Lifting Assembly to Intermediate Representation: A Novel Approach Leveraging Compilers *

Niranjan Hasabnis
Intel †
niranjan.hasabnis@intel.com

R. Sekar
Stony Brook University
sekar@cs.stonybrook.edu

Abstract

Translating low-level machine instructions into higher-level intermediate language (IL) is one of the central steps in many binary analysis and instrumentation systems. Existing systems build such translators manually. As a result, it takes a great deal of effort to support new architectures. Even for widely deployed architectures, full instruction sets may not be modeled, e.g., mature systems such as Valgrind still lack support for AVX, FMA4 and SSE4.1 for x86 processors. To overcome these difficulties, we propose a novel approach that leverages knowledge about instruction set semantics that is already embedded into modern compilers such as GCC. In particular, we present a learning-based approach for automating the translation of assembly instructions to a compiler's architecture-neutral IL. We present an experimental evaluation that demonstrates the ability of our approach to easily support many architectures (x86, ARM and AVR), including their advanced instruction sets. Our implementation is available as open-source software.

1. Introduction

Binary analysis and instrumentation form the basis of many popular systems for program debugging and monitoring (e.g., Valgrind [37], DynamoRio [13] and Pin [36]), processor virtualization (e.g., QEMU [12]), and malware analysis (e.g., BitBlaze [49] and BAP [14]). Numerous techniques and solutions in software security, including taint-tracking [38, 42, 44], control-flow integrity [6, 57], sandboxing untrusted code [23, 32, 54], malware analysis [22, 52, 55], and automated exploit and signature generation [10, 19], rely on binary analysis/instrumentation.

A central challenge faced by all these systems is the accurate modeling of instruction semantics. Errors in the model will invalidate binary analysis results. Worse, they may cause instrumented programs to crash or fail in other ways. The size and complexity of modern instruction sets compounds instruction semantics modeling: the popular x86 architecture supports over 1000 instructions that are documented in a 1500-page manual. Despite its roots in RISC, ARM v7A includes several instruction sets such as Thumb, ThumbEE and Jazelle that cause the instruction set size to balloon to about 1200, and the manual to about 1000 pages.

Existing binary analysis and instrumentation systems rely on manually developed models of instructions. Given the size of modern instruction sets, manual modeling poses a daunting challenge even for a single architecture. Supporting multiple architectures such as PowerPC, x86, ARM, SPARC, and MIPS using this approach can be prohibitive. It is no wonder that even mature systems such as Valgrind[1] support only a limited number of platforms, and moreover, omit instruction subsets such as AVX, FMA4, and SSE4.1 on x86, and ThumbEE and Jazelle on ARM.

In contrast with previous binary analysis and instrumentation works, all of which relied on manual instruction semantics modeling, we present *a novel automated approach* in this paper. Our approach leverages the knowledge of instruction sets that is already encoded in existing compilers such as GCC [40] and LLVM [24]. Specifically, their code generators can translate their (architecture-neutral) intermediate language to assembly code for many different architectures. We present techniques to (a) automatically "extract" this semantics, and (b) to test the accuracy of extracted semantics. Our approach offers several important benefits:

- *Automated instruction semantics modeling.* Our approach can greatly reduce the manual efforts needed in modeling instruction set semantics.
- *Architecture-neutrality.* This feature has long been recognized as important, starting with early works such as EEL [35] and UQBT [17]. However, the focus of these works was on the analysis and instrumentation steps that followed a *manually implemented* step to lift assembly to an intermediate language (IL). Our work addresses the significant challenge left open by these (and other subsequent) works, namely, automating the lifting step.
- *Leveraging well-tested compiler code.* Compilers are among the most extensively tested pieces of software —

* This work was supported in part by grants from NSF (CNS-0831298 and 1319137), AFOSR (FA9550-09-1-0539) and ONR (N00014-15-1-2378).

† This work was completed as a PhD student at Stony Brook University.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS '16, April 02–06, 2016, Atlanta, GA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4091-5/16/04. . . $15.00.
DOI: http://dx.doi.org/10.1145/2872362.2872380

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

both during development and in the field. As a result, errors in instruction semantics are likely to have been found and fixed. By extracting this "debugged" instruction semantics, our approach reduces the likelihood of errors that are all too common in manually lifted assembly [1, 3–5].

1.1 Approach Overview and Contributions

Problem Formulation. Modern compilers such as GCC rely on an architecture-specific *machine description (MD)* to drive their code generator. However, as discussed in Section 2, these MDs are interpreted using large amounts of architecture-specific C-code. Indeed, many MD rules directly contain C-code! As a result, it is not feasible to extract instruction semantics by directly processing the MDs. We therefore formulate the model extraction problem as a machine-learning problem: given examples of IL to assembly translation produced by a compiler, learn a mapping from assembly to IL.

Algorithm for Learning Assembly to IL Mappings. While there exist a vast array of machine-learning algorithms, most of them are focused on the problem of classification or regression, i.e., given an input, they output a (small) integer or a real-value. The output in our learning problem are more complex, consisting of tree-structured[2] IL snippets. Secondly, machine learning algorithms are designed with the assumption that errors can be tolerated, and indeed, are a necessary component of any solution that generalizes well. In contrast, translation errors are unacceptable in our context. To our knowledge, the only previous works that satisfy these constraints are on finite-state transducers (FSTs). However, these algorithms (e.g., OSTIA [39]) operate on strings, and do not address two major challenges that arise in our problem context: *parameterized translation* and *tree-structured data*. We describe these challenges in Section 3, and develop efficient techniques to overcome them.

Efficient Lifting of Whole Binaries to IL. Our learning algorithm maps (short) *sequences* of assembly instructions into IL. Given a long instruction sequence, there can be multiple ways to lift it to an IL sequence. As a result, a straight-forward lifting algorithm can have an exponential complexity. In Section 4, we show how to exploit dynamic programming to realize a linear-time lifting algorithm.

Correctness of Extracted Mappings. While compilers use the MD rules to translate IL to assembly, we are using them in the reverse direction. This raises a question as to the correctness of the approach. We provide a conceptual justification of correctness in Section 5, followed by a more formal treatment and experimental evaluation in Section 6.4.

Experimental Evaluation. Our evaluation in Section 6 establishes the following:

- *Ability to support complex instruction sets.* Our system can extract the semantics for about 1100 user-level instructions on x86, including advanced instruction sets such as SSE and AVX; and about the same number for ARM, including its Thumb, VFP, and SIMD instructions.
- *Completeness.* We show that our approach can successfully lift about 99.5% of Ubuntu/x86 and 99.8% of Debian/ARM binaries. Most of the missed instructions are various forms of NOPs and other instructions that are never used by GCC. Because of the simplicity of these instructions, it took us only a few hours to manually extend the model to incorporate these instructions.
- *Architecture-neutrality.* It took *just 9 person-hours* to support ARM, and 3 hours to support AVR, a popular microcontroller used in platforms such as Arduino.
- *Performance.* We trained the system using 1.4GB of code generator logs. This training produced a transducer with 4830 states in 10 minutes. We then used this transducer to lift all the binaries from the entire Ubuntu/x86 and Debian/ARM distributions, which total about 100M instructions. This testing took approximately 8 hours.

In a companion paper [41], we describe the application of the semantics derived by our approach. A short summary of this application is discussed in Section 6.5. Comparison with related work is presented in Section 7, followed by concluding remarks in Section 8.

Our system is available as open-source software [27].

2. Background and Problem Formulation

Compiler researchers have long worked to develop architecture-independent code generators [20]. These code generators translate code in an architecture-neutral intermediate language (IL) into assembly code. Their operation is driven by *machine descriptions* (MDs) that describe the instruction set of a target architecture. The core component of an MD is a set of rules, each mapping a snippet of IL into an assembly instruction (or in some cases, a sequence of assembly instructions). This general structure is applicable to modern compilers such as GCC and LLVM, but due to GCC's mature support for numerous architectures, our implementation is based on GCC. An example rule from GCC's x86 MD is shown below:

```
[(set (match_operand:0 "register_operand" "=a")
  (div (match_operand:1 "register_operand" "0" )
    (match_operand:2 "nonimmediate_operand" "qm")))
 (clobber (reg: FLAGS_REG))]
⟶ "div %2"
```

This rule consists of an RTL[3] pattern corresponding to the divide instruction. This RTL snippet states that the result of dividing operand 1 by operand 2 is stored in operand 0. At the assembly level, the first two operands are same and implicit, a fact captured by (the cryptic) match constraints "**0**" and "**=a**" in the RTL. The applicability of the pattern is constrained by several other (architecture-specific) con-

[2] We are referring to a parse tree of the IL snippet.

[3] RTL is the name of GCC's IL.

straints as well, including "***register_operand***" and "**qm**." Often, compilers do not need to reason about the exact value of flags after each and every arithmetic or logical instruction. Hence this rule indicates that flags are modified without specifying exactly how[4].

It may seem that MD rules can be used in reverse to translate assembly to IL, but unfortunately, as illustrated by the above example, the rules are not self-contained. Several details, such as the meanings of the constraints and the printing of assembly-level operands (i.e., how `%2` is substituted in the above example) are hard-coded into architecture-specific C-functions. Even the output assembly may not be specified for some rules: instead, the right-hand side of the rule may consist of C-code that, when compiled and executed, will return a string representing the output assembly instruction! Reversing these rules will hence require all such C-code to be inverted. Computing inverses of arbitrary C-functions is an intractable problem in general, so we need alternative methods. One possibility is to rely on human experts to study these C-functions and hand-generate the inverses. Unfortunately, we find that the amount of C-code that needs handling is far too much to make this approach attractive. For instance, the x86 MD in GCC consists of 1500 rules (40K lines), *but a far larger 90K lines of C-code.* Moreover, such manual efforts can introduce new bugs, thus negating the primary benefit of MDs, i.e., they have been tested extensively.

To overcome the above problems, we develop a *black-box* approach that *observes* the IL-to-assembly translations actually produced by a compiler, and *learns* a mapping from these observations. Such an approach eliminates the need to understand the semantics of MDs or the associated architecture-specific code. This, in turn, makes the technique easier to apply to a broad range of compilers, including those that rely on even more programmatic MDs than GCC. While coverage is a concern with learning-based approaches, note that in the context of this problem, large amounts of training data can be readily obtained by compiling (a virtually endless supply of) open-source software, and recording the IL-to-assembly mappings.

We formulate the assembly-to-IL lifting problem as one of *learning a parameterized translation* on *trees* representing assembly and IL snippets. We say "parameterized" because the translations involve parameters, which correspond to operands in the assembly and IL. In addition, we operate on parse-trees (rather than strings), as their structure provides additional clues for the learning algorithm, and moreover, the notion of parameters is more easily understood in the context of trees. The following abstract example will help illustrate some of the key requirements of the learning algorithm we will describe in the next section.

[4] Static analysis and instrumentation techniques typically require sound instruction semantics, but can typically cope with incompleteness, e.g., when flags (or in some cases, other operands) are specified as clobbered. This is why the incompleteness of `div` semantics does not pose a problem. For a more detailed discussion, see Section 6.4.1.

Example 1. We use a first-order term notation to represent trees. For instance, $a(e, b(c, d))$ represents a tree with the root symbol a and two children. The first child is the (leaf) symbol e, while the second child is a tree with the root symbol b and two subtrees c and d that are leaves. The following table shows a list of pairs given to the learning algorithm, and a (possible) rule learned from them:

Asm	IL	Rule learned
$a(4,2)$	$s(2, b(2,2,8))$	$a(x,y) \longrightarrow s(y, b(2, y, \mathbf{f}(x)))$
$a(1,1)$	$s(1, b(2,1,2))$	
$a(7,3)$	$s(3, b(2,3,14))$	

Here, $\mathbf{f}$ is a function that multiplies its argument by 2, while x and y are universally quantified variables that represent parameters. This example has been chosen to illustrate several common features we observe in the assembly to IL translation. For instance, parameters may be reordered because IL tends to list its destination operand first, while assembly may list it last. Parameter duplication is also common since assembly typically uses two-operand format for most operations, while IL uses 3-operand format. It is also relatively common for IL to add additional operators such as b — for instance, a constant 10 in assembly may be represented as (`cint` 10) in IL. Finally, constants may be transformed using simple functions such as $\mathbf{f}$, e.g., they may be offset or scaled by a small number. Thus, the learning algorithm needs to be able to recognize such transformations.

Some of the constants in the first two columns correspond to parameters, while others, such as the first child of b, don't. The learning algorithm needs to distinguish between them by observing several examples.

Operating on trees means that we need to first parse assembly code as well as RTL. The latter task is easy because RTL uses an S-expression syntax that is easy to parse, and moreover, it is a one-time effort. Parsing assembly, on the other hand, is an architecture-specific task. To minimize its development effort, we build a "rough" parser, one that recognizes the nesting of operators and operands to build a tree, but is unaware of many other syntactic distinctions, e.g., the differences between addressing modes. As a result, the entire parser code for x86 is less than 100 lines.

3. Learning Algorithm

We begin by enumerating the key requirements for the algorithm for learning translations:

1. *Error-free:* Translation errors cannot be tolerated, because they will undermine the soundness of tools relying on the mapping derived by the approach. At a minimum, our algorithm should avoid errors on the training data; ideally, it will have no errors (except, possibly, missing translations) on test data.

2. *Fully automatic:* This algorithm should not require manual supervision or any form of intervention.

3. *Parameterized translation:* Both assembly and RTL consist of operands that often range over large domains (e.g., 32-bit integers), and may go through transformations such as the addition or multiplication by small constants, big endian to little endian conversions, etc. Our algorithm should be able to learn such common transformations.

4. *Scalable to handle large data sets:* As mentioned before, a large amount of labeled training data is available in the context of this problem, and so we seek an algorithm that is efficient enough to exploit it.

To the best of our knowledge, previous techniques do not satisfy several of these requirements. For instance, machine-learning techniques target applications that can tolerate errors. Often, a conscious trade-off is made that modestly increases the error rate in order to achieve better generalization from training samples. This is particularly true in the context of modern machine translation literature, where the focus is on the very difficult problem of natural language translation.

In the simpler setting of string translations, techniques that focus on minimizing errors have been developed. Specifically, Oncina et al [39] developed an algorithm called OSTIA for learning so-called *onward subsequential transducers*[5] from examples. Being *onward* means that the transducer emits as much of the output as possible at each transducer state, instead of waiting until all of the input is examined. Doing so exposes commonalities among states, enabling more generalization (achieved by merging similar states).

A key property of OSTIA is its guarantee of zero errors on the training data. However, OSTIA does not support the concept of parameters such as those shown in Example 1, or transformations on them. As noted in Example 1, even the identification of parameters is nontrivial. Moreover, OSTIA operates on strings, while our translation algorithm operates on trees. While trees can be flattened into a string representation, this is undesirable for many reasons. First, important structural information, which can provide clues for the learning algorithm, is lost. Second, the kind of parameter reordering shown in Example 1 leads to a situation where almost no output can be emitted until all of the input is read[6]. This leads to degenerate transducers that emit all their output in the very last state, with the output being distinct for every distinct input. In effect, this leads to a degenerate case of learning, namely, *exact recall,* which memorizes input/output pairs observed during training, and can recall the output when a previously seen input is presented again. We will use this exact recall algorithm as a baseline for our comparison, while developing a new algorithm that satisfies all the requirements listed above. Our system, called LISC (Learning Instruction Semantics from Code generator), uses three main steps as discussed below: (1) *training data collection,* (2) *parameterization,* and (3) *transducer construction.*

[5] Transducers are very similar to automata: while automata accept an input language, transducers perform translations, and as such, their transitions not only consume inputs but also emit outputs.

[6] This is because the last parameter of the input produces the first parameter of the output, so the only output that can be produced before consuming all input is the symbol "*s*".

3.1 Training data collection

We first use GCC to compile many source code packages and collect concrete pairs of assembly and IL-snippets produced by the code generator. Note that this collection occurs at the very last step of code generation: at this point, GCC has completed register allocation, so the RTL refers to hardware registers. This factor simplifies the learning step. A few example pairs are shown below, where the RTL has been abbreviated a bit to improve readability.

Asm	RTL
`sub $8, %eax`	`(set (reg eax) (+ (reg eax) (cint -8)))`
`sub %ebx, %eax`	`(set (reg eax) (- (reg eax) (reg ebx)))`
`sub %eax, 8(%esp)`	`(set (mem (+ (reg esp) (cint 8))) (- (mem (+ (reg esp) (cint 8)))(reg eax)))`

Both the assembly and IL are represented as first-order terms after parsing. For instance, the term pair corresponding to the first row of the table is

$$\langle sub(8, eax), \quad set(reg(eax), +(reg(eax),\ cint(-8)))\rangle$$

Note that a function $\mathbf{g}(n) = -n$ has been applied to the parameter 8, similar to the function $\mathbf{f}(n) = 2n$ used in Example 1. Other features of Example 1, such as parameter reordering and duplication, are also present.

Typically, there is just a single assembly instruction per pair, but there are instances where there is a sequence of them. Our algorithm makes no distinction between these cases, treating the sequence operator as yet another assembly-level operator.

3.2 Parameterization

The goal of parameterization is to (a) identify all parameters from a concrete pair $\langle A, I\rangle$, and (b) determine any functions that are being applied to these parameters. Since we cannot definitively identify a parameter by just looking at one $\langle A, I\rangle$ pair, *we conservatively flag every constant (i.e., a leaf in the term) in assembly as a* ***potential parameter***[7].

We next identify, for each pair $\langle A, I\rangle$, the ways to derive each of the leaf terms in I from the potential parameters identified in A. Our design here is based on our observation that there are only a small number of transformations that are commonly found in assembly to IL mappings:

- eq*:* Identity is the most common transformation.
- $+, -$*:* Addition or subtraction of a small integer $k > 0$.
- $*, /$*:* Multiplication or division by a small integer $k > 0$.

[7] Non-leaf subterms represent expressions containing an operator at their root. Since we expect operators to have different semantics (if not syntax) in assembly and IL, we do not consider them as parameters. In contrast, constants such as "8" or "eax" can be copied over from assembly to RTL or vice-versa, and hence we consider them as potential parameters.

- $2^k, \log_2$*:* These occur when a multiplication operation by a power of two is translated into a shift operation.
- *Numeric conversions* such as big-endian to little-endian.
- *Subword extraction,* e.g., extracting a specific byte of a 32-bit value. We have observed this often in the context of SIMD instructions, where multiple 8-bit values from RTL are packed into a single 32-bit assembly operand.

Our parameterization algorithm starts with a small set $\mathbf{f}_1, \ldots, \mathbf{f}_r$ of transformation functions such as those listed above. (We explicitly omit more complex functions, e.g., $kx + l$ or kx^2.) Then, for each leaf l in I and each potential parameter x in A, it checks if $\mathbf{f}_i(x) = l$. If there is no such $\mathbf{f}_i$, l is left as is. Otherwise, l is replaced by a combination of all such transformations, plus l itself. If there is any potential parameter x that is unused in any of these transformations, then it is eliminated from consideration as a parameter. We illustrate this procedure using a few examples below.

In/out pair		After parameterization	
$a(5,b)$	$t(s)$	$a(5,b)$	$t(s)$
$a(2,c)$	$t(6)$	$a(x,c)$	$t(\{6, x*3, x+4\})$
$a(4,c)$	$t(8)$	$a(x,c)$	$t(\{8, x*2, x+4\})$
$d(2,4)$	$u(5,3)$	$d(x,y)$	$u(\{5, x+3, y+1\}, \{3, x+1, y-1\})$
$d(5,6)$	$u(7,6)$	$d(x,y)$	$u(\{7, x+2, y+1\}, \{6, x+1, y\})$

Figure 1. Examples of parameterization

In the first row, the arguments of a are not considered parameters, as they bear no relationship to the leaf s. In the second row, the leaf 2 is identified as a potential parameter. It may either be multiplied by 3, or added to 4 in order to yield the leaf 6 in IL. It is also posssible that 6 is a constant that won't be affected by changes to the parameter value 2. While parameterizing, we capture all these three possibilities by using a set notation for the output parameter value, with the semantics that *all of the expressions in the set yield the correct value* for the output parameter.

The fourth and fifth rows involve two parameters instead of one, but as before, we simply replace each leaf in I by a set of expressions. If the learning algorithm, at some point, concludes that the same parameterized rule is implied by two distinct examples, then the two parameterized pairs are merged, provided the following conditions hold:

- they match at every non-leaf position, and
- the intersection of corresponding leaves is nonempty.

The intersection corresponding to rows 4 and 5 will yield the IL-term $u(y+1, x+1)$. (Since the sets consist of just a single expression, we have omitted the braces around them.)

3.3 Transducer construction

Our transducer operates on terms, and is hence a tree-transducer. Such a transducer is similar to a trie or an acyclic DFA. Like a DFA, a transducer operates in time that is linear in the size of its input, and hence it provides fast translation

```
procedure MkDucer(S, R):
1.  j = select(R_A);  Record j in S
2.  for each b in R_A[j] do
3.    R_b = {⟨R_A, R_I⟩ ∈ R | root(R_A[j]) = b}
4.    T_b = mcp(R_b)
5.    Create new state S_b
6.    Create a transition from S to S_b annotated with T_b
7.    R'_b = {residue(R, T_b) | R ∈ R_b}
8.    if R'_b = ∅
9.      then mark S_b final
10.   else MkDucer(S_b, R'_b)
```

Figure 2. Transducer construction algorithm

from assembly into IL. At the same time, a tree transducer differs from a DFA in two important ways:

- a transducer's transitions specify not only the input symbols that must match, but also output symbols that are emitted on each transition.
- unlike a string transducer, whose input consumption (and output emission) occurs in a strictly left-to-right manner, a *tree transducer* traverses its input (and emits output) in any order, as long as it visits parent nodes before children.

Our algorithm $MkDucer$ for constructing a transducer is shown in Fig. 2. We begin with an intuitive illustration of this algorithm on the example of Fig. 1, which results in the transducer shown in Fig. 3. The states of the transducer are annotated with a set of $\langle A, I\rangle$ pairs and a position to be examined next in A. The start state S_0 of the transducer is associated with the entire set $\mathbf{R}$ of $\langle A, I\rangle$ pairs from Fig. 1. At S_0, there is only one option for the next position to examine, i.e., the root of A, so we don't show it explicitly.

Inputs in Fig. 1 are of the form $a(x, y)$ or $d(x, y)$, so we create a transitions from S_0 to capture these two cases. The output is of the form $t(z)$ in the first case. In the second case, we obtain an output of $u(y + 1, x + 1)$ using the merge step described at the end of Section 3.2. We annotate the edges from S_0 with these $\langle in, out\rangle$ pairs.

State S_1 corresponds to the first three rows of Fig. 1. At S_1, we have examined $a(x, y)$, the *maximal common prefix (mcp)* of the three (assembly) terms, and emitted $t(z)$, the mcp of the three IL terms. (A prefix of a term is obtained by omitting some of its subterms.) The subterms omitted in

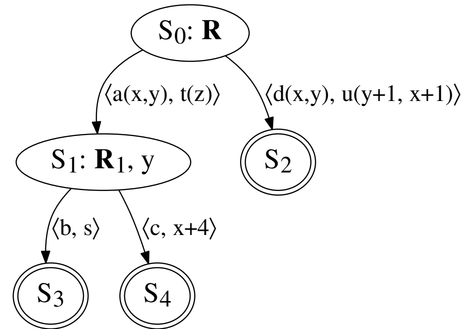

Figure 3. Result of $MkDucer$ (Fig. 2) on Fig. 1

the prefix are called *residues* — they represent the yet-to-be-examined parts of the assembly instruction, and the yet-to-be determined parts of the output. Note that a residue consist of lists of subterms $[t_1, \ldots, t_n]$ that we call as a *fringe.* The state S_1 is annotated with this set $\mathbf{R}_1$ of residues. From the first three rows of Fig. 1, it is easy to see

$$\mathbf{R}_1 = \{\langle [5, b], s\rangle, \langle [2, c], 6\rangle, \langle [4, c], 8\rangle\}$$

We have omitted brackets when a fringe is singleton.

$MkDucer$ uses a helper function $select$ to determine the next position to visit from S_1. Suppose that y represents this position. This choice is noted in S_1. Since there are two possible values for y, namely, b and c, two transitions are created out of S_1. On each of these transitions, the output is fully determined, so S_3 and S_4 are marked as final states, and $MkDucer$ terminates.

Now we proceed to a technical description of $MkDucer$. An invocation of $MkDucer(S, \mathbf{R})$ builds a subtree rooted at a state S. A prefix of the input and output terms, as specified by the path from the transducer root to S, has been examined. $\mathbf{R}$ represents the residues of $\langle A, I\rangle$ pairs that are compatible with this prefix. Thus, *each element* of $\mathbf{R}$ is a pair of fringes $\langle R_A, R_I\rangle$, where R_A and R_I are each sequences of subterms of a parameterized pair $\langle A, I\rangle$ that have not been captured in the common prefix.

$MkDucer$ uses a helper $select$ to pick the next input position to examine. Since we are interested in selecting from input positions, $select$ is a function of $\mathbf{R}_A$, the input components of $\mathbf{R}$. (For simplifying the illustration, this selection was identified using a variable name in Fig. 3, but the algorithm uses an integer index into the fringe array.)

Next, we partition $\mathbf{R}$ on the basis of the symbols occurring at this selected position. The partition corresponding to a symbol b is denoted $\mathbf{R}_b$. The members of $\mathbf{R}_b$ agree on this symbol b, but they may share more. We use the function mcp to identify their *maximal common prefix* T_b. We then create a new state S_b and a transition from S to S_b annotated with T_b.[8] This annotation means that T_b will be examined on this transition, so we compute what is left to be examined in $\mathbf{R}_b$ using the function *residue*. If the residue is empty, then we are done, and we mark S_b as final. Otherwise we make a recursive call to complete the transducer subtree rooted at S_b.

To complete this algorithm, we define mcp and *residue*:

DEFINITION 1 (***Maximal common prefix).*** *$mcp(t_1, ..., t_n)$ is the largest term t such that every t_i is an instance of t. It is defined inductively as follows:*

- *If t_1 or t_2 is a variable x, or $root(t_1) \neq root(t_2)$ then $mcp(t_1, t_2) = x$*
- *If t_1 and t_2 represent sets of expressions (from Section 3.2) then if $t_1 \cap t_2 = \phi$ then $mcp(t_1, t_2) = x$, otherwise it is $t_1 \cap t_2$*
- *Otherwise $t_1 = c(t_{11}, \ldots, t_{1r})$ and $t_2 = c(t_{21}, \ldots, t_{2r})$, and $mcp(t_1, t_2) = c(mcp(t_{11}, t_{21}), \ldots, mcp(t_{1r}, t_{2r}))$*

For two fringes $T_1 = [t_{11}, \ldots, t_{1r}]$ and $T_2 = [t_{21}, \ldots, t_{2r}]$, $mcp(T_1, T_2) = [mcp(t_{11}, t_{21}), \ldots, mcp(t_{1r}, t_{2r})]$.

DEFINITION 2 (***Residue).*** *If t_1 is an instance of t_2 then $residue(t_1, t_2) = [t_{11}, \ldots, t_{1r}]$ such that t_1 is obtained by substituting the ith variable in t_2 by t_{1i}, for $1 \leq i \leq r$. Otherwise the residue is undefined.*
For two fringes $T_1 = [t_{11}, \ldots, t_{1r}]$ and $T_2 = [t_{21}, \ldots, t_{2r}]$, $residue(T_1, T_2) = [residue(t_{11}, t_{21}), ..., residue(t_{1r}, t_{2r})]$

We illustrate mcp and $residue$ in the following tables:

t_1	t_2	$mcp(t_1, t_2)$
$t(s)$	$u(5, 3)$	x
$\{5, x+3\}$	$\{5, x+2\}$	5
$\{5, y+1\}$	$\{7, y+1\}$	$y+1$
$a(5, a(b, c))$	$a(4, a(b, b))$	$a(x, a(b, y))$

t_1	t_2	$residue(t_1, t_2)$
$t(6)$	$t(x)$	6
$a(5, a(b, c))$	$a(x, a(b, y))$	$[5, c]$
$a(b, \{x+2, y*3\})$	$a(x', y')$	$[b, \{x+2, y*3\}]$

Defining $select$***.*** For the basic version of $MkDucer$, it is easy to define $select$. It chooses only non-parameter positions. Among non-parameter positions, it chooses a position that minimizes the number of immediate children of the current state. If only parameter positions remain, this basic version of the algorithm fails. For the related problem of building matching automata for trees, such a choice has been shown to be optimal [45–47]. Extensions to this basic algorithm are described later in this section.

Building DAG Transducers. Since assembly language grammars typically have a fixed number of operands with very limited depth, cycles are not useful (or even meaningful) in our transducers. However, DAG structure can still be useful, as it can produce considerable space savings.

While conversion to DAG is usually thought of as a bottom-up post-processing optimization, such an approach wastes significant resources: Tree automata can be exponentially larger than DAG automata, so post-processing techniques can perform exponentially worse than direct DAG construction approaches.

Given our choice of passing only residues ("unseen inputs and unemitted outputs") into $MkDucer$, there is a simple and elegant approach for direct DAG construction: before creating a new state S_b corresponding to $\mathbf{R}_b$, simply check if there already exists another state S' with the same residue, and if so, create a transition to the existing state, and avoid the recursive call. Such an approach works correctly because $MkDucer$'s behavior is fully determined by the argument $\mathbf{R}$, and hence two invocations that pass in the same $\mathbf{R}$ will result in identical subautomata.

[8] By the properties of mcp, T_b includes the maximal part of the output that can be determined at this point. Eagerly emitting output in this manner isn't critical for this version of the algorithm but *is* important for an optimization we discuss later.

Improving $select$***.*** Typically, branching on parameter positions can be avoided if the assembly language parser is capable of distinguishing between basic operand types, such as immediates and registers. However, in the interest of simplicity, our parser does not make this distinction, so we needed to generalize $select$ further. Secondly, even if a better parser were available, this generalization makes $MkDucer$ more powerful, capable of handling complexities that would trip up the basic version. Thirdly, this generalization can produce space savings by avoiding multi-way branches with numerous branches, instead using two-way branches.

When $select$ is unable to find a non-parameter position to branch on, or when the branching factor is too large, it builds if-then-else branches. If the position contains parameters of different types such as integers and strings, then the branch will be on the basis of type. Otherwise, $select$ compares with a constant that divides the current set of residues into two equal halves. Values less than this constant will follow the then-branch, while the rest will follow the else-branch.

Extending $select$ in this manner requires the mcp and $residue$ operations to be extended so that we can deal with inequalities. Due to space constraints, we omit the details here, but an interested reader can find them in Reference [50], which studies the problem of building matching automata for network packets.

4. Efficient Lifting of Whole Binaries to IL

We next turn our attention to the problem of using the assembly-to-IL mapping learned in the previous section to lift whole binaries to IL. In order to do this, binaries need to be first disassembled. While this is a nontrivial problem itself, recent works (e.g., [56, 57]) have developed robust solutions, at least for our experimental platform. Thus we focus on the next step, namely, lifting (long) sequences of assembly instructions to IL.

Lifting would be straight-forward if the mapping derived in the last section operated on a single assembly instruction at a time. However, there are many instances where the translation operates on multiple instructions at the same time. This arises typically because compilers require some primitives that cannot be realized using a single instruction. For instance, manipulating stack canaries requires multiple assembly instructions that should not be separated for security reasons. Handling such groups is necessary not only because some instructions may occur only in groups[9], but also because it is advantageous to reconstruct the higher-level primitives when lifting up back to IL.

The most obvious approach for lifting in the presence of instruction groups is to use a greedy approach that maximizes the size of each group that can be lifted in one run of the transducer. The greedy strategy is motivated by the observation that larger sequences can reconstruct more of the high-level information. Unfortunately, it can be easily shown that a greedy strategy is not optimal: a greedy choice early on in the assembly sequence may preclude the use of many larger groups later on. Indeed, a greedy strategy is not even guaranteed to find *any* solution at all, unless it uses backtracking. Moreover, such a backtracking approach can have an exponential worst case complexity.

On further reflection, it becomes clear that the exponential blowup occurs due to repeated efforts to solve overlapping subproblems. By recognizing this structure, we develop an efficient algorithm that uses dynamic programming. The subproblems we consider correspond to prefixes of the given sequence $A_1 \cdots A_n$ of assembly instructions. We let C_j denote the minimum cost of lifting the prefix $A_1 \cdots A_j$. To compute this cost, we need to assign a cost GC to each group of assembly instructions that can be lifted by a single pass of the transducer. To express our preference for larger groups, we set $GC(B_1 B_2 \cdots B_r) = -r$ if the transducer can translate $B_1 B_2 \cdots B_r$, otherwise the cost is set to ∞.

The minimum cost C_j is given by the following equation, where k denotes the size of largest group that can be translated by the transducer.

$$C_j = \underset{1 \le i \le k}{Min} \, [C_{j-i} + GC(A_{j-i+1} A_{j-i+2} \cdots A_j)]$$

To see why this is correct, note that a minimum cost partitioning (into groups) of $A_1 \cdots A_j$ must have a last group. The size of this group cannot be larger than k, the largest group for which the transducer provides a translation. Moreover, preceding this last group is a partitioning of $A_1 \cdots A_{j-i}$. Clearly, the cost of this group should be C_{j-i} or else we can replace this partitioning with a lower-cost way and hence have a contradiction. This establishes the correctness of the above equation.

To analyze the complexity of computing C_j, note that it can be computed starting with $j = 1$ and going to $j = n$. Thus, we need to perform the "min" operation above n times. Each run of this min operation takes $O(k)$ time. Note that k should be property of the machine description used in the compiler, and should be small. In our experiments on x86, $k \le 4$. When k is treated a constant, computing C_j takes $O(n)$ time. Thus we have a linear-time algorithm for lifting whole binaries to IL.

As with other dynamic programming algorithms, it is not enough to compute the minimum cost, but we also need to identify the corresponding partitions. But this step is routine, so we don't describe it further.

5. Soundness

Code generators use machine descriptions to perform IL to assembly translations, so a natural question is whether it is sound to use them in reverse. We begin by describing how MDs are developed and used, and how this use supports their use in reverse. A more formal and rigorous treatment is provided in Section 6.4.

[9] In fact, we find that many instructions that operate on the gs register on x86 occur in such multi-instruction bundles.

Machine descriptions are developed by enumerating instructions in the target architecture, and specifying the semantics of those instructions in IL. Thus, the developer view indeed corresponds to an assembly to IL mapping. Secondly, note that the IL-to-assembly mapping performed by a code generator must be sound, or else it will generate incorrect code. Therefore, an assembly instruction must perform all of the actions that are included in IL. This suggests that perhaps the assembly instruction could do more than what was asked for by the IL, e.g., change an additional register. If so, assembly-to-IL translation does not capture these extra actions. However, consider the fact that the IL optimizer performs several optimizations such as removal of redundant computations and reordering of code snippets. These would be unsound if we allowed an IL snippet to be replaced by an assembly instruction that modified CPU state in ways beyond what was stated in IL. Modern compilers such as GCC make use of many such optimizations even on the lowest levels of their IL. Undocumented effects of assembly instructions will invalidate these optimizations, thereby leading to the generation of incorrect code. Since code generators undergo extensive testing (by virtue of compiling vast amounts of code), it is reasonable to expect that such bugs in MDs would long have been found and fixed.

Finally, and most importantly, we have previously formulated the notion of correctness of code generators [26]. Based on this formulation, we have developed an automated testing approach for checking the equivalence of IL semantics and the corresponding assembly instruction. We summarize this formulation, and our experimental results in applying this formulation, in Section 6.4.

6. Evaluation

Our evaluation addresses completeness (Section 6.1), architecture-neutrality (Section 6.2), performance (Section 6.3) and correctness (Section 6.4). Additionally, we evaluate the effectiveness of the extracted model by developing an application using it (Section 6.5). Our implementation platform is Linux, and the architectures we target are x86, ARM and AVR. AVR processors are widely used in embedded systems: they underpin the popular Arduino platform, as well as numerous automotive applications.

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.

6.1 Completeness

For completeness experiments, we first build a transducer using the set of concrete pairs observed while compiling a set of programs (P_{train}). We then use this transducer to translate assembly instructions from a set of test binaries (P_{test}). We used `objdump` for disassembly, as it is quite robust on Linux.

P_{train}	% insns lifted		LISC (%)		Missing Mnemonics (absolute)
	Exact Recall	LISC	Missing Mnemonics	Missing Operands	
`openssl-1.0.1f` `+binutils-2.22`	63.72	98.46	1.05	0.49	464
`+ffmpeg-2.3.3`	68.21	98.74	1.03	0.23	377
`+glibc-2.21`	68.74	98.80	1.01	0.19	346
`+ffmpeg-2.3.3`[10]	69.07	98.89	0.88	0.23	303
`+gstreamer-1.4.5`	71.07	99.10	0.79	0.11	221
`+qt-5.4.1`	72.45	99.21	0.69	0.09	161
`+linuxkern-3.19`	73.97	99.49	0.44	0.07	49
`+Manual`	74.04	100.00	0.00	0.00	0

Figure 4. Completeness result for x86

P_{test} consisted of all the unique instructions in all binaries (including kernel modules) on a standard Linux desktop distribution. While the total number of instructions in these binaries approaches a trillion, after eliminating exact duplicates, we arrived at about 40M instructions in P_{test}. Note than neither P_{test} nor P_{train} include instructions executable only in kernel mode, as GCC never generates them.

We provide two types of numbers on completeness: the first, expressed as a percentage, concerns the fraction of the approximately 40M unique instructions in P_{test} that could not be translated by LISC. To interpret these percentages, we compare them with that obtained by the baseline technique of *exact recall* (ER), which is able to translate an instruction in $A \in P_{test}$ only if $A \in P_{train}$. Note that ER is a good baseline because (a) the previously known transducer algorithm OSTIA reduces to ER in our setting, and (b) ER provides the same correctness guarantee as our system, i.e., the system will correctly lift any $A \in P_{train}$.

Our second completeness metric is an absolute number: it counts the fraction of (distinct) instruction mnemonics that are not lifted by LISC. Note that both x86 and ARM support about 1200 distinct mnemonics.

The training data set, which was always a subset of P_{test}, was varied as follows. We started with P_{train} consisting of two binaries `openssl` and `binutils`. We then identified the binary that had the lowest completion rate with this training data, and then added that binary to P_{train}. This step was repeated until maximum possible completeness was obtained. This is not 100% because some instructions are never generated by the compiler used in our experiments. We now discuss these completeness results for each architecture.

6.1.1 Completeness Results for x86

Coverage on P_{test}. The results for x86 are shown in Figure 4. Rows in the figure list the packages that were added to P_{train} 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 with commonly used GCC flags (`-O2`, `-msse1`, `-msse4.2`, `-mavx`, `-mi387`, `-mmmx`, etc.) to

[10] `ffmpeg` was recompiled with `SSE`, `i387`, `AVX` enabled one-by-one.

P_{train}	% insns lifted		LISC (%)		Missing Mnemonics (absolute)
	Exact Recall	LISC	Missing Mnemo-nics	Missing Operan-ds	
Openssl + binutils	34.58	88.21	6.86	4.94	814
+libfftw	44.60	92.77	6.01	2.22	553
+swig	51.43	95.67	3.72	0.61	422
+gcc	62.54	96.26	1.73	2.01	366
+libc	65.41	97.87	1.37	0.76	314
+gs	67.69	98.92	1.02	0.04	276
+slapd	69.54	98.95	0.85	0.2	246
+busybox	71.45	99.61	0.3	0.09	196
+libpoppler	71.90	99.66	0.3	0.04	126
+lib7z	72.10	99.78	0.21	0.01	76
+manual	72.23	100.00	0.00	0.00	0

Figure 5. Completeness result for ARM

capture a more complete set of instructions in the compiled packages. P_{test} consisted of all x86 binaries (including kernel modules) found on Ubuntu-14.04 desktop installation. To the best of our knowledge, Ubuntu uses GCC to produce these binaries. These binaries (9237 of them) were disassembled using `objdump-v2.24`.

After training LISC with a combination of `openssl` and `binutils`, which contain about 0.35M unique instructions, we could lift 98.46% of the 38M unique instructions in all of x86 binaries on Ubuntu-14.04. By adding a few more packages to the training set, coverage was increased to about 99.5%. At this point, the training data included about 3.1M unique instructions. Obtaining 100% coverage requires a small amount of manual effort, since the missing instructions are not used by GCC. This point is further elaborated below.

Coverage of Mnemonics. While counting instructions, we have counted a single mnemonic with different modes (byte vs word) separately. With this way of counting, there are 1187 total mnemonics on x86. (Recall that this number excludes instructions that are only available in the kernel mode.) Of these 1187, 49 mnemonics (4%) appear only in hand-written assembly, and are not generated by GCC. These include instructions such as `nop` and `enter` (inserted by the assembler), some rarely used arithmetic instructions (e.g., `aaa`, and `aad` operations on binary-coded decimals) and instructions to set/clear flag bits (e.g., `cld`), and low-level instructions such as `cpuid`, `invpcid`, and `rdtsc`.

Advanced instruction set extensions such as `sse`, `avx` and `fma` are covered in the semantics extracted by LISC. This contrasts with the fact that many of these extensions are not supported by mature (and popular) tools such as Valgrind.

We manually modeled these missing instructions. Since many of the instructions were simple, this wasn't a hard task, and took us only a few hours of effort.

6.1.2 Completeness Results for ARM

Coverage on P_{test}. For ARM, P_{test} contained all instructions obtained by compiling all binaries (7.3K in total) found on Debian-7.8.0 desktop installation, yielding a total of about 55M unique instructions. We used `gcc-4.7.3` cross-compiler for 32-bit ARM cortex-v7A [8], and used `arm-linux-gnueabi-objdump-v2.24` as the disassembler. The results are summarized in Figure 5. About 99.8% coverage could be obtained on P_{test} after training with about a dozen binary packages.

Component (language)	Arch-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	75	60	89
Utility code (`scripts`)	500	50	16	15

Figure 6. Breakdown of LISC implementation effort

Coverage of Mnemonics. By our count, there are about 1200 distinct mnemonics supported by ARM v7A, of which 76 (about 6%) were missing in the learned semantic model because they were unused by `gcc-4.7.3`. These can be classified as: a few miscellaneous instructions (such as `dbg` and `dmb`), some low-level instructions (such as `isb`, `mcr2`, `mrs`, and `wfi`), and a few other advanced instructions (such as `vtbl`, `vtbx`, and `vstm`).

The total time taken to implement code changes for supporting ARM, including the time for testing and debugging, was around 9.5 person-hours.

6.1.3 Completeness Results for AVR

We then selected the AVR [9] processor. We trained LISC by using the same base packages for training as the earlier experiments, and used `avr-gcc-v4.8.2` cross-compiler to compile them. We then tested the extracted model on a few coreutils binaries (`ls`, `cp`, `cat`, `echo` and `head`) for AVR.[11] We used `avr-objdump-v2.23.1` disassembler to disassemble these binaries. Unfortunately, we could not get complete `coreutils` package to cross-compile. The extracted model covered 72 of the 76 AVR mnemonics. Our system was able to translate all of the assembly instructions from the input binaries.

Manual modeling was required for just 4 mnemonics (`break`, `nop`, `wdr`, and `sleep`).

6.2 Effort for supporting multiple architectures

Figure 6 shows the breakdown of the number of lines of code that is architecture-specific. Log collection adds a small amount of code to GCC to record $\langle$assembly, IL$\rangle$ pairs for learning. (Recall that testing data comes directly from binaries.) This code, as well as the code for learning the transducer, is architecture neutral.

Manual effort needed to support a new architecture is very small. It takes less than 100 lines of OCaml Lex and

[11] These binaries were obtained by using `avr-gcc-v4.8.2` cross-compiler to compile source code of these utilities. We did not attempt to run these binaries, as that is not needed for our experiments.

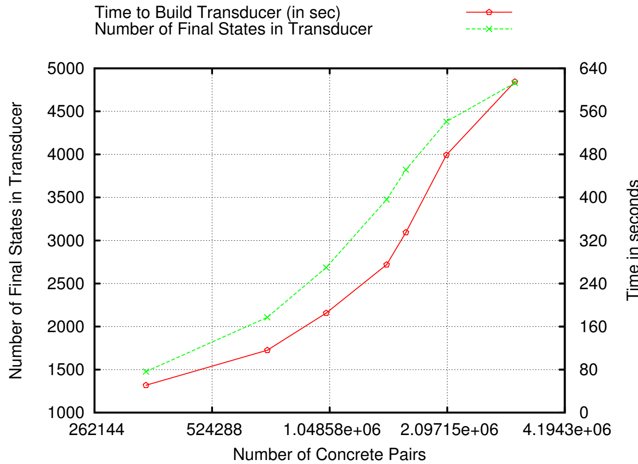

Figure 7. x86 Transducer Details

Yacc code to support each architecture. Note that this code is particularly easy to write since we are not writing full parsers for assembly, but a "rough" parser that can make out the approximate tree structure.

The effort for supporting ARM, including the development of about 100 lines of code, and verifying that all components "work as expected" took about 9.5 person-hours.

The effort required for AVR was about 3.5 person-hours.

6.3 Performance

Figure 7 captures the performance of the learning algorithm. Note that the chart shows both time and size on the Y-axis. The Y-axis is linear, while the X-axis is logarithmic. So, although the transducer time may visually appear to increase super-linearly, the actual increase is close to linear. The time needed by the algorithm is relatively small, about 10 minutes for the largest training data consisting of about 3.5M concrete pairs on x86. We have not optimized space usage yet, but it is reasonable already — about 200MB for the largest transducer we have built.

The performance charts for ARM are qualitatively similar, but the actual transducer construction is almost four times as fast, and space usage is about half as much as the x86 transducer.

Performance for Lifting Entire Binaries. We described a linear-time dynamic programming algorithm for lifting binaries. We have experimentally verified that its runtime increases roughly in proportion to the size of the binary that is lifted. Due to space constraints, we will just summarize its performance below.

Using this algorithm to translate all binaries on Ubuntu and Debian took approximately 8 hours each. The total time includes the disassembly time, which is approximately a fifth of the total time to lift a binary.

6.4 Correctness of translation

We experimentally evaluate correctness in three ways.

- *Semantic equivalence test:* We rely on ArCheck [26] to test the correctness of the mappings derived by our approach.
- *Consistency test:* Note that it is possible for distinct IL-snippets to result in the generation of the same assembly instruction A. In this case, A may be lifted to two distinct IL-snippets, which may seem like an error in lifting. We describe a systematic investigation to analyze them.
- *Loop-back test:* Given a binary with a list $\mathcal{A}$ of assembly instructions, lift it up to IL $\mathcal{I}$ using LISC, and then use GCC to generate code for $\mathcal{I}$ and check that this code matches $\mathcal{A}$.

6.4.1 Semantic equivalence test

Here, we rely on ArCheck [26], which defines two notions of correctness:

- *Soundness:* This notion permits IL to leave some effects of assembly instructions unspecified, e.g., the IL may indicate that certain registers (or memory locations) are *clobbered.* In particular, IL may over-approximate the effects of assembly, but it cannot miss any effects.
- *(Strong) Equivalence:* Here the semantics of IL and assembly must be identical. In other words, the IL precisely captures the effect of assembly.

ArCheck currently supports 140 of the basic x86 instructions. We used it to verify the soundness of the IL produced by LISC for all these instructions. While this does not eliminate the possibility of soundness errors for the remaining instructions, it does validate our approach, and suggests that such errors are unlikely.

The only soundness problem we have observed so far arises in the context of a call instruction. In particular, an IL-level call indicates the number of actual parameters, but this information is unavailable at the assembly level. We can address this problem by lifting a call instruction to indicate no parameters at the IL level.

We also used ArCheck to verify that the IL produced by LISC preserves strong equivalence for 109 of the 140 instructions. For the remaining 31, differences arise due to CPU flags as discussed below.

Impact on Binary Analysis/Instrumentation. Soundness is a requirement for most applications of binary analysis and instrumentation. Without soundness, the lifted IL would be incorrect in some cases, and lead to problems and failures. Equivalence is desirable but not critical for all applications. Indeed, we observe that equivalence failures arise primarily in the context of CPU flags, which are typically left unspecified after most arithmetic and logical instructions. GCC uses only a small subset of these instructions (e.g., the compare instruction) to translate conditionals, and hence specifies flags accurately for this subset.

Binary analyses typically rely on abstraction — what is needed is a sound approximation, which is what we focus

on. Instrumentation involves preserving original code, while adding new code to maintain metadata or to enforce policies. For instructions whose IL is not strongly equivalent, a simple liveness analysis of over-approximated fields can be performed. Specifically, a static analysis can be used to determine if subsequent instructions rely on any of state (say, flags) that is over-approximated in the IL; if not, then the approximation cannot have any negative effect. On the other hand, if there is a possible use, then the instrumentation system can be designed to ensure that the exact same instruction is emitted again, thereby preserving the original semantics.

6.4.2 Consistency test

The approach discussed above shows that every pair $\langle I, A\rangle$ produced by the code generator represents a sound translation of assembly code A to IL I. However, this leaves the following question open: if the code generator produces the same assembly code A for two distinct IL snippets I_1 and I_2, which of these should be used as the IL for A? The following possibilities arise:

- If any one of the IL translations preserves strong equivalence, then choose this IL as the preferred translation.
- Otherwise, there exist n sound ILs $I_1, \ldots, I_n$ for an assembly instruction A, but none of them are strongly equivalent to A. Since each of $I_1, \ldots, I_n$ is a sound approximation, their intersection is also a sound approximation of A. Moreover, this intersection is likely to be more complete than any individual I_j, and indeed, may be strongly equivalent.

For the ILs learned by LISC for x86, we made the following observations:

- Of the total 1187 mnemonics in x86, around 24% map to multiple possible ILs.
- Of these 24%, a vast majority of them — specifically, 22% — mapped to at least one IL that was strongly equivalent to the assembly instruction.
- For the remaining 2%, we apply the intersection operation. We found that for most of them, differences arise because the ILs define different subsets of CPU flags. By taking the intersection, we were able to achieve strong equivalence for most of these mnemonics.

Similarly for ARM, we made the following observations:

- Of the nearly 1200 assembly mnemonics, 15% (around 180) had multiple possible ILs.
- A vast majority of these — about 14% — mapped to at least one IL that was semantically-equivalent to the assembly instruction.
- For almost all of the remaining 1%, we applied the intersection operation and obtained semantic equivalence.

6.4.3 Loop-back test

There are factors to be noted in this context. First, note that GCC is unable to regenerate code for instructions such as `nop` that it never generates on its own. Second, due to the fact that two distinct assembly instructions A_1 and A_2 could be equivalent to the same IL, it is possible that lifting A_1 and regenerating code may result in A_2. We compensate for both factors in our test.

We have successfully performed this test on most of the data in P_{test} but not all. This is because GCC is not designed to accept externally produced RTL and translate it. As a result, our attempts to translate the lifted RTL can fail inexplicably in some cases. Moreover, the fact that GCC may not recognize some of the manually-introduced ILs (about 5% of assembly mnemonics) introduces an additional challenge. Our efforts have been focused on understanding GCC internals to model such instructions in a GCC-friendly way.

Despite these challenges, we have been able to carry out the loop-back tests successfully for several large binaries such as vim, gedit, python interpreter, latex, wireshark, mplayer, and packages from coreutils, binutils, and SPEC. In total, around 67% x86 binaries, 74% ARM binaries and 94% AVR binaries from our test data pass loop-back test. (Note that these numbers refer to entire binaries that passed the test. If we counted the instructions that were successfully regenerated, that number would be much closer to 100%.)

6.5 Application of extracted semantic model

We have applied our model for binary analysis, specifically, for implementing a robust shadow stack [41]. Shadow stack provides strong protection against stack smashing and ROP attacks, but the technique has not been deployed because previous approaches suffered from false positives on complex programs. In particular, complex code ends up using return instructions for purposes other than returning from a function call. A deep and precise static analysis is necessary to identify such uses of return instructions, and instrument them differently from normal returns. We applied the detailed instruction semantic models derived by LISC to perform this static analysis. As a result, we were able demonstrate the absence of false positives on a wide range of large and complex applications.

7. Related work

Binary analysis and instrumentation. Most previous binary analysis/instrumentation systems, including DynamoRio [13], Pin [36], QEMU [12], Valgrind [37], SecondWrite [7], CodeSurfer [11], UQBT [17] and many other systems [16, 21, 31, 33, 48] require a hand-written target instruction specification to drive the translator. We are not aware of other efforts that use compilers to fully automate this effort.

Some previous [15, 29] 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 [14], on the other hand, directly uses Valgrind's assembly to IR translator. These methods thus inherit any completeness issues from QEMU and Valgrind, which manifest as (a) support for only the most commonly used platforms, and (b) missing support for new and advanced instruction sets.

DisIRer [30] and Dagger [2] are two efforts that leverage compiler infrastructures to lift binaries to an IL. Dagger relies on the LLVM infrastructure, but their approach for lifting is manual. Hence it requires a considerable amount of additional code development, as well as a good understanding of LLVM internals.

DisIRer's goals are similar to ours: using MDs in reverse to lift binaries. However, as discussed earlier, there are many parts of MDs that are not specifications, and the only way to invert them is if we understand the C-code involved, and manually write functions to invert them. This requires a large amount of manual development effort for each architecture, thus negating the main advantage of using compiler MDs.

Language translation and transducers. Modern language translations efforts have been focused on natural languages. However, the complexity of natural languages implies that a certain rate of errors must be accepted. Our goals are thus closer to some of the earlier works, such as OSTIA [39] that emphasize error-free translation of simpler languages. (This relationship has already been discussed earlier in this paper.)

While we have defined tree transducers as automata, many earlier works have defined them using a formalism similar to grammars [43, 53]. Nevertheless, we are not aware of works that fully automate the learning of tree transducers. Language translation works that rely on tree transducers (e.g., [53]) still require humans to specify a (weighted) transducer, with the weights being derived using a learning process [25]. (Their choice of weighted transducers has again been influenced by the complexity of natural languages.)

While our goal has been to learn assembly to IL mappings, Derive [28] goes in the other direction: it learns the machine encoding of assembly instructions. There are some similarities between our work and theirs, e.g, they too want to identify simple transformations such as endian conversions. However, they do not concern themselves many of the problems we address, such as the learning of tree transducers (it is hard to represent machine code encodings as a parse tree), or avoiding strong assumptions about the structure of assembly code, or the interpretation of operands.

Collberg's [18] effort was more ambitious, attempting to learn MDs from a C-compiler. His approach is to construct very simple C-programs that perform operations such as an addition, analyze the instructions produced, and deduce the semantics. The approach was reported to work on common arithmetic operations. However, it is unclear how the approach can be generalized beyond simple arithmetic operations on registers.

Matching Automata and Decision Trees. The problem of efficiently matching terms has been studied extensively in the context of functional and logic programming, as well as automated theorem-proving. Given a set of terms $t_1, ..., t_n$, the problem is one of compiling the terms into an automaton that speeds up the identification of the term t_i that matches a given query term s. As compared to string matching, the key differences are (a) the need to handle variables, and (b) the ability to vary the order of traversing the nodes in the term so as to build more compact and efficient automata. Reference [45] discusses an array of techniques to handle a wide range of queries that arise in automated reasoning systems.

The criteria used for constructing compact matching automata are closely related to those used to construct decision trees in machine learning. While machine learning techniques rely on heuristics such as information gain that aren't guaranteed to construct optimal trees, it has been shown that under certain conditions, an *optimal* matching automaton can be constructed [46, 47]. Reference [51] extended the approach to matching network packets, and in that context, demonstrated major gains in terms of space and runtime over a previous technique [34] that relied on information gain. Many of the concepts used in this paper, such as mcp and $residue$, as well as the implementation of the $select$ function, have been based on that work [50, 51]. The main difference is that the current paper develops algorithms for learning translations on trees, while those previous works were focused on the (simpler) problem of term matching.

8. Conclusion

In this paper, we described an automated black-box approach called LISC for learning assembly-to-IL translators. Our experiments validate the hypothesis that by leveraging the knowledge already encoded in today's compilers, instruction semantics for diverse architectures can be obtained with relative ease. We showed that new architectures can be supported with as little as several hours of effort.

In a companion paper [41], we have demonstrated an application of the semantic model derived in this paper for performing a detailed static analysis on binaries. In our future work, we plan to study several additional applications that require accurate analysis and instrumentation of large binaries. Moreover, to aid other researchers working on similar problems, we have made an open-source version of LISC available for download from our laboratory web site [27].

Our approach can only recover the semantics of instructions generated by the compiler that feeds the ⟨assembly, IL⟩ pairs for our learning algorithm. Manual instruction modeling is required for those missing instructions. However, this factor does not reduce the value of our work: as our evaluation shows, manual effort is limited to about a twentieth of the instruction set. Moreover, these instructions tend to be very simple in nature (e.g., NOPs), so the effort needed is reduced by one to two orders of magnitude over a fully manual approach for modeling instruction semantics.

References

[1] Bad rounding in cvtsi2ss instruction. https://bugs.kde.org/show_bug.cgi?id=319393.

[2] Dagger. http://dagger.repzret.org.

[3] Incorrect decoding of vpbroadcastb,w reg,reg forms. https://bugs.kde.org/show_bug.cgi?id=340725.

[4] insn_basic test might crash because of setting and not clearing DF flag. https://bugs.kde.org/show_bug.cgi?id=326983.

[5] Power lxvw4x instruction uses 4 32 byte loads. https://bugs.kde.org/show_bug.cgi?id=339433.

[6] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.*

[7] Kapil Anand, Matthew Smithson, Aparna Kotha, Khaled Elwazeer, and Rajeev Barua. Decompilation to Compiler High IR in a binary rewriter. Technical report, Univ of Maryland, 2010.

[8] 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.

[9] Atmel. Atmel AVR 8-bit Instruction Set. www.atmel.com/images/Atmel-0856-AVR-Instruction-Set-Manual.pdf, 2014.

[10] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: Automatic Exploit Generation. In *Network and Distributed System Security Symposium*, 2011.

[11] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. CodeSurfer/x86 — a platform for analyzing x86 executables. In *Compiler Construction*, 2005.

[12] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, 2005.

[13] Derek L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Cambridge, MA, USA, 2004.

[14] 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.

[15] Vitaly Chipounov and George Candea. Dynamically Translating x86 to LLVM using QEMU. Technical Report EPFL-TR-149975, 2010.

[16] Cristina Cifuentes, Brian Lewis, and David Ung. Walkabout - A Retargetable Dynamic Binary Translation Framework. In *Workshop on Binary Translation*, 2002.

[17] 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*, 1999.

[18] 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, 1997.

[19] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of Internet worm epidemics. *ACM Trans. Comput. Syst.*, 26(4), December 2008.

[20] Jack W. Davidson and Christopher W. Fraser. Code Selection Through Object Code Optimization. *ACM Trans. Program. Lang. Syst.*, 1984.

[21] Thomas Dullien and Sebastian Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. 2009.

[22] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic Spyware Analysis. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, 2007.

[23] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, 2006.

[24] LLVM Foundation. The LLVM Compiler Infrastructure Project. http://llvm.org.

[25] Jonathan Graehl, Kevin Knight, and Jonathan May. Training Tree Transducers. *Comput. Linguist.*, 2008.

[26] Niranjan Hasabnis, Rui Qiao, and R. Sekar. Checking Correctness of Code Generator Architecture Specifications. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, 2015.

[27] Niranjan Hasabnis and R Sekar. LISC - Learning Instruction Semantics from Code Generator - software release. http://seclab.cs.sunysb.edu/seclab/lisc/.

[28] Wilson C. Hsieh, Dawson R. Engler, and Godmar Back. Reverse-Engineering Instruction Encodings. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, 2001.

[29] 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 *Parallel Processing (ICPP)*, 2011.

[30] 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, December 2010.

[31] 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, 2008.

[32] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure Execution via Program Shepherding. In *USENIX Security Symposium*, 2002.

[33] Julian Kranz, Alexander Sepp, and Axel Simon. GDSL: A Universal Toolkit for Giving Semantics to Machine Language. In *Programming Languages and Systems*, Lecture Notes in Computer Science. 2013.

[34] Christopher Kruegel and Thomas Toth. Using Decision Trees to Improve Signature-Based Intrusion Detection. In *RAID*, 2003.

[35] James R. Larus and Eric Schnarr. EEL: machine-independent executable editing. In *Proceedings of the SIGPLAN 1995 Conference on Programming Language Design and Implementation*, June 1995.

[36] 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, 2005.

[37] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, 2007.

[38] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed System Security Symposium (NDSS)*, 2005.

[39] J. Oncina, P. García, and E. Vidal. Learning Subsequential Transducers for Pattern Recognition Interpretation Tasks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 1993.

[40] GNU Project. The GNU Compiler Collection. http://gcc.gnu.org.

[41] Rui Qiao, Mingwei Zhang, and R. Sekar. A Principled Approach for ROP Defense. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, 2015.

[42] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, 2006.

[43] William C. Rounds. Mappings and grammars on trees. *Mathematical systems theory*, 4(3), 1970.

[44] Prateek Saxena, R Sekar, and Varun Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, 2008.

[45] R Sekar, IV Ramakrishnan, and Andrei Voronkov. *Term indexing, Handbook of automated reasoning*. Elsevier Science Publishers BV, Amsterdam, The Netherlands, 2001.

[46] 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, 1992.

[47] RC Sekar, R Ramesh, and IV Ramakrishnan. Adaptive pattern matching. *SIAM Journal on Computing*, 24(6):1207–1234, 1995.

[48] Alexander Sepp, Julian Kranz, and Axel Simon. GDSL: A Generic Decoder Specification Language for Interpreting Machine Language. *Electronic Notes in Theoretical Computer Science*, 2012. Third Workshop on Tools for Automatic Program Analysis (TAPAS' 2012).

[49] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, December 2008.

[50] A. Tongaonkar and R. Sekar. Condition Factorization: A Technique for Building Fast and Compact Packet Matching Automata. *IEEE Transactions on Information Forensics and Security*, 2016.

[51] 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, 2009.

[52] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *In Proceeding of the Network and Distributed System Security Symposium (NDSS)*, 2007.

[53] Kenji Yamada and Kevin Knight. A Syntax-based Statistical Translation Model. In *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, ACL'01, 2001.

[54] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Security and Privacy, 2009 30th IEEE Symposium on*, 2009.

[55] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, 2007.

[56] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R. Sekar. A platform for secure static binary instrumentation. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2014.

[57] Mingwei Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, 2013.