Automatic Generation of Assembly to IR Translators Using Compilers

Niranjan Hasabnis and R. Sekar

Stony Brook University, NY, USA

***Abstract*—Translating low-level machine instructions into higher-level intermediate representation (IR) is one of the central steps in many binary translation, analysis and instrumentation systems. Most of these systems manually build the machine instruction to IR mapping table needed for such a translation. As a result, these systems often suffer from two problems: (a) a great deal of manual effort is required to support new architectures, and (b) even for existing architectures, lack of support for recent instruction set extensions, e.g., Valgrind's lack of support for AVX, FMA4 and SSE4.1 for x86 processors. To overcome these difficulties, we propose a novel approach based on learning the assembly-to-IR mapping automatically. Modern compilers such as GCC and LLVM embed knowledge about these mappings in their code generators. By leveraging this knowledge, our approach can greatly reduce the implementation effort required for lifting binary code to IR. Moreover, such an approach is architecture-neutral, being able to support numerous architectures for which GCC (or other compilers) already have a backend. While coverage can be a challenge in learning-based approaches, note that in this problem domain, there is virtually an endless supply of training data that can be obtained by translating vast quantities of open-source code using compilers such as GCC and LLVM. We present experimental results that demonstrate the promise of our approach. Already, our implementation can support multiple architectures (x86, ARM and AVR), handle binaries of significant size (`openssl` and `binutils`), and be applied to multiple compilers (GCC and LLVM).**

I. Introduction

Binary translation has been a very popular technique in providing cross-ISA compatibility for decades with QEMU [1] being one of the popular open-source binary translators. Binary instrumentation, on the other hand, has been a popular technique in software monitoring, debugging and policy enforcement. Pin [2], Valgrind [3], and DynamoRio [4] are among the most frequently used frameworks to support such instrumentation.

Translating assembly (or binary code) to an intermediate representation (IR) is the first step in such frameworks. For instance, Valgrind translates assembly instructions to its IR called VEX. The essential idea behind using IR is to make the instrumentation code for debugging and policy enforcement architecture-independent, and thus applicable for binaries of any architecture. Architecture-independence is thus one of the desired properties of such frameworks. But all these frameworks suffer from one significant limitation: they rely on manually developed translators from assembly to their IR. Unfortunately, such a manual approach is laborious and error-prone for complex architectures such as x86, where the manuals describing the instruction set semantics run to thousands of pages (for instance, Intel's 1400-page instruction set reference (version 052) [5].) As a result, these frameworks often support only a limited number of architectures, and even for a single architecture, support only a subset of the instructions. For instance, Valgrind[1], still lacks support for several classes of x86 instructions, including AVX, FMA4, and SSE4.1, even after being in development for 10 years.

In this paper, we propose a novel automatic approach that significantly reduces the manual efforts needed to build assembly to IR translators. Our approach relies on the observation that modern compilers such as GCC [6] and LLVM [7] already contain detailed knowledge about instruction semantics for many different architectures. Specifically, their code generators can translate their IR to assembly code for many different architectures. Moreover, these compilers have been tested extensively, thereby minimizing the chances for errors in modeling instruction set semantics. Thus, an interesting open question is: *Can we leverage the knowledge encoded in modern retargetable compilers for lifting assembly back to IR?* We answer this question in the affirmative, and develop a (largely) black-box approach for extracting this knowledge into rules for translating assembly into IR.

Our approach offers three main benefits. First, it deals with most of the complexities of modern instruction sets automatically. Second, it helps in supporting many more architectures than those supported by existing manual approaches. Third, it also helps in supporting recent instruction set extensions for existing architectures.

Our evaluation shows the potential of our approach. We have demonstrated its ability to support three different architectures (x86, ARM and AVR), while taking only a few man-hours for each additional architecture. It is already able to handle moderate-sized programs such as `openssl` and `binutils`, and can work with multiple compilers (GCC and LLVM).

II. Approach and System Design

Compiler researchers have long worked to develop architecture-independent code generators [8]. These code generators start with an intermediate representation (IR), and emit assembly code. GCC's code generator is driven by

This work was supported in part by grants from NSF (CNS-0831298, CNS-1319137) and AFOSR (FA9550-09-1-0539).

[1]We used Valgrind version 3.7.0 for our testing.

an instruction set specification called a *machine description* (MD). Typically, there is a single MD rule corresponding to each assembly instruction, and it specifies the semantically-equivalent IR code. There may be additional constraints that limit when each rule is applicable, e.g., some of the operands should be registers.

Code generation now becomes a pattern-driven process: given a piece of IR code that needs to be translated to assembly, the code generator compares snippets of this IR against the IR snippets in each MD rule, and replaces the matching IR by the assembly instruction specified in the rule.

It may appear that we could simply use these MD rules in reverse to translate assembly into IR. Unfortunately, this is not possible because the MD rules are not complete specifications. While they typically include basic information such as the assembly instruction name and the number of operands, many concrete details are hard-coded directly into the source code of the code generator, and are not present in the MD rules. Missing details include (a) the addressing modes and the operand expressions (at the assembly as well as IR levels), and (b) the conditions under which a rule is applicable, and on so on. As a result, the MD rules cannot directly be used to translate assembly to IR without considerable additional work to understand the source code of the architecture-specific components of the code generator, and encoding this knowledge directly into MD rules[2].

The incompleteness of MD specifications motivates the approach we propose — a blackbox approach that discovers the mapping by simply observing the (IR) inputs and the corresponding (assembly code) outputs of a code generator, and learning the mappings from these observations. Moreover, availability of vast amounts of training data, obtained by compiling large bases of open-source software, makes such learning-based approach feasible. Another important benefit of our blackbox approach is that it isn't tied to a particular compiler, e.g., it can be applied to GCC as well as LLVM without requiring any changes.

A. System Design

We first use GCC to compile many source code packages, and collect the IR to assembly translations performed by GCC. Figure 1 illustrates a mapping observed when compiling a program containing the statement $\mathtt{x} = \mathtt{x} + 2$. All such captured IR to assembly mappings are then fed to the next step in the process.

The simplest learning approach is to memorize the exact translations observed in the training data. We call this approach *Exact recall*. However, such approach requires impractical volumes of training data, while using unacceptable amount of memory to store the mappings. For instance, the raw mapping in Figure 1 can only handle the case of adding 2, and we would need 2^{32} such rules to handle all possible 32-bit values of that can appear in place of 2. To avoid such explosion,

[2]These observations about machine descriptions apply not only to GCC but also to LLVM, whose *target descriptions* are partly in the form of specifications, with many concrete details incorporated directly into architecture-specific components of LLVM source-code.

IR instruction	Assembly instruction
[(set (reg : SI ax) (plus (reg : SI ax) (const_int 2))) (clobber (reg : FLAGS))]	add $2, %eax

Fig. 1: **x86 `add` assembly instruction and its IR**

we developed an approach for learning parameterized rules as described below.

We begin by identifying parameters in assembly instructions. Currently, we identify only numeric parameters, such as the number 2 in the above example. One reason for this choice is that numeric parameters are the main source of explosion in terms of the number of raw mappings produced; other parameters such as registers don't lead to an explosion since their domain is usually small.

Once parameters are identified, they are replaced with a symbolic parameter name in the assembly instruction. Any occurrences of the replaced value in the IR are also replaced with the corresponding parameter name. At this point, a parameterized rule can be learned. For instance, for the example shown in the Figure 1, its parameterized rule is shown in the row 1 of Figure 2. This parameterized rule captures the semantics of adding any numeric constant to the `eax` register.

Blindly replacing constant values in IR with parameter names can lead to errors in some cases. For instance, for an assembly instruction `foo` $2, $2, the corresponding IR instruction will mostly have immediate 2 a couple of times. In that case, just looking at this mapping rule, it would be hard to tell which $2 is first parameter and which one is second parameter. Note that this ambiguity arises because of limited training data (specifically, just one instance of `foo` $2, $2 is not enough to understand precise parameter mapping.) To resolve this confusion, we rely on a simple idea of processing multiple instances of the same assembly instruction with different parameter values. For instance, for above example, we will look for an instruction `foo` $2, $3. That way, parameter positions become clear. Having multiple instances eliminates the errors that might occur when we have only one instance.

Note that our parameterized rule learning predominantly assumes that constants appear unchanged between IR and assembly. Nonetheless, we have explored extensions that support simple transformations on parameters, such as addition, subtraction, multiplication, or division with a constant. For instance, occurrences of x in assembly may be replaced by $x+1$, or perhaps $x \times 2$ in IR. However, in our experiments so far, we have observed such transformations only for assembly instructions that subtract a constant (e.g., x86 `subl`). For such instructions, IR semantics is to add $-x$.

In the final step, we flag any inconsistencies identified in the parameterized rules. Two types of inconsistencies are possible. The first involves multiple distinct parameterized assembly instructions produce the same IR. This is usually not a cause for concern, as there can be several assembly instructions that have the same effect, such as `xor` %`eax`, %`eax` and `mov` $0, %`eax`. Currently, we manually examine these cases to

No	IR	Assembly
1	(set (reg : SI ax)(plus (reg : SI ax)($**P**))) (clobber (reg : FLAGS))	add $**P**, %eax
2	(parallel [(set (reg : SI r4) (minus : SI (reg : SI r3) (reg : SI r7))) (clobber (reg : CC cc))])	subs r4, r3, r7
3	(set (reg : DF st0) (neg : DF (reg : DF st0)))	fchs

Fig. 2: **Examples of parameterized rules generated by our approach**

verify that there is no ambiguity; development of automated techniques to reduce the number of manual inspections is a topic of ongoing research. The second inconsistency involves a mapping of the same assembly instruction to distinct IRs. Unless all such IRs are semantically equivalent, this inconsistency likely indicates an error in the mapping. We are currently developing techniques for recognizing common cases of equivalence among IRs, and avoid the need for manual examination to rule out inconsistencies. In our experiments so far, we have had to deal with a few instances of first inconsistency, while those of second inconsistency were none.

B. Illustration and Discussion

Figure 2 lists a few illustrative examples of the rules learned by our approach. We make following comments based on these examples.

- *Our approach requires only the barest minimum details about the architecture.* For instance, the row 2 in the figure shows an ARM assembly instruction. Someone who isn't well-versed in ARM assembly may not be able to identify the source and the destination operands of the assembly instruction. It is interesting to note that this information is obvious in the IR component of the rule learned by our system, even though our implementation has no knowledge about these details. Indeed, our implementation does not make any distinction between x86, ARM, or assembly formats.
- *Handling implicit operands.* Many architectures, especially x86, have instructions where some of the operands are implicit. To translate such assembly instructions into IR, one needs to know all the implicit operands, and expose them accurately in IR. This is yet another complex task which must be handled by manual approach. But our approach does not need to handle it at as GCC's MD rules clearly specify such implicit operands. For instance, the `fchs` instruction has no explicit operands, but at the IR-level, GCC's code generator has captured the operands affected by the instruction.
- *Capturing all the effects of instruction execution.* A potential concern in using machine descriptions is that at the IR level, some effects may be specified incompletely, or may be missing altogether.
CPU flags are the usual example of incomplete specification. From rows 1 and 2, we only have the information that flags are clobbered (i.e., modified) by the instruction, without specifying exactly how. This incompleteness will be a problem in some applications (a typical example being malware analysis), but not for binary instrumentation: note that any missing information can be obtained by directly querying the CPU at runtime using an appropriate instruction, e.g., the x86 instruction `lahf` can be used to move the value of CPU flags to the `ah` register. Binary instrumentations can use such instructions to obtain the values of any operands whose values are incompletely specified.
While incomplete specifications are easy to deal with in the context of binary instrumentation, missing effects can be a serious problem that compromises the correctness of instrumentation. Fortunately, MDs generally will not leave out effects, or else they would cause the compiler to emit incorrect code. For instance, suppose that there is an assembly instruction I that modifies a register X but this effect is missing in the corresponding IR component of the MD. This would render incorrect many low-level analyses results computed by the compiler, e.g., the set of registers modified in a basic block. This can, in turn, lead to incorrect register allocation, and/or clobbering of results previously stored in register X and accessed by an instruction following I.

III. Implementation

Our current prototype implementation works on Linux, and we have used it to automatically generate assembly-to-IR translators for x86, ARM and AVR[3] architectures using GCC-4.6's code generator.

The implementation for collecting IR to assembly translations is architecture-neutral and is done using a GCC plugin which took around 70 lines of C code. The plugin is used in a standard manner in which GCC plugins are used. To collect IR to assembly translations for `foo.c`, one would use the command `gcc -dP -fplugin=rule_collection.so -fplugin-arg-out-file=log.S foo.c`, where `-dP` is a standard GCC option to tell GCC to dump the IR corresponding to each assembly instruction as a comment. Thus the translation collection phase easily integrates with `configure` and `make` based package compilation process used commonly on Linux.

The implementation of parameterized rule-learning is independent of GCC, and it took 900 lines of C++ code. All this code is architecture-independent, but we do require minimal architecture-specific code (around 70 lines per architecture) to identify comments and actual assembly instructions from the collected dump files. Architecture-independent code identifies parameters from the assembly instructions (we do not encode architecture-specific knowledge (such as syntax) for identifying parameters — code simply looks for numeric values)

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

and maps them to IR parameters. The mapping functions implemented are exact equality and transformation functions such addition, subtraction, multiplication, and division with a constant (between 1 and 64[4]). In the last step, exact duplicate parameterized rules are eliminated.

IV. Evaluation

In this section, we provide experimental results that demonstrate the benefits outlined earlier. To evaluate effectiveness of our approach, we used it to build assembly-to-IR translators for x86, ARM, and AVR architectures. We used `openssl` and `binutils` packages to obtain compilation logs.

a) Sizes of the compilation logs and the generated assembly-to-IR translators: Figure 3 compares the sizes of the compilation logs and the assembly-to-IR translators extracted using these logs. We used 3 different metrics for comparison: the number of unique concrete rules (obtained after removing exact duplicates) from the log files, the number of parameterized rules derived by our approach from the concrete rules, and the number of mnemonics covered by the derived parameterized rules. Interesting things to note about the figure is that our approach obtains around 3X reduction in the number of concrete rules by parameterizing them and eliminating duplicate parameterized rules. Although the number of mnemonics covered by the combination of `openssl` and `binutils` is not considerably more than that covered by these packages individually, increase in the number of concrete rules indicate that a plenty of new operand combinations were covered. These combinations also translated in the increase of the number of parameterized rules when logs obtained from new packages are added to existing list.

b) Correctness: To ensure correctness, we undertook two steps. First, our system reports an error if there can be any assembly instruction that can be mapped into two distinct IRs. Second, we use a "loop back" test: we use the assembly-to-IR mapping learned by our system to lift a binary B to IR. We then run the compiler (GCC in our case) with the IR as input, and verify that it produces the exact same assembly code as in the binary B. This loop-back test was performed on all of the binaries used in the tests, and it worked without generating any discrepancies or errors.

c) Completeness: A systematic evaluation of completeness requires significant knowledge about a 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 obtain compilation logs, and test them using assembly instructions obtained from 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 rules learned from P_{train}. Specifically, for our experiments, P_{train} consists of `openssl` and `binutils` packages, while P_{test} consists of selected programs from the `coreutils` package. As a comparison point, we implemented an *exact recall* approach, which would look for exact match of assembly instructions from P_{test} binaries in the logs obtained from P_{train}. If a match is found, the corresponding IR instruction is emitted as the translation of the input assembly instruction.

The result is shown in Figure 4a. In the figure, results are presented for a combination of different training data and assembly-to-IR translation approaches (In the figure, ER stands for *exact recall*). For this testing, we used `coreutils` binaries that come along with a typical Ubuntu-14.04 desktop install[5]. Labels on x-axis contain `coreutils` program names and the number of assembly instructions in them. After training our system with GCC's logs of `openssl` and `binutils`, we could lift an average of around 94% instructions in all of `coreutils` binaries. On the other hand, after training with the log of only `openssl`, our system could translate an average of around 90.7% of the instructions. It clearly indicates that `binutils` package provided some operand combinations which were not found in the `openssl` log, and these combinations helped in translating 3.3% of the additional instructions. This observation also falls in line with a slight improvement in the results of *exact recall*. Nonetheless, even with the combined log, *exact recall* could translate an average of around 42% instructions only. This underlines the potential of replacing constants with parameters, which helped our system cover around 50% additional instructions.

The reason we could not lift 100% of the instructions can be attributed to certain instructions we could not lift. One such case is that of instructions (e.g., `nop`) generated by the assembler for padding bytes[6]. These instructions are not generated by GCC, and hence are not learned by our system. Another source of incompleteness is that the compiler itself may not use all of the instructions from a target architecture. To address these sources of incompleteness, it may be necessary to manually specify rules for missing instructions. Still, the amount of manual effort required is greatly reduced as compared to approaches that rely entirely on manual specifications: the vast majority of instructions are already handled by our learning approach, so only a small fraction may have to be manually specified. Lastly, we also speculate that register combinations might also be contributing some percentages (though not significant) to the instructions that could not be lifted. To address this issue, we might consider about treating registers as parameters in the future.

d) Support for multiple architectures: We then repeated the completeness tests described in the preceding paragraph for the ARM architecture. After training our system with the

[4]Limits 1–64 are configurable. We found this range to be sufficient for all the tested instructions.

[5]To the best of our knowledge, Ubuntu uses GCC to compile programs. We speculate that the exact version of GCC used is 4.7.

[6]Since these instructions are padding bytes, we might think that these need not be lifted. But we do not want to encode such architecture-specific knowledge in our system.

Arch	Parameter	Packages used for compilation		
		`openssl`	`binutils`	`openssl+binutils`
x86	# of unique concrete rules (in K)	21.8	40.3	55.3
	# of parameterized rules (in K)	6.7	14.2	19.4
	# of unique mnemonics	100	132	135
ARM	# of unique concrete rules (in K)	32.4	38.7	45.1
	# of parameterized rules (in K)	7.3	11.5	15.2
	# of unique mnemonics	87	97	104
AVR	# of unique concrete rules (in K)	0.3	0.43	0.56
	# of parameterized rules (in K)	0.17	0.25	0.31
	# of unique mnemonics	23	27	35

Fig. 3: **Details of training data used for learning purpose**

compilation logs of `openssl` and `binutils` (obtained using GCC's cross-compiler for ARM), we could lift an average of around 91% of the instructions of the ARM's `coreutils` binaries (same ones used in x86 test). On the other hand, *exact recall* could reach the maximum of around 33%.

We then selected AVR architecture, which is not supported by any of the existing binary instrumentation frameworks. We trained our system by compiling `openssl` and `binutils` packages using GCC's cross-compiler for AVR, and then tested it on same set of `coreutils` binaries (for AVR) used in x86 test. Our system was able to translate 92% of the instructions, while *exact recall* was able to translate the maximum of around 48%. The total time taken including the time to port architecture-specific part of our implementation was hardly 3 man hours.

e) Compiler Independence: To find out how many of the instructions produced by compilers other than GCC can be lifted by our system, we used LLVM compiler (`clang3.3`) to produce `coreutils` binaries for x86 by compiling `coreutils-2.23` package. We lifted the compiled binaries to IR using the rules learned from GCC's compilation logs of `openssl` and `binutils`. Results of this evaluation are summarized in Figure 4b. Our system was able to lift an average of around 91% of the instructions. We found that 9% instructions that we could not lift were using mnemonics and operand combinations not covered in the training data. This finding is also reflected in the result of *exact recall*, which is around 37% for LLVM produced binaries (while it is around 42% for GCC produced binaries.)

f) Lifting advanced x86 instructions: Recall our comment that Valgrind does not yet support some of the advanced x86 instructions. So we decided to evaluate our ability in handling them. Specifically, we included AVX, FMA4, SSE4.1, AES, and RAND in our evaluation. We trained our system using the source code of scientific and image editing packages such as `gimp` (in addition to training it on `openssl`), where these advanced instructions were most likely to be used. After training our system, we measured that our mapping table covers 97% of these advanced instructions at least once. (This means that 97% of the opcodes could be lifted, but there is no guarantee that all possible operand combinations of these instructions can be lifted.)

V. Related work

All of the previous approaches to translate low-level instruction into high-level IR are based on manually building the components needed for such translation. Approaches such as [2], [4], [9], [3], [1], [10], [11], [12], [13], [14] require a hand-written target instruction specification to drive the translator. Notably, Valgrind [3], one of the popular dynamic binary instrumentation systems, relies on manually building assembly to VEX (its IR) translators. QEMU [1], one of the popular binary translators, requires manually-developed backend to support new architectures. SecondWrite [11] requires an XML specification of the architecture instructions, whereas UQBT [12] has designed its own format for such specifications. All these approaches suffer from the drawbacks mentioned in the Introduction.

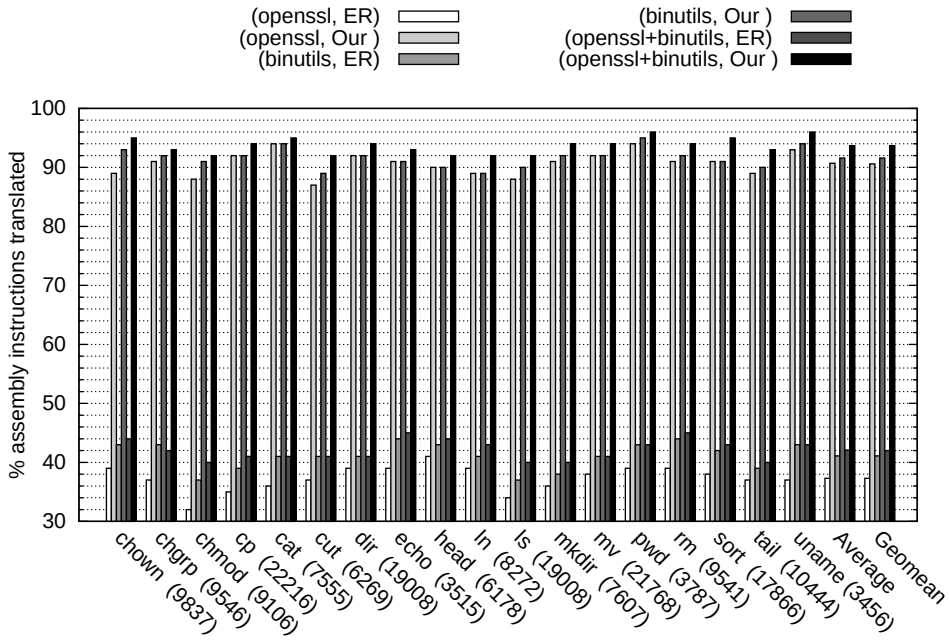
Approaches [15], [16] have developed assembly-to-IR translators by relying on QEMU's support for multiple architectures. Specifically, they have written a backend for QEMU to translate QEMU's IR to LLVM's IR. BAP [17], on the other hand, directly uses Valgrind's assembly to IR translator. Unfortunately, all these systems suffer from their dependence on QEMU/Valgrind, and they inherit all the limitations of QEMU and Valgrind discussed above.

Dagger [18] is one of the recent decompilation frameworks that uses LLVM compiler for assembly-to-IR translation. In particular, it relies on LLVM code generator's architecture modelling but adds new libraries to LLVM for decompilation. Unfortunately, Dagger's biggest limitation is that it treats LLVM as a white-box. This limitation makes Dagger specific to LLVM, and is not easily portable at all. Moreover, with changes to LLVM, Dagger needs to be updated as well. Our approach, on the other hand, treats a compiler as a black-box, and is thus very portable.

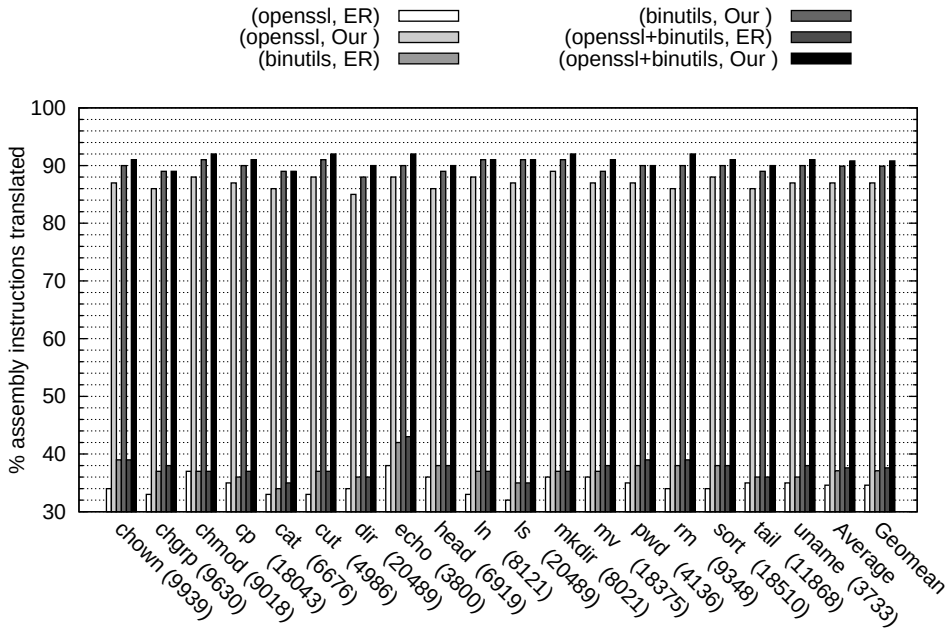
VI. Conclusions and Future Work

In this paper, we outlined an automated black-box approach for assembly-to-IR translation which can potentially reduce most of the manual efforts in building assembly-to-IR translators. The key contribution of our approach is that it reduces most of the manual efforts in modelling the semantics of instruction sets. Instead, it leverages knowledge about instruction sets that is already incorporated into modern multi-target compilers.

We believe that our experimental results demonstrate the



(a) `coreutils` **binaries (GCC compiled) found in Ubuntu-14.04 distribution**



(b) **LLVM compiled** `coreutils` **binaries**

Fig. 4: **Details of completeness results for** `coreutils` **binaries on x86**

potential of our approach. Our ongoing and future work is aimed at a comprehensive evaluation of completeness using

a large training suite, and determining whether full coverage can be obtained. We also plan to address related questions such as the relative completeness of different compilers (such as ICC [19]) in modeling instruction sets. Additionally, if completeness results demand consideration of registers as parameters, then we would plan to extend our approach to handle them as well. Finally, and most importantly, we plan to develop binary analysis and instrumentation applications based on this approach.

References

[1] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX ATC*, 2005.

[2] C.-K. Luk et al., "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.

[3] N. Nethercote et al., "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *PLDI*, 2007.

[4] V. Kiriansky et al., "Secure Execution via Program Shepherding," in *USENIX Security*, 2002.

[5] "Intel 64 and IA-32 Instruction Set Reference, A-Z," http://www.intel.com/.

[6] "The GNU Compiler Collection," http://gcc.gnu.org.

[7] "The LLVM Compiler Infrastructure Project," http://llvm.org.

[8] J. W. Davidson and C. W. Fraser, "Code selection through object code optimization," *ACM Trans. Program. Lang. Syst.*, 1984.

[9] C. C. et al., "Walkabout - a retargetable dynamic binary translation framework," in *Workshop on Binary Translation*, 2002.

[10] T. Dullien et al., "REIL: A platform-independent intermediate representation of disassembled code for static code analysis," 2009. [Online]. Available: http://zynamics.com/downloads/csw09.pdf

[11] K. Anand et al., "Decompilation to compiler high IR in a binary rewriter," Univ of Maryland, Tech. Rep., 2010.

[12] C. Cifuentes et al., "The design of a resourceable and retargetable binary translator," in *WCRE*. IEEE, 1999.

[13] G. Balakrishnan et al., "CodeSurfer/x86 a platform for analyzing x86 executables," in *CC*, 2005.

[14] J. Kinder et al., "Jakstab: A static analysis platform for binaries," in *CAV*, 2008.

[15] V. Chipounov et al., "Dynamically Translating x86 to LLVM using QEMU," Tech. Rep., 2010.

[16] C.-C. Hsu et al., "LnQ: Building high performance dynamic binary translators with existing compiler backends," in *ICPP*, 2011.

[17] D. Brumley et al., "BAP: a binary analysis platform," in *CAV*, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2032305.2032342

[18] "Dagger," http://dagger.repzret.org.

[19] "Intel Compilers," https://software.intel.com/en-us/intel-compilers.