Checking Correctness of Code Generator Architecture Specifications [*]

Niranjan Hasabnis Rui Qiao R. Sekar

Stony Brook University, NY

Abstract

Modern instruction sets are complex, and extensions are proposed to them frequently. This makes the task of modelling architecture specifications used by the code generators of modern compilers complex and error-prone. Given the important role played by the compilers, it is necessary that they are tested thoroughly, so that most of the bugs are detected early on. Unfortunately, modern compilers such as GCC do not target testing of individual components of a compiler, but instead perform end-to-end testing.

In this paper, we target the problem of checking correctness of the architecture specifications used by code generators of modern compilers. Our solution leverages the architecture of modern compilers where a language-specific front-end compiles source-code into an intermediate representation (IR), which is then translated by the compiler's code generator into assembly code. Hence our approach is to test code generators by testing the equivalence of IR snippets and the corresponding assembly code generated. For this purpose, we have developed an efficient, *architecture-neutral* test case generation strategy. Using our prototype implementation, we performed correctness checking of 140 assembly instructions (80 general-purpose and 60 SSE out of around 600 x86 instructions) of GCC's x86 code generator, and found semantic differences in 39 of them, at least one of which has already been fixed by the GCC community in response to our report. We believe that our approach can be invaluable when developing support for a new architecture, as well as during frequent updates made to existing architectures such as x86 for the purpose of supporting new instructions.

1. Introduction

Modern instruction sets such as x86 and ARM are complex. Moreover, new instructions are added frequently. To get an idea of the complexity of modern instruction sets, consider the fact that Intel's instruction set reference manual (version 052) consists of around 1400 pages split across 3 volumes [3]. This makes the task of developing architecture specifications used by the code generators of modern compilers difficult. This situation is further complicated by the fact that existing approaches to develop architecture specifications mostly rely on manual modelling of instruction semantics. Given the complexity of modern instruction sets, the task of writing these specifications manually becomes cumbersome and error-prone, which eventually leads to bugs. Furthermore, the scale of the problem is vast, given the number of architectures supported by modern compilers. For instance, GCC-4.8.2 supports more than 40 target architectures, and the specifications for all these architectures constitute around 400K lines in total. GCC's Machine Description[1](MD) for x86 architecture alone has around 30K lines. We found that, over the period of 1 year from July, 2013 to July, 2014, around 25 bugs have been reported in GCC's bug tracking system, which have resulted in updates to its x86 machine description.

Modern compilers rely primarily on end-to-end testing to uncover potential bugs in code generators. In particular, a suite of programs are compiled and run, and their outputs matched with expected outputs. Any deviation observed from expected outputs points to a possible bug in the compiler, which then needs to be manually tracked down by the compiler writer. It is hard to perform comprehensive testing of the code generator using such an end-to-end approach. Modern compilers are complex, consisting of numerous components. Just the code optimizer consists of several phases and hundreds of optimizations. These components tend to have complex interactions, and end-to-end testing is not capable of exercising these components sufficiently. As a result, research efforts such as CSmith [23] have uncovered a significant number of bugs in today's compilers.

One could gain significantly more confidence in the code generator if there were means to directly test the code generator in a comprehensive manner. Unfortunately, today's compilers don't have code generator unit-testing frameworks that are up to the task[2]. We address this problem in this

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

[1] GCC's terminology for architecture specification

[2] As per our discussion with GCC developers [6], GCC does not have a comprehensive unit-testing framework for machine descriptions (architecture-specific component of its code generator), but instead relies on its main end-to-end test suite. To the best of our knowledge, LLVM too relies on similar end-to-end testing for uncovering code generator bugs.

paper, and develop a systematic, architecture-neutral framework for in-depth testing of code generators.

Our approach, called $ArCheck$ (Architecture Specification Checking), leverages the structure of a modern compiler to achieve architecture independence. In particular, today's compilers consist of front-ends, typically one per source-language, that translate source-code to a common intermediate representation (IR). This IR is then translated by different compiler back-ends (typically, one per target architecture) into assembly/machine code. Our approach is based on observing this IR-to-assembly translation, and testing if the two are semantically equivalent. In particular, for each IR-to-assembly translation used by the compiler, $ArCheck$ tests if the IR and assembly instructions have the same behavior. An important contribution of our approach is that it is very practical: given the challenging problem of *verifying* modern compilers, we develop an approach which can be easily applied by code generator developers at the time of development to detect and diagnose bugs in code generators and/or architecture specifications. We believe that such checking will in turn help developers in discovering many compiler bugs ahead of time, which, otherwise, could lead to compiler crashes, or worse, generation of incorrect code.

There are two main challenges that need to be addressed in $ArCheck$. First concerns the generation of test cases that will bring out the differences between IR and its assembly translation provided by the compiler. A simple approach is that of random test case generation, but such an approach is wasteful, generating many test cases that are meaningless, or otherwise don't contribute to finding differences in behavior. We overcome this challenge by developing a goal-oriented testing strategy that leverages the semantics of IR. In particular, since the semantics of IR is fixed and well-documented, our test case generator can rely on this semantics to generate useful test cases. Note that such an approach is consistent with our goal of architecture independence.

The second challenge is that of running the generated test cases on the IR and machine code, and compare the behaviors. Running the test case on the IR is conceptually straightforward — as mentioned above, there is a single IR for the compiler. "Running" a machine instruction, however, seems to require an architecture-specific solution. We show that, in fact, an architecture-neutral approach can be developed that makes of a real processor as an "oracle" to determine the behavior of a machine instruction.

We have so far applied $ArCheck$ to 80 general-purpose (out of around 200) and 60 SSE (out of around 70) x86 instructions. In total, these 140 instructions cover around 23% of around 600 total x86 instructions. This testing has uncovered 39 instances where the semantics of the IR and assembly instructions don't match. Not all of these are bugs, but most of them at least exposed assumptions made in the implementation that were otherwise unclear. This kind of programming practice can lead to latent bugs which are

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

Figure 1. x86 `add` assembly instruction and its IR

not easily caught. So far, we have reported one semantic difference to the GCC community, which has been accepted as a bug and has been fixed promptly.

1.1 Contributions.

- *Advancing the state of the art in compiler testing.* We believe that our first and the most important contribution is the advancement of the state of the art in compiler testing. Furthermore, to the best of our knowledge, ours is the first work in developing a systematic technique for testing the correctness of code generators.
- *Efficient and architecture-neutral approach to test generation.* Given the challenging problem of designing test cases to check the correctness of code generator mapping rules, we develop an efficient and architecture-neutral approach by leveraging observations about modern compilers. Furthermore, our approach is also very practical: it can be easily applied at the time of development to detect most of the code generator correctness issues. Moreover, our approach is compiler-neutral, and hence applicable to all modern compilers.
- *Evaluation.* Our prototype implementation for GCC has uncovered semantic differences in 39 of around quarter of total instructions from its x86 code generator. Out of 39 differences found, GCC community has accepted one reported difference as a bug and has fixed it promptly.

2. Problem Definition

We model the problem of checking the correctness of code generator as follows. Assuming that the code generator maps an IR instruction I to an assembly instruction A for some target architecture, is the semantics represented by I same as that of A? In the formulation below, we represent code generator architecture specification by M. M contains a set of mapping pairs between IR and assembly instructions, represented as $M = \{\langle I, A\rangle\}$.

Figure 1 is one concrete example of such $\langle I, A\rangle$ pair in the case of GCC and x86. The right hand side shows an x86 `add` instruction, while on the left hand side, it is the corresponding GCC IR, in the form of Register Transfer Language (RTL). GCC's RTL has a Lisp-like syntax, and RTL expressions are defined recursively. In this example, `set` and `plus` are RTL operators, `reg` and `const_int` are operands. `SI` indicates that the operands are in their single integer mode, i.e., they are treated as 32-bit integers. This RTL instruction explicitly specifies that register `eax` should be set as the sum of `eax` and constant 2. The `clobber` expression at the end simply says that EFLAGS are modified.

Now, in order to compare semantics of an assembly and IR instruction, first we need to formalize their semantics. Both IR and assembly instructions operate on processor state. Hence their semantics can be formalized in terms of the changes they make to the processor state. Note that IR's view of processor state can be more limited than the assembly-level view. For instance, the processor state will capture the exact state of every condition code flag after every instruction. In contrast, the code generator may be interested only in a subset of flags, and that too, after specific instructions such as those used for comparison. A natural approach, therefore, is to expose only a subset of the assembly-level state at the IR-level. More specifically, we consider user-space accessible CPU registers and memory to model state for this problem. This discussion leads to the following definition of the state and the semantics.

Definition 1 (Assembly State $\mathbf{S}_A$)**.** $\mathbf{S}_A$ *is modelled in terms of assignments to a set of variables* V_A *representing the processor's general-purpose user-space accessible registers and memory.*

Each variable $v \in V_A$ takes values from an appropriate domain, e.g., 8-bit or 32-bit integral values.

Definition 2 (IR State $\mathbf{S}_I$)**.** $\mathbf{S}_I$ *is given by an assignment of values to a set of variables* $V_I \subseteq V_A$ *representing the processor's state as viewed by the compiler. The domain of each variable in* V_I *is expanded over the corresponding variable in* V_A *to include* $\top$*, which denotes an unknown or unspecified value.*

The semantics of an instruction at the IR (or assembly) level can be understood in terms of how it modifies the processor state. We use the notation $I : \mathbf{S}' \longrightarrow \mathbf{S}''$ to denote that the execution of I in state $\mathbf{S}'$ leads to a new state $\mathbf{S}''$. Note that S' may assign $\top$ to some variables, and if any such variable is read by I then there does not seem to be a reasonable way to define the semantics of I. For this reason, we require $\mathbf{S}'$ to be *valid* for I.

Definition 3 (Processor state correspondance)**.** *States* $\mathbf{S}_A$ *and* $\mathbf{S}_I$ *are said to correspond, denoted* $\mathbf{S}_A \sim \mathbf{S}_I$*, if:*

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

Alternatively, one can say that $\mathbf{S}_I$ *is a conservative approximation of* $\mathbf{S}_A$*:* either they agree on the value of a state variable, or $\mathbf{S}_I$ leaves it unspecified. The latter choice is made by the developers of machine descriptions, who choose not to model the exact value of a state variable, but simply state that an instruction "clobbers" it. This imprecision is deliberate, as it makes it possible to develop machine descriptions that work across variants of a processor. They also provide a way out when the instruction set description is ambiguous or unclear about the final value of a state variable. Figure 1 serves as one such example: the IR specifies that EFLAGS are clobbered, without further information about which bits or how the bits are modified.

Definition 4 (Soundness of IR)**.** *I is said to be a* <u>*sound*</u> *abstraction of the semantics of A, denoted* $I \sim A$*, if the following condition holds for every state* $\mathbf{S_I}$ *that is valid for I and all states* $\mathbf{S_A} \sim \mathbf{S_I}$*:*

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

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

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

Definition 5 (Code generator testing)**.** *Let I be an IR snippet and A be the assembly code generated by a compiler for this snippet. The problem of code generator testing is to test every such* $\langle I, A\rangle$ *pair to determine if I is sound for A.*

3. Approach Overview

Ideally, we would obtain all possible IR snippets and their assembly translations that can ever be generated by a compiler, and check them all. One possible way to obtain such a complete set of mappings is to consult the architecture specifications used in a compiler, such as GCC's machine descriptions (MDs). Unfortunately, this requires significant manual effort for each architecture. Such effort is required for the following main reasons:

- Some parts of the architecture specifications are not specifications at all, but are programs written in C or C++. Clearly, it is not feasible to automatically extract the mappings that can result from all executions of such code snippets.
- The specification language is complex. For instance, GCC MDs consist of several architecture-specific constraints, conditions and predicates. The semantics of these constraints are again expressed as C/C++ code,

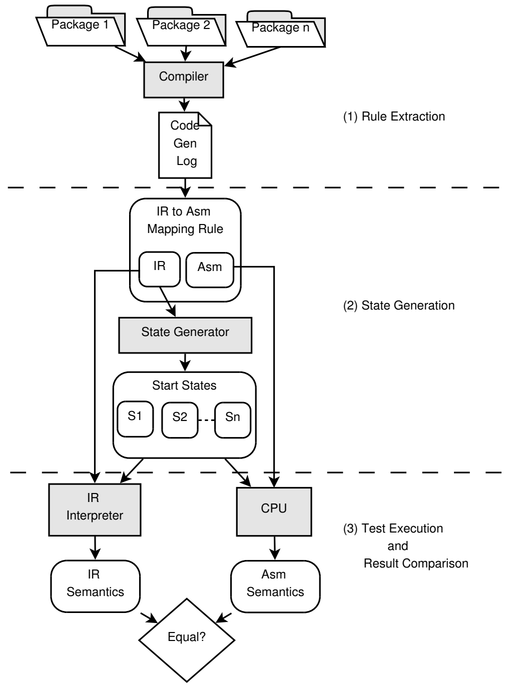

Figure 2. Design of $ArCheck$

making it difficult to automate the discovery of mappings that can result from the MD.

Indeed, it is nontrivial even for expert programmers to infer all the mappings that may be produced by an architecture specification[3].

Due to these difficulties, our approach does not attempt to test all possible IR to assembly mappings that may ever be produced. Instead, $ArCheck$ only tests the actual mappings generated by a compiler while compiling large software suites. By compiling large enough collections of programs (e.g., entire OS distributions) we believe that adequate coverage can be achieved using this approach.

After collecting $\langle I, A\rangle$ mappings using the above process, $ArCheck$ then proceeds to generate and execute test cases that are aimed at discovering soundness violations. In particular, we develop a test case generation strategy that is aimed at discovering states where I and A do not satisfy Definition 4.

Figure 2 illustrates the main steps used by $ArCheck$, which consists of the following phases:

- *Rule extraction:* This step is concerned with the collection of pairs $\langle I, A\rangle$ that need to be tested. As discussed above, this step is performed while treating the code generator as a black-box by compiling several software packages and observing and collecting the translations exercised during this process.
- *Start state generation.* For each pair $\langle I, A\rangle$ we then proceed to generate starting states $\mathbf{S_I}$ and $\mathbf{S_A}$ used in Definition 4. Given that a test strategy based on analysis of assembly instruction would become architecture-specific, $ArCheck$ avoids such strategy and instead relies on white-box analysis of IR instructions to generate $\mathbf{S_I}$, and Definition 3 to generate the corresponding $\mathbf{S_A}$. Fortunately, the fact that IR instructions expose all details of semantics of assembly instructions make such strategy feasible.
- *Test execution and result comparison:* In this step, we compute $\mathbf{S'_I}$ and $\mathbf{S'_A}$ of Definition 4 and check if $\mathbf{S'_I} \sim \mathbf{S'_A}$ as given by Definition 3.

Each of these steps is described in more detail below.

3.1 Rule extraction

For rule extraction, our approach is to compile several software packages and observe the translations of IR to assembly used by the compiler. To ensure different types of instructions are covered, our strategy is to compile different types of software packages (e.g., scientific packages that are most likely to use x86's SSE instructions) and to compile packages using different compiler options (e.g., with and without optimizations). With this strategy, we have been able to cover 75% of x86's general-purpose and SSE instructions.

When compiling large packages, the number of $\langle I, A\rangle$ mapping rules produced can range from millions to tens of millions. It is not feasible to test each of the pairs individually. Indeed, it is not even useful to test them all, as there is a substantial amount of overlap between different pairs, with numerous pairs being identical. So, we need to develop a strategy for selecting a subset of $\langle I, A\rangle$ mapping rules that are most useful to test. Currently, $ArCheck$ supports three selection strategies:

- *Mnemonic mode* selects a mapping rule for testing if the mnemonic of an assembly instruction from the mapping rule has not been tested previously in the same test run. This mode can be thought of as a quick way to test all assembly instructions from a log file which have a unique mnemonic. For instance, with mnemonic mode, we will skip "`add 4(%ebx), %eax`" from testing if we have previously tested "`add $2, %eax`."
- *Template mode* selects a mapping rule for testing if the "template" of an assembly instruction has not been seen previously in the same test run. We define "template" of an assembly instruction as the instruction with the register names and constants replaced by placeholders. Template mode of "`add $2, %eax`" is "`add I, R`." Thus, template mode will not test "`add $2, %ebx`" after testing "`add $2, %eax`," as they both have the same tem-

[3] These observations about machine descriptions apply not only to GCC but also to LLVM, whose Target Descriptions (TD) are partly in the form of specifications, with many concrete details incorporated directly into the architecture-specific components of LLVM code generator.

plate. However, "add $4(\%\texttt{ebx}), \%\texttt{eax}$" has a different template, and hence will be tested.

- *Instruction mode* selects a mapping rule for testing if the exact same mapping rule has not been seen previously in the same test run. For instance, $ArCheck$ will test "add $\$2, \%\texttt{ebx}$" in this mode even if it has previously tested "add $\$2, \%\texttt{eax}$."

Of the three modes, instruction mode is the most conservative, as it filters out only those instruction mappings whose testing is provably redundant. On the downside, because it does not filter out as many instructions as the other modes, it can take significantly more time for testing.

3.2 Start state generation

This step is concerned with the generation of $\mathbf{S_I}$ values to be used in testing a mapping rule $\langle I, A \rangle$. One way to generate $\mathbf{S_I}$ values is exhaustive enumeration: simply generate every possible $\mathbf{S_I}$ that is valid for executing I. However, note that even a single register can have 2^{32} states, so the number of distinct $\mathbf{S_I}$ values is far too large to admit exhaustive enumeration.

One way to control the state-space explosion of the exhaustive approach is to limit testing to a few randomly chosen input values. However, such a *random test generation* suffers from two significant problems. First, many operations may be defined only when certain conditions hold on their input. For instance, a memory access operator works correctly only if the address input corresponds to a valid region of memory, together with appropriate permissions. Random test generation may not respect such constraints and relationships and hence may generate many useless input combinations. Second, random generation approach is not systematic, and hence may not provide sufficient confidence in the test results. For instance, one may want to specifically verify the behavior of a division operation when the second operand is larger than the first, or is zero. To incorporate these preferences into the testing process, we therefore seek a start-state generation strategy that satisfies the following objectives:

- provide high confidence in the soundness of the mapping rule,
- utilize a small enough number of start states to ensure adequate performance, and
- ensure architecture-neutrality of the technique.

Note that we are achieving the third objective by basing the test generation strategy on I and $\mathbf{S_I}$ that have only a minimal dependence on the architecture, and generating the corresponding $\mathbf{S_A}$ from it. The first two objectives can be realized by partitioning the space of $\mathbf{S_I}$ values into a small number of "equivalence classes," and selecting one representative from each such class. (We sometimes use the term "interesting inputs" to refer to these representatives.) "Equivalence" in this regard means that I behaves the same way for all start states in the same class.

Note that an IR snippet I typically consists of more than a single operator. For instance, GCC's RTL (IR) snippet corresponding to the "add $\$2, \%\texttt{eax}$" instruction is "(set (reg eax) (plus (reg eax)(const_int 2))),"which contains multiple operators, including set and plus. For testing compositions of two IR-operators f and g, say, $f(g(...))$, it is no longer enough to know the representative inputs most suitable for testing f and g. Instead, we need to answer a more general question: what input values should be used to test g so that it yields outputs suitable for testing f. Based on this observation, we have developed a strategy that consists of the following steps:

- For each IR operator, we specify a set of equivalence classes for the *output* of an operator.
- For each IR operator, we also specify constraints relating its input and output, and propagate these constraints on the output of an operator to its inputs. In other words, we use these constraints to generate inputs that will yield the desired output.

The first step enables us to leverage an expert in IR to answer the difficult question of how to partition input (or output) space into equivalence classes. The constraint propagation step can also leverage this human expertise since this step can also be manually programmed. We describe these two steps in detail below.

Generating test cases using constraint propagation. Consider the addition operation discussed above. In this example, the plus operator in the IR performs a signed 32-bit addition. We can divide the output value into three classes:

1. a positive value in the range of $[1, \texttt{INT32\_MAX}]$,
2. 0, and
3. a negative value in the range of $[\texttt{INT32\_MIN}, -1]$.

As discussed before, this partitioning is guided by the intuition of an expert programmer well-versed with the IR. In this example, their choice may be guided by knowledge of the use of sign and zero flags in the IR, and the fact that they can have different values for each of the three categories of output values.

Given the range of values of outputs, one simple way to pick concrete values from these ranges is to focus on the boundary values, e.g., $\{\texttt{INT32\_MIN}, 0, 1, \texttt{INT32\_MAX}\}$.

An interesting and useful feature of our approach of partitioning outputs is that the interesting categories do not seem to depend very much on the operator. For instance, the above categories would make sense for most arithmetic operations. This factor reduces the manual effort needed to specify useful ranges of outputs.

Given the discussion about possible constraints on the inputs and outputs, it is easy to see that the test case generation problem that we are trying to solve can be formulated as a

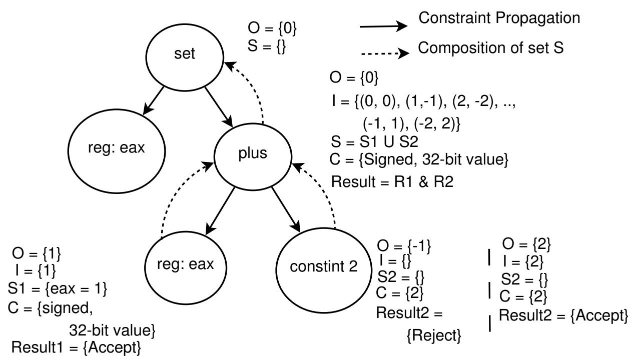

Figure 3. Constraint generation and propagation for start state generation

constraint satisfaction problem, and "interesting" test inputs for a mapping rule are simply solutions of such a problem.

A way to formulate constraint satisfaction problem from the white-box analysis of RTL instruction for "`add $2, %eax`" is shown in Figure 3. Note that the RTL instruction is represented as a tree. O represents the set of outputs of the RTL instruction. In the figure, we have set it to $\{0\}$, i.e, we are trying to solve the following constraint satisfaction problem: "find out a 4-byte signed integer value for `eax`, which, when added with 2, will yield the output of 0." S represents a state that satisfies these constraints. For the example in the figure, S contains an assignment of `eax`. Once we decide set O for a given mapping pair, the analysis starts by propagating the output at the root level to the inputs of the operator at the root. In this case, we have a `set` operator at the root level, and we consider its input as also its output. Thus, the set O is propagated to the node in Figure 3 labeled with the operator `plus`. Additional constraints that may be propagated include those based on operand types, e.g., signed 32-bit values.

Next, the constraint C on the output of `plus` operator is propagated to its inputs. In principle, this can result in combinations of two inputs to `plus` that yield a zero output. However, many of these combinations cannot be satisfied, e.g., the right child of `plus` can only accept a single constraint, namely, that its input is 2. Our constraint propagation algorithm implements this type of "pruning" automatically during the constraint propagation step.

An algorithm to generate "interesting" inputs for IR to assembly mapping rules is given in Figure 4. This algorithm starts with a call to `generate_start_states` function with ir as the IR instruction of a mapping rule, and it ends by returning set S, the set of start states to test a given mapping pair. Using function `get_outputs` and by passing it the top-level IR operator *op* and its type, it obtains a set of expected outputs for the given mapping pair. For instance, for RTL 32-bit signed `plus` operator, it would obtain $\{$`INT32_MIN`$, 0, 1,$ `INT32_MAX`$\}$ as the expected outputs.

Note that `get_outputs` is manually specified, and its implementation will typically be different for each IR operator. For this reason, we have not shown it in the figure. Since the

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

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

Figure 4. An algorithm for obtaining "interesting" inputs from an IR instruction of a mapping rule

number of operators in the IR is relatively small, the effort involved in this step is relatively small. Despite being manual, our approach does provide a systematic way to divide outputs into equivalence classes, while enabling us to leverage human knowledge and insight to minimize the number of such classes.

Once a set of expected outputs is obtained, function `get_inputs_for_output` is called to obtain a set of inputs for each of the expected outputs. In order to find inputs which produce the specified output, `get_inputs_for_output` first calls `get_inputs` by passing it the operator, the expected output value, and the type. For the "`add`" example, one of the calls would be `get_inputs`$($`plus`$, 0,$ `Int32`$)$. Note that the set C discussed in the Figure 3 can be seen as a collection of $\{$`plus`$,$ `Int32`$\}$ here. Note that `get_inputs` returns a set I of all 32-bit integers, which, when added, produce the expected output. I is a set of tuples whose arity matches that of the IR operator. For instance, for `plus`, tuples will have 2 elements, while for unary negation, tuples will have only 1 element.

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

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

3.3 Test execution and result comparison

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

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

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

Nonetheless, we made an important observation which helped us optimize simple approach considerably. Specifically, we observed that the difference between start and end state of memory needs to be calculated only if memory is updated by the test instruction. In other words, if memory is only read by the test instruction, then we do not need to compare memory snapshots at all. To put this observation to use, $ArCheck$ marks memory as read-only. With this setup, the framework can receive memory access fault in 2 cases: when test instruction accesses invalid memory location, and when the test instruction performs write access. We distinguish between these 2 cases by making memory read-write and re-executing the test instruction. If the framework receives one more fault, then it is because of first case now, and it is dealt as an exception case. If the framework does not receive any fault, then it is the second case, and the framework makes a note that memory snapshots must be compared in such situation. Note that as an additional optimization, the framework also notes the pages that will be modified and only considers those pages in comparing memory snapshots. Furthermore, if the framework does not receive memory fault in the first place, then it means that either the instruction does not access memory at all, or it performs read access. In both situations, we do not need to compare snapshots at all. Consequently, in both these situations, our system does not even snapshot memory at the end of the execution.

Once end states of assembly and IR executions are obtained, we follow Definition 3 and compare them.

4. Implementation

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

4.1 Obtaining code generator mapping rules

We developed a standard GCC plugin to obtain the code generator mapping rules during compilation. Implementation of the plugin took around 70 lines of C code. To collect RTL to assembly mapping for `foo.c`, one would use the command "gcc -dP -fplugin=rulcol.so -fplugin-arg-out=log.S foo.c", where -dP is a standard GCC option to dump an RTL corresponding to each assembly instruction as a comment in the output .S file. Thus, our tool easily integrates with configure and make based standard compilation process. Moreover, the plugin has very minimal dependencies on a particular version of GCC, so porting it to other versions of GCC is very trivial.

4.2 Obtaining start states for a mapping rule

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

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

4.3 Obtaining RTL semantics

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

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

4.4 Obtaining assembly semantics

Implementation of our user-level process monitoring framework relies on parent-child relationship and ptrace() interface of Linux. Test execution starts with the framework assembling a test program containing the test instruction and then creating a child process (using fork()) for executing the test program. The test program consists of a test instruction wrapped with 2 trap instructions. Trap instructions allow the parent (i.e., the framework) to intercept child's execution (by writing a trap handler) just before and after the execution of the test instruction, to set the start state, and to capture the end state of the execution. After fork(), child calls mmap() to map the binary encoding of the assembled test program and jumps to its beginning, while parent calls wait() and blocks itself. Once trap handler performs necessary actions such as setting or capturing the state, parent continues child's execution by sending PTRACE_CONT request to child. Along with trap, parent also handles other signals that might be raised by child by registering signal handlers for them. Although, cases leading to signals are not interesting for our purpose, parent handles them to exit gracefully.

Our user-level process monitoring framework is implemented in 2000 lines of C code and 50 lines of shell script C code has some architecture-specific features, such as the use of trap instruction, but is mostly architecture-neutral. The shell script uses GNU assembler gas, objdump, and objcopy to encode a test program containing a given assembly instruction wrapped in 2 traps and obtain its binary encoding.

4.5 Handling memory

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

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

4.6 Test execution and result comparison

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

5. Evaluation

We evaluated the effectiveness of our approach by testing x86 code generator of GCC-4.5.1 for general-purpose and SSE x86 instructions. (Supporting the rest of the instruc-

Description	*ArCheck*			**Random Testing**		
	Mnemonic	Template	Instruction	Mnemonic	Template	Instruction
# of Mapping Rules	140	1132	150K	140	1132	150K
# of Test Cases	1056	5762	421,090	1056	5762	421,090
% of Useful Test Cases	92	85	79	64	59	52
Time To Run Test Cases	5 mins 7 sec	31 mins	1 day 6 hrs	4 mins 10 sec	24 mins	1 day 1 hrs

Figure 5. Analysis of mapping rules obtained from GCC's x86 code generator logs and test cases generated for them

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

Figure 6. Statistics of evaluating mapping rules obtained from GCC's x86 code generator

tion set requires further engineering work on the RTL interpreter.) We conducted our experiments on 32-bit Linux running on a quad-core Intel Core i7 processor.

5.1 Testing setup

For comparison purposes, we implemented a random start state generation strategy. Note that a random strategy can be used to generate any number of start states, but to simplify comparisons, we generated the same number of start states as $ArCheck$. Our testing was done in all three of mnemonic, template and instruction modes.

We used GCC's x86 code generator to compile `openssl`, `opencv` and `ffmpeg` packages. We specifically chose these packages because they have a good mix of general-purpose and SSE x86 instructions. Summary of the collected mapping rules and start states generated for them is shown in Figure 5. Combined logs had roughly 150K unique mapping rules after eliminating exact duplicate rules. The 150K mapping rules had 140 unique mnemonics, and 1132 unique templates. These 140 mnemonics cover roughly 23% of total 32-bit x86 instructions. Breaking down these 140 mnemonics, we found that they covered 80 of around 200 general-purpose and 60 of around 70 SSE instructions. Of the remaining 120 general-purpose instructions, around 78 were system-related and I/O instructions, 12 were control-transfer[4] instructions, and 30 were not covered in the compilation logs.

Following observations can be made from these results:

- Average number of generated test cases in 3 modes differs. $ArCheck$ generated an average of 8, 5, and 3 test cases per instruction in mnemonic, template and instruction modes respectively. The average number for instruction mode is smaller because (1) commonly occurring instructions, such as "`mov`" for x86, dominate the total number of mapping pairs found in the logs, and (2) $ArCheck$ generates less number of test cases for "`mov`" than most others.
- Numbers of useful test cases generated by $ArCheck$ are significantly higher than those generated by random state generation approach. We call a test case "useful" if $ArCheck$ completes a test run for it without raising exceptions. For instance, when `eax` is 0, "`mov 0(%eax), %esp`" leads to a null-pointer dereferencing exception.

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

5.2 Detecting new semantic differences and soundness violations

Figure 6 compares $ArCheck$ with random testing in terms of different categories of instances in which semantics of IR and assembly instructions don't match. Note that for this evaluation, we checked for both soundness and semantic equivalence. Unlike soundness check, semantic equivalence is a strict check: it demands that the semantics of an assembly instruction is strictly modelled by an IR instruction. We enforced both these checks because we wanted to understand the type and the number of instances which belong to both categories. Note that the numbers reported in the Figure 6

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

are for semantic equivalence check. We will discuss which of these differences are soundness violations shortly.

Following observations can be made from these results:

- The semantic differences found by all three modes of *ArCheck* are considerably more than those found by random testing. Note that we have grouped semantic differences by mnemonics, and have counted 1 difference per group. We did this in order to eliminate duplicate counting of semantic differences for instructions with same mnemonic but different operand combinations. So 31 differences that we found using mnemonic mode of *ArCheck* means that 31 different x86 mnemonics were found having at least one semantic difference.
- Comparing results from mnemonic and template mode with those from instruction mode, we can see that the first 2 modes found fairly comparable number of semantic differences in significantly less time. This suggests that most of the semantic differences manifest for most operand combinations. The results are quite different for random testing, where detection is a lot less reliable.

  Though mnemonic and template mode did well in terms of finding semantic differences, instruction mode found the most in all tests. This suggests that testing a mapping pair with multiple operand combinations might help in finding more differences. On the other hand, instruction mode also took considerably longer to finish its run. We speculate that one of the major reasons for the amount of time taken can be explosion caused by the immediate values. To eliminate this source of explosion, but also to exploit the advantage of instruction mode over others, in the future, we can think of a modified-template mode, where only immediate values are replaced by placeholders. We speculate that such a mode should be able to finish its test run lot faster than that of the instruction mode, but at the same time, perform equally better as instruction mode.
- Unlike *ArCheck*, which detected the new bug in all three modes, only the instruction mode of random testing detected this bug. We will discuss this point shortly.

Figure 6 shows that instruction mode of *ArCheck* found the largest number (39) of semantic differences. We have classified these differences into 4 different categories:

- **D1:** *Imprecise modelling of* `EFLAGS`*.* This kind of difference arises frequently in GCC's x86 code generator when an RTL instruction does not capture precise bits of `EFLAGS` that are modified by the corresponding assembly instruction. For instance, the example of "`add $2, %eax`" discussed earlier falls into this category. This type of difference does not represent a soundness issue in the code generator because "`(clobber (reg EFLAGS))`" is actually an over-approximation of the instruction semantics.
- **D2:** *Incorrect value in the destination operand.* This kind of difference arises when an RTL instruction does not perform some operation, such as zeroing or sign-extending a value, that is performed by an assembly instruction. For instance, for `movzwl` instruction in the Figure 7, RTL simply moves lower 2 bytes of the source into destination, but fails to zero out upper 2 bytes of the destination. This kind of difference is a soundness violation.

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

Figure 7. Example of `movzwl` instruction and its RTL

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

Figure 8. Example of `shrdl` instruction and its RTL

- **D3:** *Incorrect operation in RTL.* This kind of difference arises when an RTL instruction performs different operation than the corresponding assembly instruction. This kind of difference represents a soundness violation. For instance, for `shrdl` instruction in the Figure 8, RTL uses arithmetic shift operator (`ashiftrt`), whereas assembly instruction performs a logical shift (`lshiftrt`).

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

  We reported this difference to GCC's bug reporting system. GCC developers have acknowledged that this is a bug, and have fixed it promptly. Details of our bug report can be found at [4].
- **D4:** *Update to a destination not specified.* This kind of difference arises when GCC uses some implicit assumptions not mentioned in the RTL specification. For

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

Figure 9. Example of `mull` instruction and its RTL

instance, "`mull %ebx`" instruction shown in Figure 9 modifies register pair [`edx:eax`], where the top 4-bytes of the product are stored in `edx`, and lower 4-bytes of the product are stored in `eax`. But RTL for `mull` stores lower 4-bytes of the result in `edx`, and says `eax` is clobbered.

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

Given our focus on a mature subset of x86 instruction set, and the use of perhaps the most-heavily used code generator, we did not expect to find many bugs. So, it should come as no surprise that most of the deviations we found were not soundness-related. But a significant minority — 7 deviations — do impact soundness (**D2** and **D3** categories). So far, only one of these have been acknowledged as a bug by GCC developers and fixed. We have not been able to exercise the rest of these 7 because they may be guarded by implicit assumptions in GCC code. It goes without saying that confidence in the correctness of the code generator would be significantly improved by avoiding reliance on such implicit assumptions, and instead updating the machine descriptions to eliminate the problem.

5.3 Detecting known soundness issues

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

We will now discuss some of these bugs and the way we detected[5] them.

- The bug reported in [5] is about failing to model possible updates to `EFLAGS` by an execution of `sbbl` instruction. *ArCheck* detected this bug because the start state generator initialized start states to unset bits of `EFLAGS` (because as per RTL semantics `EFLAGS` neither affected the end result, nor `EFLAGS` were clobbered by it), but the result produced by the assembly execution system had those bits of `EFLAGS` set. This bug belongs to category **D2**. Out of 15 bugs we collected, 3 were of this type.
- One older bug [2] (older than 4 years) detected by *ArCheck* was about an incorrect modelling of `movsd` SSE2 instruction. This instruction operates on two `XMM` registers and moves the lower 64-bits (double precision floating point number) of the source register to the lower 64-bits of the destination, and preserves the upper 64-bits of the destination. The issue was an incorrect use of RTL's `vec_merge` operator because of which exactly opposite semantics (assign the upper 64-bits of the source to the upper 64-bits of the destination, and preserve the lower 64-bits of the destination) was modelled. We detected this bug because *ArCheck* initialized the source and the destination `XMM` register with different values (because instructions which move values from a source to a destination have no "interesting outputs", so the state generator initializes the source and the destination with different byte patterns such as all bits set to 0 in the source and all bits set to 1 in the destination or vice versa[6]), and the result produced by the RTL interpreter was different than the one produced by the assembly execution system. This bug is a representative of the category **D3**. The list of 15 bugs had a couple of this type.
- An interesting type of bugs detected by *ArCheck* is syntactic bugs and syntactic errors which lead to soundness violations. For instance, the bug reported in [1] is about an incorrect order of operands in `bextr` assembly instruction generated by GCC's x86 code generator. Specifically, `bextr` instruction takes 3 operands of which first and third can only be registers, while the middle one can

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

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

be a register or memory operand. Modelling of this instruction in x86 machine description was incorrect — it allowed first operand to be either a register or memory. We detected this bug when we attempted to assemble the assembly instruction using `as`. Although, this particular bug is not a soundness violation, and $ArCheck$ is not designed to detect these type of bugs, some syntactic errors can lead to soundness violations. For instance, the bug reported in [7] is about missed square brackets around an immediate constant which changed the semantics of an instruction from moving a value from a memory location to moving an immediate value. Although the particular bug in [7] is a x86-64 bit bug, it is easy to see that it is indeed a soundness violation. The list of 15 bugs had 1 bug of this type.

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

6. Related Work

Compiler testing, verification, and bug finding. Compiler testing has been an active area of research for several decades. One of the earlier works, which described use of machine descriptions for compiler testing, is [20]. It describes an interesting approach to compiler testing by checking equivalence of source program and its object code by translating both to a common intermediate language. More recently, by randomly generating valid C programs, the CSmith system [23] performed a systematic end-to-end testing of modern compilers such as GCC and LLVM and found a number of bugs in them. Along with CSmith, other works such as [14, 17, 21, 24] have applied the technique of automatically generating valid C/C++ programs to test compilers. A fundamental difference between $ArCheck$ and all these works is the targeted testing of code generators performed by $ArCheck$. In contrast, systems such as CSmith are targeted at testing all components of the compiler. Thus, tools such as $ArCheck$ are complementary, and serve to provide much more in-depth testing of individual components of the compiler.

Compiler verification has also been a prominent area for compiler research with techniques such as *certified compiler* [12, 13] and *translation validation* [19, 22] being developed. CompCert [12, 13] has been a popular compiler verification work with promising results. However, scaling formal verification to production compilers such as GCC remains a challenge. While recent work has made significant progress in tackling components of production compilers, e.g., mem2reg optimization in LLVM [25], scaling these to components of the size of code generator represents a significant challenge. Testing-based approaches such as $ArCheck$ thus represent an important complementary approach that can work on industry-strength compilers.

Whereas $ArCheck$ is focused on the correctness of IR to assembly translations, the work of Fernández and Ramsey [9] targets the correctness of assembly to machine-code translations. They utilize a language called SLED (Specification Language for Encoding and Decoding) to specify instruction encodings at a high-level, and to associate assembly and machine code instructions. These specifications can then be used to generate machine code from assembly, or vice-versa. The main focus of their work is that of correctness checking of the mappings specified in SLED. This is accomplished using a combination of static checks and by comparing SLED-based translations with the translations produced by another well-tested independent tool such as the system assembler.

Approaches for test case generation. Generating better test inputs by improving test case generation strategies has been an area of active research. One can broadly classify existing testing strategies into black-box and white-box. Black-box testing strategies such as fuzz testing (random testing) [18] and grammar-based fuzz testing [15, 16], can also be applied for testing code generators. A drawback, however, is that it is difficult to ensure that all "relevant" and/or "interesting" input values have been tested. In contrast, $ArCheck$ has been explicitly designed to leverage the semantics of IR, and intuition and insight of human experts, to generate relevant/interesting test cases.

White-box testing strategies, such as symbolic execution [8, 10, 11], on the other hand, generate more "interesting" inputs because they treat the system-under-test as a white-box. Symbolic execution, in particular, seems best suited for our problem since it is a coverage testing technique; checking soundness of code generator mapping rules is a coverage testing problem. Nonetheless, given the simplicity of white-box static analysis of IR instructions, we preferred it over symbolic execution for building robust tools that can operate on production compilers.

7. Conclusion

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

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

References

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

[2] i386.md strangeness in sse2_movsd. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=14941.

[3] Intel 64 and IA-32 Instruction Set Reference, A-Z. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[4] RTL representation of i386 shrdl instruction is incorrect? https://gcc.gnu.org/bugzilla/show_bug.cgi?id=61503.

[5] Spaceship operator miscompiled. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=53138.

[6] Testing machine descriptions. https://gcc.gnu.org/ml/gcc/2014-03/msg00434.html.

[7] x86_64-linux-gnu-gcc generate wrong asm instruction MOVABS for intel syntax. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=56114.

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *OSDI*, 2008.

[9] Mary Fernández and Norman Ramsey. Automatic Checking of Instruction Specifications. In *ICSE*, 1997.

[10] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *PLDI*, 2005.

[11] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 1976.

[12] Xavier Leroy. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. POPL, 2006.

[13] Xavier Leroy. A Formally Verified Compiler Back-end. *J. Autom. Reason.*, 2009.

[14] Christian Lindig. Random Testing of C Calling Conventions. In *AADEBUG*, 2005.

[15] Rupak Majumdar and Ru-Gang Xu. Directed Test Generation Using Symbolic Grammars. In *ASE*, 2007.

[16] Peter M. Maurer. Generating Test Data with Enhanced Context-Free Grammars. *IEEE Softw.*, 1990.

[17] William M. McKeeman. Differential Testing for Software. *DIGITAL TECHNICAL JOURNAL*, 1998.

[18] Barton P. Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*, 1990.

[19] George C. Necula. Translation Validation for an Optimizing Compiler. In *PLDI*, 2000.

[20] H. Samet. A Machine Description Facility for Compiler Testing. *Software Engineering, IEEE Transactions on*, 1977.

[21] Flash Sheridan. Practical Testing of a C99 Compiler Using Output Comparison. *Softw. Pract. Exper.*, 2007.

[22] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. Evaluating Value-graph Translation Validation for LLVM. In *PLDI*, 2011.

[23] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *PLDI*, 2011.

[24] Chen Zhao, Yunzhi Xue, Qiuming Tao, Liang Guo, and Zhaohui Wang. Automated Test Program Generation for an Industrial Optimizing Compiler. In *Automation of Software Test, 2009. AST '09*, 2009.

[25] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formal Verification of SSA-based Optimizations for LLVM. In *PLDI*, 2013.