PAPER 3

ABSTRACT

This paper describes a cross-version compiler validator and measures its effectiveness on the CLR JIT compiler. The validator checks for semantically equivalent assembly language output from various versions of the compiler, including versions across a seven-month time period, across two architectures (x86 and ARM), across two compilation scenarios (JIT and MDIL), and across optimizations levels. For month-to-month comparisons, the validator achieves a false alarm rate of just 2.2%. To help understand reported semantic differences, the validator performs a root-cause analysis on the counterexample traces generated by the underlying automated theorem proving tools. This root-cause analysis groups most of the counterexamples into a small number of buckets, reducing the number of counterexamples analyzed by hand by anywhere from 53% to 96%. The validator ran on over 500,000 methods across a large suite of test programs, finding 12 previously unknown correctness and performance bugs in the CLR compiler.

Categories and Subject Descriptors

D.2.4 [**Validation**]: Compiler Validation

General Terms

Verification, Reliability

Keywords

Compilers, Verification, Translation Validation

1. INTRODUCTION

Compilers have grown enormously complicated in response to demands for new language features and better optimizations. When adding new features and optimizations to an existing compiler, there's always a danger of introducing new bugs into the compiler. Such bugs might cause existing programs to fail when recompiled with a new version of a compiler; this danger is particularly worrisome for just-in-time (JIT) compilers, where installed programs might suddenly stop working due to a JIT compiler upgrade. Worse, a JIT compiler bug might introduce a security vulnerability into a system that trusts the compiler for secure execution of downloaded code, as in the case of JavaScript engines, Java virtual machines, and .NET Silverlight virtual machines.

To prevent new bugs in existing compilers, compiler developers typically run the compiler on large suites of regression tests. Even very large test suites will not catch all compiler bugs, though [17]. Therefore, many researchers have turned their attention to static validation techniques for compilers, using theorem proving technology to perform *compiler verification* [8] or *translation validation* [11, 9, 16, 14]. Compiler verification uses a theorem prover or proof assistant to verify a compiler correct, once and for all, so that all compiler output is guaranteed correct for all valid source programs. Translation validation, on the other hand, runs a theorem prover after each compilation to check that the output of the compiler is semantically equivalent to the source program.

Compiler verification requires verifying the compiler implementation, which is difficult for large compilers. So far, compiler verification has scaled to moderately-sized research compilers [8]. Although larger compilers have been formalized to some extent [18], they have not been proven correct, and it remains a daunting task to do so. By contrast, translation validation requires only verification of the compiler output, not the compiler implementation. As a result, translation validation has been used to test gcc and LLVM, both large, widely-used compilers [9, 16, 14].

Nevertheless, translation validation has long suffered from false alarms: because program equivalence is undecidable, the theorem prover often fails to prove equivalence between the compiler output and the source program, even when equivalence actually holds. Necula [9] reports false alarm rates of up to 3% per function for *individual compiler optimization passes*, while Tristan et. al. [16] and Stepp et. al. [14] report false alarm rates of 10%-40% for combined series of optimizations. When running the theorem prover on tens of thousands of test functions, it's currently very difficult to investigate all of these failures for all optimization passes by hand. To make static compiler validation practical for day-to-day compiler development, we must reduce the burden of false alarms on compiler developers.

This paper proposes two techniques for static compiler validation with a low false-alarm burden:

- We compare multiple versions of the compiler assembly language output, rather than comparing the source

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC/FSE '13, August 18-26, 2013, Saint Petersburg, Russia
Copyright 2013 ACM 978-1-4503-2237-9/13/08 ...$15.00.

code to the assembly language code. We test several incarnations of this idea: comparing the output of multiple versions of a compiler over time (specifically, across check-ins to the compiler implementation over many months), comparing the output of a compiler for two different architectures (ARM and x86), comparing the output of a compiler in JIT mode to the compiler in mostly-ahead-of-time mode (based on the MDIL [12] machine-dependent intermediate language), and comparing the output of a compiler with different optimizations enabled.

- When an alarm is raised, our tool automatically produces counterexample traces showing values that cause divergent behavior in the two assembly language programs. The tool then uses the traces to automatically infer the suspected root cause of the divergent behavior. The root causes classify the counterexamples into a small number of buckets, greatly reducing the human effort needed to investigate equivalence failures.

Comparing multiple versions of a compiler over time produces a particularly low false alarm rate, because much compiler development consists of refactoring of the existing compiler implementation or adding new features to the compiler. In these cases, the goal is to *not* disturb the compiler output for existing programs, to minimize the risk of new bugs that break compilation of existing programs. Thus, across check-ins to the compiler, we often find that the compiler assembly language output remains syntactically identical. Even when the output is syntactically different, the differences are often easy for a theorem prover to check for semantic equivalence.

One concern with this cross-version validation is a lack of "ground truth": the validation may miss a bug that appears in both the old version of a compiler and the new version. Thus, one might worry that even if cross-version validation is easier to use than traditional translation validation, it might provide weaker correctness guarantees. We believe, though, that cross-version validation and translation validation have complementary strengths. For example, a developer might use translation validation once to provide "ground truth" for one version of the compiler, but then use cross-version validation to check subsequent versions of the compiler relative to the original translation-validated version. In addition, existing implementations of translation validation don't check the complete translation from source code to assembly language code. Instead, they usually assume that the source code has been parsed into an abstract syntax tree, and thus miss bugs in the compiler's parser. Furthermore, they typically check the intermediate compiler representation, but not the generation of assembly language code [9, 16, 14]. By contrast, cross-version validation catches any changes to the compiler that cause semantically differing output, including changes in parsing and assembly-language generation.

The rest of this paper describes our implementation of cross-version validation and root-cause analysis, and describes their application to a real-world optimizing compiler, the .NET Common Language Runtime (CLR) JIT compiler:

- Section 2 describes the implementation of our cross-version validator, based on the SymDiff [7] symbolic differencing tool, the Boogie [1] program verifier, and the Z3 [4] automated theorem prover. These tools make the implementation of the validator considerably easier, allowing us to focus our attention on the semantics of assembly language instructions, the model of memory, and the model of the run-time system.

- Section 3 describes our root-cause analysis, based on the output from the SymDiff, Boogie, and Z3 tools.

- Section 4 presents experimental results containing measurements of false alarms across 7 months of compiler check-ins for 2 source control branches of the CLR under active development, including x86-to-ARM validation, JIT-to-MDIL validation, and optimized-to-unoptimized code validation. The measurements show that the average false alarm rate for month-to-month differences is 2.2%, an order of magnitude lower than the 10%-40% false alarm rate for recent translation validators [16, 14] and the 9%-31% false alarm rate for month-to-month syntactic identity checking. The measurements also show the effectiveness of root cause analysis, which grouped anywhere from 53% to 96% of the false alarms into a small number of buckets (with no false alarms ever placed in the wrong bucket).

One limitation of our current work is that the validation is not 100% sound; section 2 discusses the reasons for unsoundness, such as assumptions about aliasing, assumptions about the run-time system, and unsound modeling of loops. Nevertheless, even without an absolute guarantee of correctness, the tool is still useful for catching bugs; measurements in section 4.3 show that the tool is very effective on real code with artificially injected bugs, with just a 3% false negative rate, while section 4.6 describes 12 real bugs found with the tool during CLR development and testing.

2. IMPLEMENTATION

Our validator takes assembly language programs generated by two versions of the CLR compiler and tries to prove the assembly language programs equivalent. Each assembly language program consists of a series of compiled method bodies (functions), each consisting of a sequence of assembly language instructions. As in earlier translation validation work [9, 16, 14], our validator works on one method body at a time. In other words, if compiled program P consists of method bodies M1..Mn and compiled program P' consists of method bodies M1'..Mn', the validator tries to prove M1 equivalent to M1', M2 equivalent to M2', etc.

The validator converts each method body into a procedure in the Boogie [1] programming language. Boogie is a simple imperative language supporting assertions (e.g. preconditions, postconditions, loop invariants) that can be statically checked for validity using the Boogie verification generator. Boogie is typically not used to write programs directly, but instead is used to encode programs from other languages, such as C and assembly language. Boogie provides basic statements, such as if/else, goto, and assignment, and basic expressions, such as variables, integers, bit vectors, and function applications. These constructs can encode a variety of other programming language constructs. This section describes the encoding of assembly language used by our validator, including encodings of arithmetic, memory, and calls to other methods and run-time system functions. It then describes how the SymDiff tool, Boogie tool, and Z3 automated theorem prover process the encoded assembly language.

2.1 Arithmetic

As a simple example, the validator encodes the x86 instruction “add eax, ebx” as a single Boogie statement, “eax := ADD(eax, ebx)”. Here, ADD is a Boogie function and eax and ebx are Boogie variables:

```
type val;
var eax:val, ebx:val, ecx:val, edx:val;
var esi:val, edi:val, esp:val, ebp:val;
function ADD(x:val, y:val):val { ...  }
```

There are several possible definitions of ADD. First, it could be uninterpreted, declaring just that there exists an ADD that computes an output deterministically as a function of its two inputs:

```
function ADD(x:val, y:val):val;
```

However, an uninterpreted ADD would lack properties such as commutativity and associativity. Thus, if method Mk contained “add eax, ebx” while method Mk’ contained “add ebx, eax”, the validator would be unable to verify that both add instructions compute the same value. It’s possible to specify some of these properties as Boogie axioms:

```
axiom (forall x:val, y:val::ADD(x, y) == ADD(y, x));
```

Rather than axiomatizing arithmetic operations from scratch, though, it’s more efficient to use the arithmetic functions built into modern theorem provers like Z3 [4], either using Z3’s big-vectors or Z3’s integers:

```
type val = bv32;
function ADD(x:val, y:val):val { $add32(x, y) }

type val = int;
function ADD(x:val, y:val):val { x + y }
```

The first definition declares the “val” type to be a 32-bit “bit vector” value, which exactly matches the register sizes on 32-bit architectures, such as x86. (The $add32 function is the theorem prover’s underlying addition operation for 32-bit numbers.) The second definition declares the “val” type to be an arbitrarily sized mathematical integer. Clearly, the second definition is only an approximation of the underlying hardware register values, which are not arbitrary-sized. However, an automated theorem prover usually reasons more quickly about mathematical integers than bit vectors. Thus, there is a trade-off between a sound representation of the hardware and theorem proving performance. After implementing both definitions, we found that bit vectors caused the theorem prover to time out on medium-sized and large methods, making the validator only useful for small methods. Therefore, we chose to model machine arithmetic using mathematical integers, even though this can lead to both extra false alarms and missed bugs.

One drawback of using mathematical integers is a lack of built-in definitions for bitwise operations, such as bitwise-and and bitwise-exclusive-or. Compilers often make clever use of such operations in generated assembly code. For example, x86 compilers often exclusive-or a register with itself to produce the value 0, since the exclusive-or instruction happens to have an efficient x86 encoding. Compilers may also use bitwise-and to align the stack pointer to 8-byte or 16-byte boundaries, or to truncate 32-bit values to 16 bits or 8 bits. To support such idioms, we declare a small set of axioms for bitwise operations (e.g. the exclusive-or of an integer with itself equals 0).

The validator also encodes the x86 floating point stack and status flags; we omit these details for brevity.

2.2 Memory

Boogie provides an array type, written “[t1]t2”, that maps values of type t1 to values of type t2, along with expressions to read array elements and update arrays with new elements. The validator encodes memory as an array mapping addresses to values. However, reasoning effectively about memory loads and stores requires some understanding of the memory’s structure and potential aliasing between different pointer values. For example, consider the following Boogie encoding of the x86 instruction sequence “mov [esp + 12], eax; mov [ebx + 4 * ecx + 8], edx; mov eax, [esp + 12]”:

```
type val = int;
var Mem:[val]val;
...
Mem[esp + 12] := eax;
Mem[ebx + 4 * ecx + 8] := edx;
eax := Mem[esp + 12];
```

If ebx points to a heap object (e.g. an array) while esp points to a stack frame, the compiler will assume that addresses esp + 8 and ebx + 4 * ecx + 12 do not overlap, so the final value of eax in this code should equal the initial value. To prove that the code does not modify eax, the validator needs to track the compiler’s non-aliasing assumptions.

To represent such non-aliasing information, we model memory as a set of disjoint regions, with one region per stack frame, one region per heap object, and one region for static fields. We assume that stores to one region will not affect loads from another region. We also assume that adding an offset to an address in one region produces an address in the same region, so that esp + 12 resides in the same region as esp and ebx + 4 * ecx + 8 resides in the same region as ebx. These assumptions are unsound; a big enough ecx will cause ebx + 4 * ecx + 8 to overlap the stack frame. (For type-safe code with array bounds checks, the compiler enforces the soundness of these assumptions, but for unsafe code like C++ and unsafe C#, the compiler makes these assumptions without enforcement.) To track the region associated with each address, we define each register value to be a pair of a region identifier and an integer value:

```
type ref; // region identifier
type word = int;
type val; // pair of (ref, word)
function Val(r:ref, i:word):val;
function ValRef(v:val):ref;
function ValWord(v:val):word;
```

The Val constructor creates a value from a region and an integer word, while ValRef and ValWord extract the region

and integer word components from a value. Non-address values use a special null region identifier as their region component. We lift arithmetic operations to work on region-word pairs. For example, we define ADD as:

```
function ADD(x:val, y:val):val {
  ValWithRegion(ValRef(x), ValRef(y),
        ValWord(x) + ValWord(y))
}
```

where ValWithRegion builds a value for x + y with a region chosen as follows: x's region if y's region is null, y's region if x's region is null, and a dummy region if both x and y have non-null regions. This allows the validator to tell, for example, that ADD(esp, Val(nullRegion, 12)) keeps esp's region, and when ebx is an address and 4 * ecx is a non-address, ADD(ebx, 4 * ecx) keeps ebx's region.

Consider the x86 instruction "mov [esp + 12], eax" again. After incorporating regions, the following Boogie statement expresses the assignment to memory:

```
Mem[ADD(esp, Val(nullRegion, 12))] := eax;
```

This statement is still inaccurate in one way: storing a 32-bit value to memory should actually update 4 separate memory locations, esp + 12, esp + 13, esp + 14, and esp + 15, placing 8 bits into each location, rather than putting the whole 32-bit word in the single location esp + 12. Although we did implement this byte-accurate model, we found that the theorem prover performance degraded significantly relative to the less accurate word-oriented model shown in the statement above, leading to excessive theorem prover timeouts. Therefore, this paper's experiments use the unsound word-oriented model (which may cause missed bugs or false alarms) rather than the byte-accurate model.

The validator can handle complex instructions that modify multiple words of memory, such as the x86 "rep movs" and "rep stos" instructions (often used by the CLR compiler to copy or initialize large values). For example, the "rep stosb" instruction fills memory addresses edi ... edi+ecx-1 with a byte value from register al, expressed in Boogie as:

```
function REP_STOSB(Mem:[val]val, ecx:val, edi:val,
    al:val):[val]val;
axiom (forall Mem:[val]val, ..., al:val, i:val::
  (...((ValRef(edi) == ValRef(i)
    && ValWord(edi) <= ValWord(i)
    && ValWord(i) < ValWord(edi) + ValWord(ecx))
    ==> REP_STOSB(Mem, ecx, edi, al)[i] == al)...))
```

2.3 Control Flow

Boogie supports goto statements and if/else statements, so it is straightforward to encode assembly language jump and conditional jump instructions. For call instructions, the validator uses uninterpreted functions to model the call's effect on the registers and heap. For example, it encodes the instruction "call eax" as:

```
function CallMem(addr:val, heapSig:int, args:list):
         [val]val;
function CallOut(addr:val, heapSig:int, args:list):
         [int]val;
...
Heap := CallMem(eax, HeapSig(Mem),
      Cons(arg1, ...Cons(argm)...));
ret1 := CallOut(eax, HeapSig(Mem), ...)[1];
...
retn := CallOut(eax, HeapSig(Mem), ...)[n];
Mem := ...combine stack, Heap...
...  assign ret1...retn to registers, stack slots...
```

Here, arg1...argm are the arguments to the call, and ret1...retn are the return values from the call. By-reference parameters are treated as both arguments and return values. To generate the encoding, the validator must know how arguments and return values are laid out in registers and stack slots. Therefore, the validator requires a type annotation on each call instruction (generated by the CLR in our experiments). The validator uses this type and the CLR's calling conventions for primitive types, structs, generics, pass-by-reference parameters, and so on, to compute which registers and stack locations hold arguments and return values.

We assume that the call's output depends on the state of the heap, but not on the state of the caller's stack frame (except for by-reference arguments from the stack, which explicitly appear in the list arg1...argn). Therefore, we do not make CallMem a function of the whole memory state Mem, but rather just a function of the heap portion of Mem. The function HeapSig strips away the stack portion of memory, compressing the heap portion of memory into an integer signature; HeapSig(Mem1) equals HeapSig(Mem2) if and only if Mem1 and Mem2 are identical at all non-stack addresses. HeapSig considers Mem1 and Mem2 equal regardless of the order of stores to Mem1 and Mem2, allowing the compiler to reorder heap stores without upsetting the validator.

Since CallMem and CallOut are uninterpreted, the validator has no information about the internal behavior of the called function. Thus, if method Mk calls method Mj, while method Mk' inlines a call to Mj, the validator will fail to prove Mk and Mk' equivalent — the validator sees the uninterpreted functions for Mj as different from the concrete inlined statements in Mj. To avoid false alarms, we currently keep inlining disabled when validating the CLR compiler.

While keeping calls uninterpreted is generally sound, leading to possible false alarms but not to false negatives, in theory it may sometimes be unsound to represent calls to run-time system functions as uninterpreted functions. This is because the run-time system may change behavior from version to version, so it might not always be the case that calling the same run-time function with the same arguments produces the same values. To be completely sound would require an accurate model of every version of the run-time system, including the behavior of casts, allocation, memory barriers, lazy static initialization, lazy JIT compilation, etc.. In practice, we have written models for a few run-time system functions (mainly write barriers and 8-byte arithmetic), but have otherwise left the run-time system uninterpreted.

Currently, the validator has only limited support for exception handling. The validator assumes that thrown exceptions exit the method; it lacks the control-flow edges into the method's exception handlers. Because if this, the validation may fail to notice semantic differences in exception handlers. Section 4 measures the impact of this unsoundness.

2.4 Running Symdiff, Boogie, and Z3

After encoding each method Mk and Mk' in Boogie, the validator invokes the SymDiff symbolic differencing tool [7] to compare the Boogie encodings for semantic equivalence. SymDiff combines the encodings of Mk and Mk' into a single block of Boogie code that executes Mk on some memory Mem and register state eax...esp, executes Mk' on an independent memory Mem' and state eax'...esp', and then asserts the final state of Mk and Mk' is the same:

```
...encoding of Mk...
...encoding of Mk'...
assert HeapSig(Mem) == HeapSig(Mem');
assert returnVal == returnVal';
assert calleeSaved == calleeSaved';
```

The assertions say that for Mk and Mk' to be considered equivalent, the final heap state must be the same, the return values must be the same, and the callee-save state must be the same. The variable calleeSaved is a boolean that is true if the method correctly restores callee-save registers (e.g. ebx, ebp, etc.) and returns to the correct return address. Initially, we created a separate assertion for each callee-save register (assert ebx == ebx', assert ebp == ebp', etc.), but found that this didn't quite work for our cross-architecture experiments, because the x86 method and the ARM method have different sets of callee-save registers.

SymDiff feeds the combined Boogie code to the Boogie verification tool, which attempts to prove that the assertions hold. The Boogie tool converts the assertions into a "verification condition" — a pure logical formula that encodes both the assertions and the meaning of the statements in Mk and Mk'. The Z3 automated theorem prover attempts to prove that the verification conditions are valid. If this proof succeeds, then the validator deems Mk and Mk' equivalent.

For code with loops, the verification condition generation and proof is not entirely automatic — Boogie needs program annotations in the form of "loop invariants" to avoid generating an infinite verification condition. Generating loop invariants is undecidable, but earlier work by Necula [9] describes how symbolic evaluation can be used to build simulation relations that serve as potential loop invariants. For ease of implementation, however, the validator currently employs a more expedient solution: in the spirit of automated, unsound bug finding tools [2], the validator simply eliminates loops by unrolling them n times, ignoring any behaviors past the n'th iteration (n = 2 in the experiments in this paper). While this is certainly unsound, because it fails to capture semantic differences that require more than n iterations to appear, we still observe a fairly low rate of false negatives (see section 4), as most differences are observable after only one or two iterations. After this loop unrolling, the encodings of Mk and Mk' are loop-free and thus require no loop invariants during verification condition generation.

3. ROOT CAUSE ANALYSIS

If the proof of the verification condition fails, Z3 generates a model (a counterexample) showing values of variables that make the verification condition false. The SymDiff tool parses this model and annotates the variables in Mk and Mk' with values from the model. This annotation provides a human-readable trace through each method body, helping to see where the values in the variables of Mk differ from the values in the variables of Mk'. Nevertheless, for long methods, the trace may contain hundreds of values from the model, which may take 5 or 10 minutes for a human to make sense of; when analyzing hundreds of false alarms, this time adds up. Therefore, it's useful to provide an automated mechanism for finding the location in the trace most responsible for the semantic difference.

This section briefly describes our root-cause analysis, which attempts to find the most relevant values, highlight them in the trace, and provide information useful for grouping related counterexamples into buckets. Our main root cause analysis technique works by following counterexample traces backwards through the dataflow graphs of the two methods being compared. For comparison with other known techniques, we also implemented a second analysis based on MAX-SAT [6]; space constraints preclude a full description of this, but Section 4.5 summarizes the results. Both analyses work on the Boogie code generated from the assembly language code. As a running example, consider the following Boogie programs, with input i and outputs z1 and z2:

`x1 := F(i);`	`x2 := F(i);`
`if (i > 10)`	`if (i <= 10)`
`  y1 := G1(x1);`	`  y2 := 1 + x2;`
`else`	`else`
`  y1 := x1 + 1;`	`  y2 := G2(x2);`
`z1 := H(x1, y1);`	`z2 := H(x2, y2);`

This example contains only one meaningful difference between the two programs: the statement `y1 := G1(x1)` differs from `y2 := G2(x2)` because the functions G1 and G2 differ. In this example, we assume that the functions F, G1, G2, and H are *uninterpreted*: there are no axioms constraining their outputs, so the theorem prover must assume that the outputs of the functions may differ if the function names differ or any inputs to the functions differ. By contrast, functions like "+" and "<=" are *interpreted*: the theorem prover knows that 1 + x = x + 1 and that x > 10 is the negation of x <= 10. During root cause analysis, we treat functions such as load, store, CallMem, and CallOut as uninterpreted.

The key insight of our analysis is that while identifying root causes is difficult in general, uninterpreted functions are relatively easy to reason about. For example, the statements `z1 := H(x1, y1)` and `z2 := H(x2, y2)` both call the same uninterpreted function H. Given a counterexample showing that z1 and z2 differ, we can safely conclude that differing inputs to H cause the differing outputs from H: differing x1/x2 or differing y1/y2 values cause the differing z1 and z2 values. Furthermore, Z3's counterexample trace provides the values assigned to x1, x2, y1, and y2 for a particular counterexample, allowing us to pinpoint the offending inputs (y1/y2 in this case). Thus, we reduce the question of why z1 and z2 differ to the question of why y1 and y2 differ.

This insight leads to a straightforward algorithm for programs with only uninterpreted functions: if an uninterpreted function output differs, use the counterexample trace to select the differing inputs, and recurse on the uninterpreted function that generates those differing inputs. Of course, real programs have a mixture of interpreted and uninterpreted functions. For example, various interpreted functions (if/then/else, ">", "<=", "+") produce y1 and y2:

```
y1 = (i > 10) ?  G1(F(i)) :  F(i) + 1
y2 = (i <= 10) ?  1 + F(i) :  G2(F(i))
```

Our solution is crude but effective in practice: we collapse

multiple interpreted functions into single uninterpreted functions, ignoring the interpreted functions semantics:

```
y1 = U1({i, 10, G1(F(i)), F(i), 1})
y2 = U2({i, 10, 1, F(i), G2(F(i))})
```

Since the inputs to the interpreted functions may appear in different orders (e.g. for commutative operators like "+"), we group all inputs together in unordered sets. The question of why y1 and y2 differ then reduces to the question of why the two sets above differ. To answer this, our algorithm uses heuristics to match the members of the set as well as possible, taking into account values from the trace and the uninterpreted function names and input names that appear in the dataflow subgraphs for each member. For example, the two F(i) members are matched based on both their trace value and their common names F and i. After matching i, 10, 1, and F(i), the members G1(F(i)) and G2(F(i)) remain, and thus are considered the cause of the differing y1 and y2 values. The algorithm then recurses on G1(F(i)) and G2(F(i)), leading to the root cause that G1 differs from G2. In cases where all members of the sets match, the analysis blames the interpreted functions that were erased when creating the sets (although the analysis is unable to pinpoint exactly which interpreted function is to blame).

Our algorithm employs additional heuristics, such as using Mem/Heap values in the trace to skip over some of the backwards search, replacing stack memory accesses with variable accesses, and taking into account the semantics of some interpreted functions (e.g. the x86 exclusive-or idiom for setting a register to 0); we omit details of these for brevity.

3.1 Bucketing

Once a root cause is identified, the algorithm emits a short summary of the root cause (e.g. G1/G2 are mismatched function names) and a list of matched uninterpreted functions traversed by the algorithm on the way to the root cause (just H in the example above). A user of the root cause algorithm can then write simple classifiers to group related root causes together into buckets, eliminating the need to manually review all the counterexamples in each bucket.

For example, mismatched runtime system calls appear as mismatched uninterpreted function names; we created a classifier for each such mismatch so that different run-time system function mismatches are grouped together into different buckets. As another example, we created a classifier to detect when the algorithm traverses a load uninterpreted function whose inputs contain mismatched integer constants or mismatched symbolic constants, indicating mismatched field offsets. Each classifier relies only on the short summary generated by the root cause analysis, so new classifiers do not require modifying the root cause analysis.

4. EXPERIMENTAL RESULTS

This section presents results from running the validator on the output of the CLR (Common Language Runtime) compiler, including measurements of month-to-month version differences, optimized vs. unoptimized assembly language differences, differences across architectures (x86 and ARM), and different compilation scenarios (JIT vs. ahead-of-time MDIL). This section also presents results from running the validator on some of the same tests, but with faults artificially injected to check for false negatives (missed bugs).

The CLR just-in-time compiler compiles managed-language bytecode into assembly language for execution in the CLR virtual machine. (Other, higher-level compilers generate bytecode from various managed languages, such as C#, F#, and managed C++.) All assembly language in the measurements was generated by the CoreCLR [10] subset of the CLR, which is used to run mobile code (Silverlight applications [10]) and is hence a particularly security-critical version of the CLR, worth extensive testing. This subset uses the same just-in-time compiler as the full desktop CLR, but has a smaller set of libraries and an easier installation process, allowing us to easily build and run many versions of the compiler on the same machine.

Although the CLR compiler runs in just-in-time mode by default, it also supports ahead-of-time compilation (the native image generation, or "NGEN" feature), which we used to generate the assembly language files for the validator. One aspect of the CLR's JIT-oriented design made cross-version validation slightly difficult, though: the generated code contains many embedded addresses and field offsets, rather than symbolic names. These addresses and offsets vary from version to version, and even from compilation to compilation, leading to many false alarms that merely report mismatched addresses (e.g "0x004fe208 != 0x003dd484"). Therefore, we modified the CLR to print symbolic information for addresses and fields. These modifications consisted of about 230 lines of code, scattered across 20 files. We also extended the CLR to print more information about methods and their argument types so that the validator could determine register and stack usage for each call; this modification consisted of 370 lines of code added to one file. These modifications were shared across all the CLR versions in our experiments.

As input to the CLR compiler, we used 16 test programs from two sources: the publically available Silverlight libraries (10 bytecode files), and the 6 largest bytecode files in the Bartok compiler [3, 5] test suite.

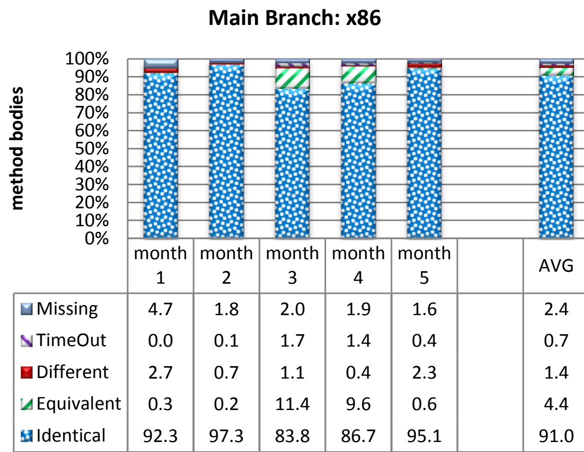
4.1 Month-to-month comparisons

Comparing just two versions of the compiler would provide only a limited glimpse of how version-to-version comparison works in practice across long periods of time, so we used a diverse set of CLR versions in several dimensions:

- First, we selected versions from the CLR source control server across seven months. Specifically, we chose eight different dates, each spaced exactly one month apart; we refer to those dates as "date 0"..."date 7", and the months that separate them as "month 1"..."month 7".
- Second, we selected versions from two different source control branches. CLR compiler development takes place across many source control branches, including one "main" branch and many "feature" branches where independent features are developed. We chose the two branches with the most check-ins to the compiler source files over the seven-month time period, the main branch and one feature branch.
- Third, we configured the compilers in x86-generation mode and ARM-generation mode.

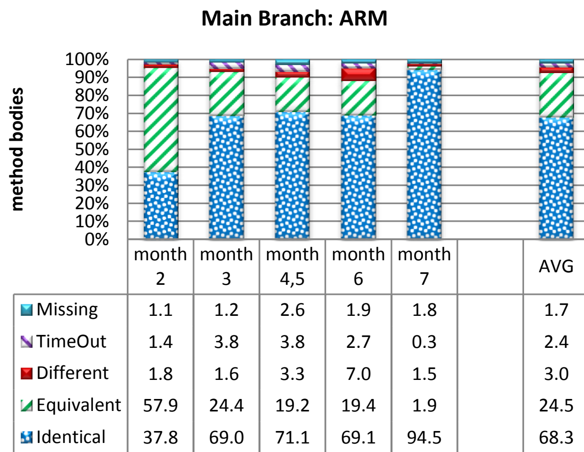
In each configuration, we left all optimizations enabled except for inlining (as explained in section 2).

Since these are internal versions under development rather than released compilers, not all versions could compile all the test files. Some versions did not work at all in the CoreCLR NGEN configuration; we omitted these from the re-



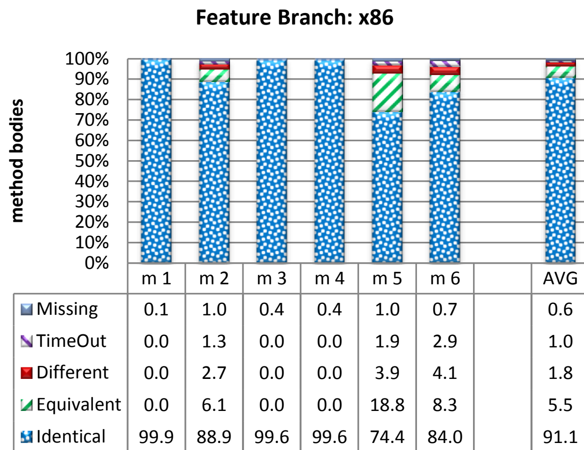
	month 1	month 2	month 3	month 4	month 5		AVG
Missing	4.7	1.8	2.0	1.9	1.6		2.4
TimeOut	0.0	0.1	1.7	1.4	0.4		0.7
Different	2.7	0.7	1.1	0.4	2.3		1.4
Equivalent	0.3	0.2	11.4	9.6	0.6		4.4
Identical	92.3	97.3	83.8	86.7	95.1		91.0

Figure 1: Month-to-month comparisons for feature branch, x86 code



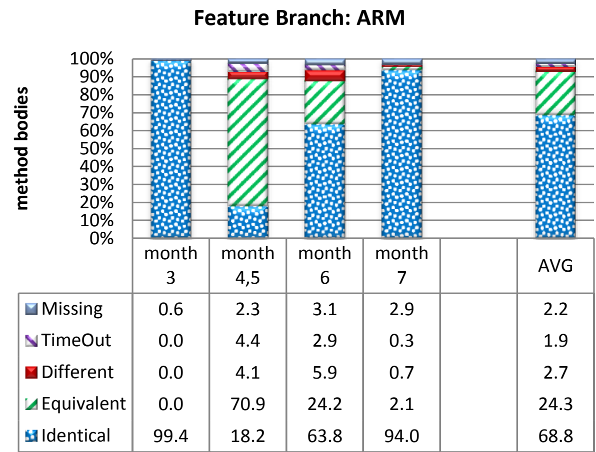
	month 2	month 3	month 4,5	month 6	month 7		AVG
Missing	1.1	1.2	2.6	1.9	1.8		1.7
TimeOut	1.4	3.8	3.8	2.7	0.3		2.4
Different	1.8	1.6	3.3	7.0	1.5		3.0
Equivalent	57.9	24.4	19.2	19.4	1.9		24.5
Identical	37.8	69.0	71.1	69.1	94.5		68.3

Figure 2: Month-to-month comparisons for feature branch, ARM code



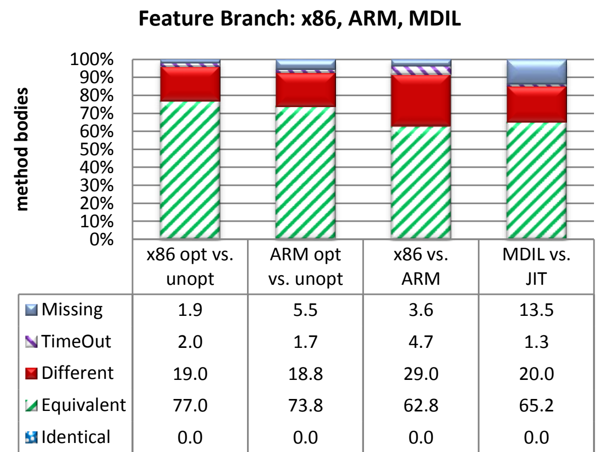
	m 1	m 2	m 3	m 4	m 5	m 6		AVG
Missing	0.1	1.0	0.4	0.4	1.0	0.7		0.6
TimeOut	0.0	1.3	0.0	0.0	1.9	2.9		1.0
Different	0.0	2.7	0.0	0.0	3.9	4.1		1.8
Equivalent	0.0	6.1	0.0	0.0	18.8	8.3		5.5
Identical	99.9	88.9	99.6	99.6	74.4	84.0		91.1

Figure 3: Month-to-month comparisons for main branch, x86 code



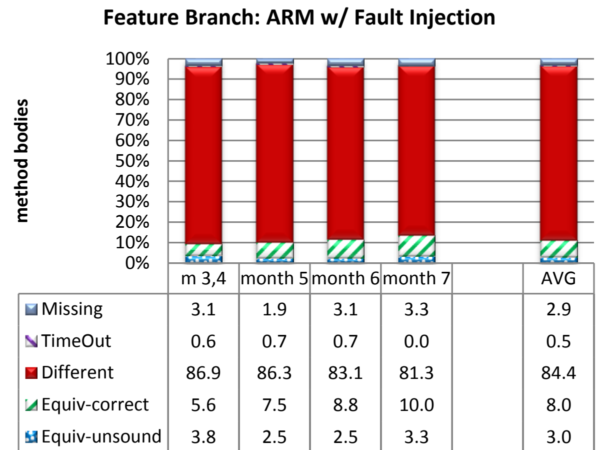
	month 3	month 4,5	month 6	month 7		AVG
Missing	0.6	2.3	3.1	2.9		2.2
TimeOut	0.0	4.4	2.9	0.3		1.9
Different	0.0	4.1	5.9	0.7		2.7
Equivalent	0.0	70.9	24.2	2.1		24.3
Identical	99.4	18.2	63.8	94.0		68.8

Figure 4: Month-to-month comparisons for main branch, ARM code



	x86 opt vs. unopt	ARM opt vs. unopt	x86 vs. ARM	MDIL vs. JIT
Missing	1.9	5.5	3.6	13.5
TimeOut	2.0	1.7	4.7	1.3
Different	19.0	18.8	29.0	20.0
Equivalent	77.0	73.8	62.8	65.2
Identical	0.0	0.0	0.0	0.0

Figure 5: Optimized vs. unoptimized comparisons and x86 vs. ARM comparisons



	m 3,4	month 5	month 6	month 7		AVG
Missing	3.1	1.9	3.1	3.3		2.9
TimeOut	0.6	0.7	0.7	0.0		0.5
Different	86.9	86.3	83.1	81.3		84.4
Equiv-correct	5.6	7.5	8.8	10.0		8.0
Equiv-unsound	3.8	2.5	2.5	3.3		3.0

Figure 6: Month-to-month comparisons for feature branch, ARM code with injected faults

sults. Other versions compiled some test programs but not others: one date in each branch compiled just over half the of 16 test programs to x86, and another date in each branch compiled only 15 of 16 test programs to ARM. For these, we kept the successful test programs and omitted the failing test programs from the results. Other versions compiled most but not all methods in each test program; we kept these failed methods in the results, but classified the methods as "Missing" in the measurements. In addition, the validator itself sometimes failed to process a method, usually due to missing address or call information from the CLR; these methods are also classified as "Missing".

We randomly sampled 100 method bodies from each of the 16 test programs, for each of the 7 months, for each of the 2 source control branches, and for each of the 2 architectures (with optimizations enabled), for a total of 29300 method-body-to-method-body comparisons (less than 100*16*7*2*2 = 44800 due to the omissions described above).

Figures 1, 2, 3, and 4 show the validation output for these 29300 method-body pairs. The "Different" category shows the percentage of method bodies categorized by the validator as semantically different. The measured difference rate for x86 code is 1.4% (main branch) and 1.8% (feature branch), while the rate for ARM code is 3.0% (main branch) and 2.7% (feature branch), for an overall average of 2.2%. The higher rate for ARM may reflect the youth of the ARM code generator relative to the x86 code generator. Nevertheless, even if the ARM code generator experiences more churn than the x86 code generator, the x86 validation is still important to ensure that changes to the compiler for ARM compilation do not introduce bugs into the x86 compilation.

The "Equivalent" and "Identical" categories show the method-body pairs judged equivalent by the validator. The "Equivalent" method-body pairs are methods judged equivalent by SymDiff. The "Identical" method-body pairs are method bodies with exactly the same instructions (i.e. they are syntactically identical) after replacing integer addresses and field offsets with their corresponding symbols. For these pairs, SymDiff would always consider the method bodies equivalent, so there's no reason to run SymDiff.

For the x86 code, about 90% of methods are syntactically identical. This is an important reason for the low false alarm rate of version-to-version testing: even a simple syntactic diff tool could achieve a false alarm rate of less than 10% for the x86 tests. Nevertheless, reducing the false alarm rate to less than 2% requires semantic checking. Furthermore, fewer than 70% of ARM methods are syntactically identical; for these, a syntactic diff tool would have too many false alarms, and semantic checking is essential.

Some method-body pairs are too large for Boogie/Z3 to handle in a reasonable amount of time. We set a limit of 200 seconds on the time Z3 spends on any one method-body pair. We also set a limit of 1000 assembly instructions on any method we pass through Boogie (due to a Boogie stack overflow issue), although this instruction limit affected less than 0.3% of method-body pairs. The "TimeOut" category shows the method-body pairs that exceeded either of these limits; the time-out rate ranged from 1-2%.

We randomly sampled 100 pairs reported as "Different" from across Figures 1, 2, 3, and 4, and then used the root cause analysis to examine the causes of the differences. The most common differences were embedded addresses (31%; some addresses slipped through in spite of our efforts to convert integer addresses to symbolic names inside the CLR), lack of aliasing information (22%), interprocedural optimizations (9%), and bit-level arithmetic (6%).

Note that the false alarm rate depends on how often the validator is run: we chose month-to-month comparisons, but developers might run week-to-week validations or validations after each check-in. In this case, we'd expect a lower false alarm rate, as fewer changes occur in a week than in a month.

4.2 Optimization level, architecture, and compilation scenario comparisons

Figure 5 shows measurements for varying optimization levels, architectures, and compilation scenarios. Each of these measurements uses the same version of the CLR compiler in two different configurations. The two left measurements, taken for the feature branch at date 6 (the most recent date common to both x86 and ARM), compare the CLR output with default optimizations (except inlining) to the CLR output with minimum optimizations. This is effectively a validation of the optimization phases, and the results show about a 19% false alarm rate, similar to other measurements of false alarms for compiler optimizations [16, 14], and much higher than our 2.2% month-to-month validation rate. This suggests that compiler-generated code is more similar across time than across optimization levels.

Figure 5 also shows validation of x86 code against ARM code, both with optimizations enabled (except for inlining). The false alarm rate of 29% makes this validation too unwieldy to perform very often, but it might still be useful for bootstrapping the validation of code generation for a new architecture relative to an existing, well-tested code generator for another architecture. After this bootstrapping, month-to-month validation over the new architecture can be used.

The fourth measurement in Figure 5 shows validation of ARM code generated directly by the CLR compiler in JIT mode vs. ARM code generated via machine-dependent intermediate language (MDIL [12]) for a set of ten popular phone apps. The MDIL mode is used by Windows Phone 8 to shift compilation work from the phone to a dedicated server: the server's CLR compiler compiles phone apps to MDIL, and the phone compiles the MDIL to ARM code. The MDIL code contains ARM instructions, but uses symbolic placeholders for field accesses, method calls, and run-time system calls. This leads to some differences in the ARM code for method calls and run-time system calls, which account for most of the false alarms in Figure 5.

4.3 Fault injection

The month-to-month comparisons demonstrate a low false alarm rate, but this by itself is not enough – a tool that always said "Equivalent" would have a low false-alarm rate, but would not be useful. We also need to know that when two pieces of assembly language are code semantically different, the tool is likely to report them as different. In other words, we want few "false negatives" (semantically different method bodies reported as "Equivalent").

As an example, we ran the tool on three known incorrect assembly language files generated by past CLR versions, and the tool reported all three as different. While this was reassuring, we also wanted to quantify the false negative rate on a broader set of bugs. Therefore, we performed the following fault-injection experiment. We re-ran the ARM month-to-month tests for the Feature branch, but for each compari-

son of method body Mk to method body Mk', we injected a random fault into Mk' before performing the comparison. Each fault was one of the following single-instruction modifications: changed arithmetic opcode, changed register, changed integer constant (randomly flipping one of lower 4 bits), changed symbolic offset/address, changed branch condition, changed branch target.

We then hand-inspected each method body pair reported as "Equivalent" to see whether it was really equivalent. To keep the number of hand-inspections reasonable, we ran the validator on 10 random method bodies per test program rather than 100. This resulted in 630 method body comparisons, of which 69 were reported as "Equivalent". Of these, hand-inspection showed 50 to actually be equivalent (marked as "Equiv-correct" in Figure 6). Nearly all of these involved modifications to registers whose values were no longer needed; in particular many small methods created a frame pointer but never used the frame pointer; faults that corrupted the frame pointer did not change the meaning of the method body. Another interesting fault was changing a conditional jump if a register equaled 0 to a conditional jump if the register was less than or equal to zero using unsigned comparison; these conditions are equivalent.

The remaining 19 (3%) were genuine false negatives (marked as "Equiv-unsound" in Figure 6):

- 7 were modifications to load instructions whose results were ignored. The compiler emitted these instructions not for their results, but for the exception they cause in case of a null pointer access; our encoding of loads does not capture this side effect.
- 7 were due to unsound loop handling.
- 3 were faults in exception handling code considered unreachable in our encoding.
- 1 was due to a 4-byte store to address [sp+2], which overlaps address [sp+4]. Because our memory model does not model word memory operations byte-accurately, the validator does not detect that the store to [sp+2] corrupts the contents of [sp+4].
- 1 was due to the validator omitting the implicit type dictionary argument to a generic method call.

4.4 Performance

Figure 7 shows the time taken to validate each method for the months 4-5 from Figure 4 (feature branch, ARM code). We chose this particular comparison as a worst-case-measurement: it contains the minimum number of syntactically identical method bodies from all the month-to-month comparisons (just 18%), and syntactically identical methods take almost no time to validate.

Figure 7 shows the total bytecode size of each test file along with the number of assembly language instructions (taken from date 5, the most recent of the two compiler outputs in the month 4-5 comparison) and the number of method bodies generated by the CLR compiler for the entire test file. For the validator performance, the average instructions per method is correlated with the average time taken to validate each method. In particular, the go test has far larger method bodies (454 instructions per method body) than the other tests, takes longer to validate (53 seconds on average), and contains the majority of theorem prover time-outs (39%) from among the tests. On average, each method-body pair takes 13.5 seconds to validate on a single core of a 2.4GHz Intel Core2 Q6600 processor with 6GB of memory. While not as fast as more specialized approaches to translation validation [16], this is fast enough to process tens of thousands of methods with few time-outs.

	bytecode size (bytes)	assembly language instructions	compiled method bodies	instructions per method body	time per method body (seconds)	time-outs (percent)
ByteMark	77824	17919	216	83	21.1	4
Crafty	184320	56366	328	172	17.4	7
go	479232	202151	445	454	53.1	39
ijpeg	110592	20521	265	77	19.5	7
sat_solver	106496	21423	251	85	5.0	1
xlisp	98304	19395	546	36	9.6	0
Microsoft.VisualBasic	253952	78776	1062	74	16.8	5
System.Core	536576	226902	4978	46	7.4	1
System	233472	51067	1137	45	12.1	2
System.Net	225280	40773	1106	37	10.2	1
System.Runtime.Serialization	413696	106064	3036	35	7.0	0
System.ServiceModel	520192	121407	3355	36	3.0	0
System.ServiceModel.Web	73728	11020	154	72	21.5	4
System.Windows.Browser	143360	35179	706	50	5.5	0
System.Windows	1478656	351060	10401	34	3.1	0
System.Xml	319488	69879	1775	39	4.3	0
AVERAGE	328448	89369	1860	85.9	13.5	4.4

Figure 7: per-test results; ARM feature branch months 4,5

4.5 Root cause analysis and bucketing

We evaluated the effectiveness of root cause analysis and bucketing on three sets of results: the monthly results and the two results from Figure 5 with the most false alarms (x86 vs. ARM and MDIL vs. JIT). For each, we ran Section 3's dataflow-graph-based root cause analysis on a random sample, developed appropriate buckets for the output of the root cause analysis, and manually checked the coverage and accuracy of the automated bucketing. The MDIL vs. JIT results were best: of 600 methods classified as "different", 578 (96%) were grouped into 21 buckets, leaving only 4% of the methods unbucketed. By contrast, the monthly results (60 out of 111 grouped into 7 buckets) and x86 vs. ARM (232 out of 436 grouped into 14 buckets) only bucketed 53-54% of the differences. In all cases, no differences were ever placed incorrectly in a bucket (i.e. failures in the root cause analysis always caused an unbucketed result rather than a misbucketed result). The difference in success rates was due to the nature of the root causes; the MDIL vs. JIT causes were dominated by differences at function call sites, which were relatively easy to diagnose and bucket, while the other two sets contained more differences further away from call sites. Even when the reported root cause wasn't precise enough for bucketing, the root cause analysis correctly highlighted most of the relevant instructions in the majority of the cases, still greatly reducing the human effort required for diagnosis.

For comparison, we evaluated the effectiveness of MAX-SAT root cause analysis [6] on these tests, using Z3's support for computing maximum satisfiability. The MAX-SAT analysis uses inputs from a particular counterexample, transforms the programs into a system of constraints (e.g. x1 = F(i), z1 = H(x1, y1), etc. for the example from Section 3), and tries to satisfy as many of the constraints as possible while allowing the program to succeed on the inputs from

the counterexample. Unsatisfied constraints correspond to possible root causes. The resulting algorithm has the advantage over the dataflow-graph-based analysis that it retains the semantics of interpreted functions during its search.

In contrast to the dataflow graph analysis, which tries to pinpoint a single cause, MAX-SAT tends to return a larger set of possible causes. This is useful for assisting manual analysis, but doesn't lend itself directly to bucketing. For example, naive application of MAX-SAT highlights `x1 := F(i)` as well as `y1 := G1(x1)` in the example from Section 3, on the grounds that an unconstrained x1 can contain an arbitrary value that causes G1(x1) to equal G2(x2), for some possible G1 and G2. To produce more bucketable results, we applied MAX-SAT in a more constrained way, disallowing equality between values unless both values were produced by the same uninterpreted function. We also forced MAX-SAT to report pairs of statements (one in each program) rather than individual statements. We then manually compared results from 129 examples randomly sampled from the x86 vs. ARM and MDIL vs. JIT tests. To prevent MAX-SAT from always timing out, we also limited the analysis to focus only on causes at function call sites. Overall, the resulting algorithm bucketed fewer causes than the dataflow-graph approach, and never bucketed a cause that the dataflow-graph approach failed to bucket. However, on some unbucketed examples, MAX-SAT came closer to the real cause, suggesting that combination the MAX-SAT and dataflow-graph approaches might lead to better bucketing than either alone.

4.6 Bugs found

The CLR test team ran the validator on over 500,000 methods from various test programs, revealing 12 bugs in the CLR compiler that were previously unknown (i.e. had not been discovered even during extensive testing). Of these, 6 were correctness bugs, including: multiple cases of incorrect runtime system functions being called; incorrect calling conventions for return buffers in some corner cases; compatibility issues with floating point rounding due to conversions between 8-byte numbers and 10-byte numbers; incorrect treatment of static fields and exceptions.

The other 6 bugs were performance bugs (unintended performance regressions). Although we hadn't intended to look for performance issues, some differences that were false alarms for correctness were nevertheless real bugs for performance. These included calls to slow versions of runtime system functions, runtime system calls that were supposed to be inlined, and unintentionally disabled common subexpression eliminations. This suggests that false-alarm differences can yield useful insights, so bucketing false alarms may be better than eliminating all false alarms.

5. RELATED WORK

Although research on translation validation goes back over a decade [11, 9], few validators checked industrial-scale compilers: Necula's validator [9] targeted gcc, Peggy [15] targeted LLVM, and Tristan et. al. [16] targeted LLVM. These focused on validating optimizations in the compiler's intermediate representation rather than end-to-end validation from source language to assembly language. Of these, only Necula's work reports false alarm rates of less than 10%, and this seems to be due to reporting rates for just a handful of individual optimizations, which have false alarm rates as high as 3.5%, rather than a whole optimization pipeline, for which it's unclear how well Necula's approach would work. Peggy reports false alarm rates of 10%-25% for SPEC 2006 benchmarks under LLVM, while Tristan et. al. report false alarm rates of 10%-40%. Ramos and Engler [13] report a semantic difference rate of 11% for optimized-to-unoptimized code comparisions, although the semantic equivalence rate was only 54% due to timeouts and tool issues.

When applied to the Soot Java optimizer rather than to LLVM, Peggy produces false alarm rates of just 2% [15], indicating some promise for translation validation's practicality; perhaps given more experience, a 2% false alarm rate is also within reach for compilers like gcc, LLVM, and the CLR. On the other hand, compared to relatively clean research projects like Soot, the many complexities of industrial-scale compilers (tricky optimizations, complex language features, intricate run-time system interactions, unexpected special cases) may simply overwhelm translation validators. If so, version-to-version comparison may have an easier job, since most compiler complexities become inextricably engrained in the compiler source code, and are thus stable over time.

When discussing Peggy, the authors mention the desire for a finer-grained heap model [15] and SMT theorem proving technology [14]. Having implemented finer-grained heap models for cross-version validation and having used the Z3 SMT solver for cross-version validation, we can report that these techniques work, and enable the handling of complex instructions like "rep stosb", but that it can take 10 seconds or more per method, acceptable for offline testing via random sampling but not practical as an online compiler pass. By contrast, Tristan et. al. [16] move in the opposite direction, towards more specialized, better-performing representations with less proof search. It would be interesting to see the false alarm rate for these specialized representations for cross-version validation. An open question is whether such specialized representations are as easy for humans to diagnose as counterexample traces generated from Z3's models.

Yang et. al. [17] found a large number of compiler bugs by comparing the output of different compilers to each other, including gcc and LLVM. Rather than using semantic validation techniques, they simply executed the assembly language generated by the different compilers and looked at the output. While execution on concrete inputs may have limited coverage compared to semantic equivalence checking, Yang et. al. compensated by randomly generating a large set of test programs to cover a large fraction of the compilers' behaviors. Current translation validators are limited to the compiler behaviors exercised by their test suites; on the other hand, translation validation and cross-version validation gives greater assurance about the behavior of the chosen test programs (which are often widely deployed programs, like Silverlight), since they consider all possible inputs to the test programs; some of the bugs that our tool found would have been very unlikely to occur on random inputs.

6. CONCLUSIONS

Our results show that month-to-month cross-version validation achieves a low false alarm rate in practice (2.2% using our validator on the CLR) when compared to recent translation validation results [16, 14] (10%-40% false alarm rates) and to the results of running our validator on unoptimized vs. optimized code (19% false alarm rate). This indicates that cross-version validation can serve as a useful supplement to translation validation and traditional testing.

7. REFERENCES

[1] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO)*, volume 4111, 2006.

[2] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. In *Communications of the ACM*, Feb. 2010.

[3] J. Chen, C. Hawblitzel, F. Perry, M. Emmi, J. Condit, D. Coetzee, and P. Pratikakis. Type-preserving compilation for large-scale optimizing object-oriented compilers. *SIGPLAN Not.*, 43(6):183–192, 2008.

[4] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[5] C. Hawblitzel and E. Petrank. Automated verification of practical garbage collectors. In *POPL*, pages 441–453, 2009.

[6] M. Jose and R. Majumdar. Cause clue clauses: Error localization using maximum satisfiability. In *PLDI*, 2011.

[7] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Computer Aided Verification (CAV Š12)*, 2012.

[8] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Principles of Programming Languages (POPL '06)*, pages 42–54, 2006.

[9] G. C. Necula. Translation validation for an optimizing compiler. In *Programming Language Design and Implementation (PLDI '00)*, pages 83–94, 2000.

[10] A. Pardoe. Clr inside out: Program silverlight with the coreclr. In *MSDN Magazine*, Aug. 2008.

[11] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS '98)*, pages 151–166, 1998.

[12] S. Ramaswamy. Deep dive into the kernel of .NET on Windows Phone 8. In *Build Conference*, Nov. 2012.

[13] D. Ramos and D. Engler. Practical, low-effort equivalence verification of real code. In *CAV*, 2011.

[14] M. Stepp, R. Tate, and S. Lerner. Equality-based translation validator for llvm. In *CAV*, 2011.

[15] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *Principles of Programming Languages (POPL '09)*, pages 264–276, 2009.

[16] J. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for llvm. In *Programming Language Design and Implementation (PLDI '11)*, pages 295–305, 2011.

[17] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *PLDI*. ACM Press, 2011.

[18] J. Zhao, S. Zdancewic, S. Nagarakatte, and M. M. K. Martin. Formalizing the llvm intermediate representation for verified program transformation. In *POPL*, 2012.