binary analysis and transformation system, because it is not a standard step in these systems.

Handling control-flow transfers. A control-flow graph (CFG) is one of the program properties central to most program analyses. Thus, to allow a user of our translator to perform sophisticated static analyses and instrumentations on the IR, we must reconstruct the program's CFG from the list of IR instructions produced by our translator.

We construct the CFG by processing the list of IR instructions produced by the translator. Our approach to link control-flow sources and their targets is as follows. First, we use the runtime address of the assembly instructions (or their offset, in case of PIC code) produced by a disassembler to generate unique code labels. To be specific, an instruction's address is prefixed by "L_" to produce a unique string label for that instruction. Once this is done, we update the places where these instruction addresses are used and use labels instead of concrete memory addresses. This is done by identifying these addresses using pattern matching and replacing them by their labels. This takes care of fixing the targets of jumps and linking their sources to the targets. Unfortunately, function calls introduce additional complication. On Unix-based systems, external function calls are performed using the GOT and PLT tables. As a result, the address of the call operand printed by a disassembler is that of an address of the PLT stub for that function. Fortunately, the disassemblers also print the name of the function being called in the disassembly. We exploit this observation to replace the target of a call instruction with the actual function being called. Internal function calls do not need special treatment, because the address of their callee is also present in the assembly instruction, and the approach of handling jump targets can be easily applied in such case.

6.2 Implementation

The implementation of our assembly-to-IR translator comprises approximately 400 lines of OCaml code and 140 lines of shell scripts. OCaml code implements the algorithm and translates the list of assembly instructions. The scripts disassemble binaries, and implement post-disassembly steps such as processing the assembly instructions and fixing the instruction labels.

In addition to above code to translate a binary into a list of IR instructions, we implement a parallel translation framework (around 110 lines of shell script) to perform batch

Parameter	**x86**	**ARM**	**AVR**
Total number of binaries	9237	7326	12
Avg # of insns per binary	74K	117K	30K
Number of binaries translated in parallel	24	24	12
Total translation time (including disassembly)	7 hrs 10 mins	8 hrs 30 mins	34 secs
Avg translation time per binary	1 min 8 secs	1 min 36 secs	34 secs
Avg disassembly time per binary	16 secs	23 secs	7 secs
Total avg virtual memory consumed by the translator	245 MB	181 MB	32 MB

Figure 47: **Details of Assembly to IR translator performance**

translations. The framework divides the list of input binaries among the available number of workers and keeps track of progress. We have utilized this parallel translation framework to translate all x86 and ARM binaries on standard desktop OSes.

6.3 Evaluation

Translation time and memory consumption. The translation algorithm given in this section is general and does not put any restrictions on the highest length of consecutive assembly instructions that are translated to a single IR. However as we mentioned earlier, for x86, ARM and AVR, we have fixed the maximum sequence size to be 4. With this setting, we translated all the x86 binaries on Ubuntu-14.04 desktop OS, all the ARM binaries on debian-7.8.0, and the `coreutils-2.22` binaries for AVR. The translations were performed on a 32-bit Intel i7 CPU running Ubuntu-14.04. The disassemblers used were `objdump-2.24` for x86, `arm-linux-gnueabi-objdump-2.24` for ARM, and `avr-objdump-2.23.1` for AVR[25].

Figure 47 summarizes the translation time and memory consumption for x86, ARM, and AVR translations performed using the `LISC` generated semantic models. We exploited our parallel translation framework to translate multiple binaries simultaneously. As we found, our translator translates approximately one binary per minute. The total translation time includes the disassembly time and the time to translate a list of disassembled assembly instructions. Although disassembly is fast, model (extracted semantics model same as `LISC` transducer) loading takes time. For instance, it takes around 2.76 seconds to load the x86

[25]On Ubuntu-14.04, one can simply install `binutils-arm-linux-gnueabi` package for ARM, and `binutils-avr` and `gcc-avr` package for AVR to get them.

model of size 192MB in memory. To avoid loading of the model for the translation of every binary, our parallel execution framework loads the model in memory only once, and then forks its execution for parallel translations. Time to load automata in-memory is thus offset by the parallelization. The amount of memory taken by the translator is also reasonably small as compared to the RAM available on modern desktop systems.

Correctness. One of the important evaluation criteria for an assembly-to-IR translator is the correctness of the translation. Specifically, how many of the assembly-to-IR translations performed by the translator are correct. Since we used the translation algorithm given in this section for the evaluation of `LISC` and `EISSEC`, we would like to point an interested reader to the evaluation sections of these sections.

In addition to the correctness test based on semantic equivalence, we use a *loop-back* test to check the correctness of translating a list of assembly instructions into a list of IR instructions. The loop-back test consists of three steps: (1) applying the extracted models to translate a list of assembly instructions L to IR, (2) then running the compiler (GCC in our case) with the IR as input, and (3) verifying that the compiler produces the exact same list of assembly instructions as L. If GCC cannot translate an IR instruction — possibly because of the manually modeled mapping rules — then we perform such a translation ourselves and emit the corresponding assembly instruction(s) modeled in the matching rule. All the binaries used for experiments in this section passed on the loop-back test.

6.4 Related Work

All of the previous approaches to translate low-level instructions into high-level IRs are based on manually building the components needed for such translation. To further classify, one can make two categories of existing approaches. Approaches falling into the first category require a hand-written target instruction specification to drive the translator. The approaches that fall in this category are [25, 67, 15, 35, 11, 27, 14, 48]. For instance, SecondWrite [11] requires the XML specification of the architecture instructions, whereas UQBT [27] has designed its own format for such specification.

Approaches falling into second category make a clever use of existing systems such as QEMU and Valgrind. QEMU translates a source architecture instruction into the target. All existing architectures such as x86, ARM, etc have their own backend written for QEMU which takes instruction written in QEMU IR and translates it into target instruction.

Approaches [23, 45] have written a LLVM backend for QEMU, i.e., a backend to convert QEMU IR into LLVM IR. While approaches [18] directly use Valgrind's assembly to IR translator as it is.

One important drawback of all existing approaches is that they require manual effort for build the translators. Approaches falling in the second category suffer from their dependence on QEMU and Valgrind. QEMU and Valgrind's lack of support for some architectures such as AVR restricts application of these approaches to those architectures also.

DisIRer [46], on the other hand, is an approach similar to ours in that it relies on GCC to build assembly-to-IR translators automatically. Unfortunately, their approach relies on using architecture specifications (machine description files in GCC) to build inverse mapping table for assembly-to-IR translators. Their approach is thus specific to GCC and may also be specific to a particular version of GCC. `LISC`, on the other hand, is a compiler-neutral approach. We believe that compiler-neutrality helps `LISC` in avoiding issues such as changes between different versions of a compiler or different internal details across multiple compilers.

6.5 Summary

In this section, we described our approach to use the extracted semantic models to build assembly-to-IR translators automatically. Assembly-to-IR translators need to address a challenge of understanding how to divide the input list of assembly instructions for translation. This challenge arises because an IR instruction could map to a single assembly instruction or a sequence of them. We described our algorithm to address this challenge. In addition to the challenge of selecting assembly instructions for translation, we also discussed various other challenges in lifting input binaries to compiler IR. We believe that our evaluation consisting of models for multiple architectures and the input set of all binaries found on desktop OSes validate the effectiveness of our approach to build assembly-to-IR translators automatically. More importantly, we believe that it demonstrates the effectiveness of our approach to reduce the manual effort needed in building these translators.

7 Conclusion and Future Work

Extraction of instruction-set semantic models is one of the common problems faced by a number of software systems (such as system emulators, virtual machine monitors, etc). Unfortunately, the common approach of a manual model extraction reduces applicability and effectiveness of these systems. This dissertation proposes two approaches (`LISC` and `EISSEC`) to address this problem by automatically extracting instruction-set semantic models using code generators in modern compilers. To the best of our knowledge, there does not exist any research work which attempts the automatic semantic extraction using modern compilers. Automatically-extracted models enable these systems to support new architectures quickly, and saves time, money and human efforts required in porting these systems. The approach of automatic extraction thus leads to better software engineering practices.

In terms of contributions of the approaches, `LISC` demonstrates that the code generators of modern compilers contain architecture-specific knowledge, and that one can extract such knowledge by learning it from the code generator logs. More importantly, it demonstrates that one can learn the model for a new architecture quickly, and that such an approach can reduce the manual effort required in supporting new architectures. By evaluating against all the binaries (for x86 and ARM) found on commodity OS distributions, `LISC` demonstrates that the approach is generic and can deal with the complexities of modern architectures.

`EISSEC`, on the other hand, demonstrates that one can perform the symbolic execution of compiler code generators and extract their function (thus extracting complete instruction-set semantic models). Symbolic execution is a popular technique that is mostly applied in the context of test case generation and bug finding. By extracting the function of a code generator, we believe that `EISSEC` demonstrates how one can address the path explosion problem for complex software such as code generators. We believe that some of the techniques developed in `EISSEC` can be applied in the general context of the function extraction of other software.

Implementing program analysis and instrumentation passes at the compiler IR level is one of the approaches for source-to-source transformation and static analysis. This approach has been made popular by GCC and LLVM by introducing mechanisms to write plug-ins which can inspect or modify intermediate representations. To exploit such IR-level program analysis, compiler-specific assembly-to-IR translators are developed manually. In comparison to these compiler-specific tools, the approaches proposed in this dissertation are more general and applicable to a wide variety of compilers. `LISC`, in particular, is applicable to any compiler, and it does not need to be worry about modifications to the internals of compilers. Systematic, architecture- and compiler-neutral approaches to develop assembly-to-IR translators is thus yet another important contribution of this dissertation.

To demonstrate an applicability of the extracted semantic models, we developed an approach to test the correctness of the code generators. Testing compilers and ensuring that they are bug-free is one of the important problems in software engineering. By developing an approach to check the correctness of the code generators, this dissertation makes an important contribution towards solving this problem. Moreover, by developing a compiler-neutral approach and demonstrating its effectiveness in finding the modeling bugs in a real-world compiler (GCC), `ArCheck` demonstrates that the approach is practical and can be applied to test real-world compilers.

Broadly speaking, developing assembly-to-IR translators and testing code generators are two of the possible applications of the semantic models. This dissertation abstracts out the common problem of semantic model extraction across many applications, and addresses it by developing approaches for the automatic extraction. In other words, applicability of this dissertation goes beyond development of assembly-to-IR translators.

7.1 Future Work

One of the important contributions of this dissertation is to develop a framework for the extraction of semantic models using compilers. There exist many interesting applications of this framework. One such application is to use the extracted semantic models to build binary translation, instrumentation and analysis systems. Although, we demonstrated that the extracted models can be used to build assembly-to-IR translators quickly, building a binary analysis, translation and instrumentation systems on top of these translators will be a useful application of the extracted models. Many binary analyses and instrumentations are platform-independent, and can be applied to the binaries from many architectures. Unfortunately, a limited architecture support restricts their effectiveness. Our approach can address this problem and increase the effectiveness of existing systems.

Given that `LISC` is a black-box approach to learn the models followed by a code generator, we believe that it can be extended to learn the models from other systems such as assemblers, virtual machine, etc. Moreover, it seems that the idea of testing the extracted models can be extended to test these systems by defining appropriate correctness criteria. Checking the correctness of these systems can help in eliminating modeling bugs, and can also reduce the programmer efforts needed in bug detection, analysis and fixing.

We believe that the models extracted using our techniques have several applications beyond testing and using them to build other systems. For instance, the models can be used to check the conformance of a system with a standard or reference model. For instance, by extracting an assembly to machine code model embedded inside an assembler, we can check the conformance of the assembler to a standard specification for instruction encoding. More broadly, these models can also be checked using techniques from the model checking domain, and one can also approach the verification problem with this direction. Furthermore, the models of multiple systems and multiple implementations of same specification can also be compared to find issues in them using techniques from n-version testing.

References

[1] BEXTR intrinsic has memory operands switched around. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=57623.

[2] Decision tree learning. http://en.wikipedia.org/wiki/Decision_tree_learning.

[3] GCC, the GNU Compiler Collection. http://gcc.gnu.org/.

[4] i386.md strangeness in sse2_movsd. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=14941.

[5] Intel 64 and IA-32 Instruction Set Reference, A-Z. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[6] RTL representation of i386 shrdl instruction is incorrect? https://gcc.gnu.org/bugzilla/show_bug.cgi?id=61503.

[7] Spaceship operator miscompiled. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=53138.

[8] The LLVM Compiler Infrastructure. http://llvm.org/.

[9] x86_64-linux-gnu-gcc generate wrong asm instruction MOVABS for intel syntax. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=56114.

[10] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. Extracting and verifying cryptographic models from c protocol code by symbolic execution. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 331–340, 2011.

[11] Kapil Anand, Matthew Smithson, Aparna Kotha, Khaled Elwazeer, and Rajeev Barua. Decompilation to Compiler High IR in a binary rewriter. Technical report, Tech. rep., University of Maryland (November 2010), http://www.ece.umd.edu/barua/high-IR-technical-report10.pdf.

[12] ARM. ARM Architecture Reference Manual ARMv7A and ARMV7-R edition. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html, 2014.

[13] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. Finding Bugs in Dynamic Web Applications. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 261–272, 2008.

[14] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. CodeSurfer/x86 — A Platform for Analyzing x86 Executables. In *Proceedings of the 14th International Conference on Compiler Construction*, CC'05, pages 250–254, 2005.

[15] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, 2005.

[16] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. RWset: attacking path explosion in constraint-based test generation. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 351–366, 2008.

[17] Derek L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Cambridge, MA, USA, 2004.

[18] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, 2011.

[19] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008.

[20] Cristian Cadar and Dawson Engler. Execution generated test cases: How to make systems code crash itself. In *Proceedings of the 12th International Conference on Model Checking Software*, SPIN'05, pages 2–23, 2005.

[21] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, pages 322–335, 2006.

[22] Anton Chernoff and Ray Hookway. DIGITAL FX! 32 running 32-bit x86 applications on alpha NT. In *Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997*, pages 2–2, 1997.

[23] Vitaly Chipounov and George Candea. Dynamically Translating x86 to LLVM using QEMU. Technical report, 2010.

[24] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems*, 2009.

[25] Cristina Cifuentes, Brian Lewis, and David Ung. Walkabout - A Retargetable Dynamic Binary Translation Framework. In *In Proceedings of the 2002 Workshop on Binary Translation*, 2002.

[26] Cristina Cifuentes and Doug Simon. Procedure Abstraction Recovery from Binary Code. In *4th European Conference on Software Maintenance and Reengineering, CSMR 2000, Zurich, Switzerland, February 29 - March 3, 2000.*, pages 55–64, 2000.

[27] Cristina Cifuentes, Mike Van Emmerik, and Norman Ramsey. The design of a resourceable and retargetable binary translator. In *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, pages 280–291, 1999.

[28] Christian S. Collberg. Reverse Interpretation + Mutation Analysis = Automatic Retargeting. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 57–70, 1997.

[29] Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. Symbolic crosschecking of floating-point and SIMD code. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 315–328, 2011.

[30] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

[31] Atmel Corp. Atmel AVR 8-bit Instruction Set. www.atmel.com/images/Atmel-0856-AVR-Instruction-Set-Manual.pdf, 2014.

[32] Jack W. Davidson and Christopher W. Fraser. Automatic generation of peephole optimizations. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '84, 1984.

[33] Jack W. Davidson and Christopher W. Fraser. Code Selection Through Object Code Optimization. *ACM Trans. Program. Lang. Syst.*, 1984.

[34] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, 2008.

[35] Thomas Dullien and Sebastian Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. *Proceeding of CanSecWest*, 2009.

[36] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. Scalable Variable and Data Type Detection in a Binary Rewriter. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 51–60, 2013.

[37] Wen Fu, Rongcai Zhao, Jianmin Pang, and Jingbo Zhang. Recovering Variable-Argument Functions from Binary Executables. In *Computer and Information Science, 2008. ICIS 08. Seventh IEEE/ACIS International Conference on*, pages 545–550, May 2008.

[38] Vijay Ganesh and David L. Dill. A Decision Procedure for Bit-vectors and Arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 519–531, 2007.

[39] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, 2005.

[40] Patrice Godefroid, Michael Y Levin, and David A Molnar. Automated Whitebox Fuzz Testing. In *Network Distributed Security Symposium (NDSS)*, volume 8, pages 151–166, 2008.

[41] Patrice Godefroid and Ankur Taly. Automated Synthesis of Symbolic Instruction Encodings from I/O Samples. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 441–452, 2012.

[42] J.N. Hooker. Solving the incremental satisfiability problem. *The Journal of Logic Programming*, 15(1–2):177 – 186, 1993.

[43] William E. Howden and Suehee Pak. The Derivation of Functional Specifications from Source Code. In *Proceedings of the Third Asia-Pacific Software Engineering Conference*, APSEC '96, pages 166–, 1996.

[44] Wilson C. Hsieh, Dawson R. Engler, and Godmar Back. Reverse-Engineering Instruction Encodings. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, pages 133–145, 2001.

[45] Chun-Chen Hsu, Pangfeng Liu, Chien-Min Wang, Jan-Jan Wu, Ding-Yong Hong, Pen-Chung Yew, and Wei-Chung Hsu. LnQ: Building High Performance Dynamic Binary Translators with Existing Compiler Backends. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 226–234, 2011.

[46] Yuan-Shin Hwang, Tzong-Yen Lin, and Rong-Guey Chang. DisIRer: Converting a retargetable compiler into a multiplatform binary translator. *ACM Trans. Archit. Code Optim.*, 7(4):18:1–18:36, December 2010.

[47] Intel. Intel 64 and IA-32 Architectures Software Developer Manuals. http://www.intel.com.

[48] Johannes Kinder and Helmut Veith. Jakstab: A Static Analysis Platform for Binaries. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pages 423–427, 2008.

[49] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[50] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient State Merging in Symbolic Execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 193–204, 2012.

[51] Lillian Lee. Learning of Context-Free Languages: A Survey of the Literature. Technical Report TR-12-96, Harvard University, 1996.

[52] Xavier Leroy. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. POPL '06, 2006.

[53] Xavier Leroy. A Formally Verified Compiler Back-end. *J. Autom. Reason.*, December 2009.

[54] Christian Lindig. Random Testing of C Calling Conventions. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, AADEBUG'05, 2005.

[55] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, 2005.

[56] Rupak Majumdar and Ru-Gang Xu. Directed test generation using symbolic grammars. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 134–143, 2007.

[57] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing CPU Emulators. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, 2009.

[58] Peter M. Maurer. Generating Test Data with Enhanced Context-Free Grammars. *IEEE Softw.*, July 1990.

[59] William M. McKeeman. Differential Testing for Software. *DIGITAL TECHNICAL JOURNAL*, 1998.

[60] Microsoft. Microsoft common object file format. http://www.skyfree.org/linux/references/coff.pdf.

[61] Barton P. Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*, December 1990.

[62] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[63] Mehryar Mohri. Finite-state Transducers in Language and Speech Processing. *Comput. Linguist.*, 23(2):269–311, June 1997.

[64] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012.

[65] George C. Necula. Translation Validation for an Optimizing Compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, 2000.

[66] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, 2002.

[67] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, 2007.

[68] J. Oncina, P. García, and E. Vidal. Learning subsequential transducers for pattern recognition interpretation tasks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(5):448–458, May 1993.

[69] J. Oncina and Miguel Angel Varó. Using Domain Information During the Learning of a Subsequential Transducer. In *Proceedings of the 3rd International Colloquium on Grammatical Inference: Learning Syntax from Sentences*, ICG! '96, pages 301–312, 1996.

[70] J. Pichler. Specification extraction by symbolic execution. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 462–466, Oct 2013.

[71] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation Validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, pages 151–166, 1998.

[72] Corina S. Păsăreanu and Neha Rungta. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 179–180, 2010.

[73] J. R. Quinlan. Induction of Decision Trees. *Mach. Learn.*, 1(1):81–106, March 1986.

[74] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. 1993.

[75] Gerd Ritter. *Formal Sequential Equivalence Checking of Digital Systems by Symbolic Simulation*. PhD thesis, TU Darmstadt, March 2001.

[76] Lior Rokach and Oded Maimon. *Data Mining with Decision Trees: Theory and Applications*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2008.

[77] William C. Rounds. Mappings and grammars on trees. *Mathematical systems theory*, 4(3):257–287, 1970.

[78] Yasubumi Sakakibara. Learning context-free grammars from structural data in polynomial time. *Theoretical Computer Science*, 76(2–3):223 – 242, 1990.

[79] H. Samet. A Machine Description Facility for Compiler Testing. *Software Engineering, IEEE Transactions on*, 1977.

[80] R. Sekar, I. V. Ramakrishnan, and Andrei Voronkov. Handbook of automated reasoning. chapter Term Indexing, pages 1853–1964. 2001.

[81] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive Pattern Matching. In *Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, ICALP '92, pages 247–260, 1992.

[82] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference*

held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/FSE-13, pages 263–272, 2005.

[83] Flash Sheridan. Practical Testing of a C99 Compiler Using Output Comparison. *Softw. Pract. Exper.*, November 2007.

[84] Junaid Haroon Siddiqui and Sarfraz Khurshid. Scaling symbolic execution using ranged analysis. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 523–536, 2012.

[85] Bochs Team. Bochs IA-32 emulator. `http://bochs.sourceforge.net/`.

[86] James W. Thatcher. Generalized sequential machine maps. *Journal of Computer and System Sciences*, 4(4):339 – 367, 1970.

[87] Nikolai Tillmann and Jonathan De Halleux. Pex: White Box Test Generation for .NET. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, 2008.

[88] Tool Interface Standard (TIS). Executable and Linking Format (ELF) Specification. `http://www.uclibc.org/docs/elf.pdf`, 1995.

[89] Alok Tongaonkar, R. Sekar, and Sreenaath Vasudevan. Fast Packet Classification Using Condition Factorization. In *Proceedings of the 7th International Conference on Applied Cryptography and Network Security*, ACNS '09, pages 417–436, 2009.

[90] Markus Triska. The finite domain constraint solver of SWI-Prolog. In *FLOPS*, volume 7294 of *LNCS*, pages 307–316, 2012.

[91] Markus Triska. *Correctness Considerations in CLP(FD) Systems*. PhD thesis, Vienna University of Technology, 2014.

[92] Peter Vanbroekhoven, Gerda Janssens, Maurice Bruynooghe, and Francky Catthoor. A Practical Dynamic Single Assignment Transformation. *ACM Trans. Des. Autom. Electron. Syst.*, 12(4), September 2007.

[93] VMWare. Vmware. `http://www.vmware.com/`.

[94] Shaohui Wang, Srinivasan Dwarakanathan, Oleg Sokolsky, and Insup Lee. High-level model extraction via symbolic execution. Technical Report MS-CIS-12-04, Department of Computer and Information Science, University of Pennsylvania, 2012.

[95] J. Whittemore, Joonyoung Kim, and K. Sakallah. SATIRE: A new incremental satisfiability engine. In *Design Automation Conference, 2001. Proceedings*, pages 542–545, 2001.

[96] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[97] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, 2011.

[98] Mingwei Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 337–352, Berkeley, CA, USA, 2013.

[99] Chen Zhao, Yunzhi Xue, Qiuming Tao, Liang Guo, and Zhaohui Wang. Automated test program generation for an industrial optimizing compiler. In *Automation of Software Test, 2009. AST '09. ICSE Workshop on*, May 2009.

[100] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formal Verification of SSA-based Optimizations for LLVM. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, 2013.