Using Csmith to Find Compiler Bugs

Kai Tamkun

December 9, 2021

Contents

1 Introduction

For my final project, I used Csmith to find bugs in a compiler I wrote. Csmith is a fuzzing tool that generates C programs and has been used to find hundreds of bugs in popular compilers, including GCC, Clang and CompCert (Yang et al., 2011). Programs generated by Csmith convolutedly calculate a checksum and print it. It's most useful for comparing multiple compilers; if two different compilers' outputs for the same Csmith-generated program differ, then there must be a bug in one of the compilers. In this project, I used Clang as a reference and assumed its output to be correct.

2 C-Reduce

C-Reduce is a program that reduces the size of C files while still ensuring they exhibit some desired behavior (Regehr et al., 2012). It makes changes to a C file and runs a user-provided script to determine whether to accept or reject the changes. It does this indefinitely, but at some point the rate of change becomes so slow that the program is practically unable to reduce the C file any further. The script I wrote checked whether the reduced program printed the same checksum as the original program.

3 ll2w

My compiler is called ll2w. It translates LLVM intermediate representation into bytecode for Why, my custom ISA that I created in 2017 after I learned MIPS assembly. Why was originally created for use in webpages, and until recently the assembler, linker and VM were all written in JavaScript. More recently, I ported all the code to C++. My goal with Why is to make it support enough LLVM IR to make it possible to write an operating system for Why, and thanks to Csmith, I've gotten much closer to that goal.

3.1 Compilation Process

ll2w starts by reading in an LLVM IR source file and parsing it. Currently only the text format is supported, but support for the binary IR format may be added later. Once it's produced an AST, it extracts struct definitions, function declarations and definitions, global variables and debug data. (Debug data correlates LLVM IR instructions to instructions in the original source code compiled by an LLVM frontend.) It then goes over each function definition and gradually converts each LLVM IR instruction into some number of Why instructions. This number can be zero if the original instruction is removed, which happens for certain intrinsics, greater than one for more complex instructions or in most cases just one. There are three phases

of function compilation: initial compilation, register allocation and final compilation. Initial compilation contains passes that don't require registers to have been allocated and includes passes like `IgnoreIntrinsics`, `SplitBlocks`, `CopyArguments`, `MovePhi` and most instruction lowering. The register allocation phase doesn't contain multiple passes, but instead keeps attempting register allocation via graph coloring, spilling when necessary, until a coloring can be successfully found. Final compilation finishes up the rest of what's necessary and includes function prologue and epilogue insertion. After every function has been compiled, ll2w stringifies all the global variables, debug data and functions and emits Why assembly.

3.2 Structs

Many of the bugs I encountered were in how ll2w handles structs. Most structs are padded, meaning that elements of the struct don't always occur immediately one after the other but sometimes include empty space between them to maintain alignment rules. Because LLVM makes assumptions about what these offsets will be when casting types, it's necessary for ll2w to implement struct padding properly. Struct padding is target-dependent, but because LLVM doesn't support Why as a target (ll2w is a separate thing entirely that doesn't make use of any LLVM code or libraries), I use MIPS64 as it's the closest supported architecture.

4 1.c

In the beginning, I generated a C file with Csmith and named it 1.c. This is the only file I ran C-Reduce on; in later files, I decided to forego C-Reduce in case the reductions removed any code that my compiler would compile incorrectly. To help track down where the bugs occurred, I inserted printf statements at many points in the program that would print the line number of the printf statement and the current value of the checksum. Comparing the output produced by the Clang-compiled x86_64 binary and the ll2w-compiled Why bytecode told me the range of lines in which the next error occurred.

4.1 Overflow

Certain Why instructions (collectively called I-types) can take an immediate value as part of the instruction. These immediate values are 32 bits wide and are usually sign extended by the VM when used, so their value ranges from -2^{31} to $2^{31} - 1$. However, LLVM IR places no restrictions on the immediate values it uses. My compiler would detect cases when the LLVM IR immediate value was out of the valid range for Why immediate values, but it used to print a warning and then truncate the value to 32 bits. This was the first bug I noticed in 1.c. The fix was to check for overflowing values in the code that translates mathematical

instructions and use two instructions to load the upper and lower halves of the immediate value into a register and use that register as an operand in the Why instruction instead of an immediate value.

4.2 Packed Structs

The next bug I found involved structs. Field `f4` of the struct in Figure 1 was being read incorrectly. ll2w was using the assumption that all structs were packed (i.e., not padded), but this was incorrect. In some places, the LLVM IR for 1.c was bitcasting arrays of 32 8-bit integers into an entire `S1` struct. LLVM translated the `S1` struct from C into a `{i32, i32, i24, i24, i32, i24, i24, i32}` struct in LLVM IR, where `i32` represents a 32-bit integer and `i24` represents a 24-bit integer. The sum of these widths, assuming no padding, is 28 bytes, but LLVM was treating the struct as if it were 32 bytes in size. At this point in time, ll2w had code for struct padding, but it was disabled. The code would insert padding until every element of the struct was aligned to the size of the largest element of the struct. In this case, that would entail inserting a padding byte after each `i24` element, which would increase the width to 32 bytes as expected. Although this caused the correct behavior in this case, it would later turn out that the struct padding was incorrect and worked correctly in this case only coincidentally.

```
struct S1 {
   const signed      f0 : 30;
   signed            f1 : 27;
   volatile unsigned f2 : 19;
   volatile signed   f3 : 21;
   volatile unsigned f4 : 29;
   signed            f5 : 23;
   unsigned          f6 : 18;
   volatile unsigned f7 : 31;
};
```

Figure 1: The C struct that was being incorrectly accessed.

4.3 Sign Extension

LLVM's mathematical instructions like `add`, `sub` and `ashr` (arithmetic shift right) have the same name regardless of the sizes of their operands. ll2w was ignoring the sizes of the operands, treating everything as if it were 64 bits in width, because registers in Why are 64 bits wide. This is incorrect in many cases, however. One example is arithmetic right shift. It's supposed to preserve the signedness of the shifted operand, but the Why instruction does that only for 64-bit values. As a result, it wasn't preserving signedness of `i32` operands and other small operands. The fix was to sign extend values shorter than 64 bits before mathematical operations.

```
unsigned b; // number of bits to sign extend from
const int m = 1u << (b - 1);
input = input & ((1u << b) - 1);
int result = (x ^ m) - m;
```

Figure 2: The code I initially used for sign extension before translation into assembly (Anderson, 2005).

Figure 2 shows the C equivalent of the assembly I had ll2w insert for sign extension from arbitrary widths. However, it wouldn't always work. In one case, the number it ended up computing for a 32-bit integer was `11111111111111111111111111111101111111111111111100101101110010` when the result should have been `11111111111111111111111111111111111111111111111100101101110010`. Instead of trying to fix my assembly translation of Figure 2, I decided to add dedicated instructions to Why for sign extension from 8-bit, 16-bit and 32-bit values. This allows the sign extension to be computed natively by the VM, instead of emulated in software. Using these instructions fixed the bug.

4.4 String Parsing

LLVM supports strings as global variables. One of the responsibilities of my LLVM IR parser is to deal with escape sequences in these strings. However, there was a bug in which some characters after hexadecimal escapes were being skipped. This caused strings like `"\06\FFw"` to be translated as `%string "\x06\FF"` (note the missing "w").

```
@.str = private unnamed_addr constant [7 x i8] c"g_8.f0\00", align 1
```

Figure 3: An example of an LLVM IR global variable containing the string "g_8.f0".

4.5 Struct Padding

4.5.1 In the Code Section

As mentioned earlier, my struct padding code was incorrect. I wrote a program that hooks into LLVM and uses its offset calculation code to print the offsets for a given struct. I used its output to reverse engineer how struct padding should work. For the struct `{i32, i8, [3 x i8], i8, i8, i8, i8, i64, i16, i32}`, the offsets it printed were

0, 4, 5, 8, 9, 10, 11, 16, 24, 28.

I had my compiler print what it thought the offsets were and it printed

0, 4, 8, 12, 13, 14, 15, 16, 24, 28. Half the offsets were incorrect. I implemented a fix that as far as I can tell has once and for all fixed the issue with struct member offset calculation.

4.5.2 In the Data Section

However, I wasn't using the offset calculation code in all the places I needed to. When outputting global struct instances in the data section, padding wasn't being taken into account. The x86_64 assembly in Figure 4 contains 4 bytes of zeros before the 8-byte 6949690704508322118 value, while the Why assembly in Figure 5 does not. After I implemented a fix, the Why code started printing the same checksums as the Clang-compiled x86_64 executable.

```
g_884:
    .long   2629993089
    .byte   222
    .zero   3
    .byte   187
    .byte   90
    .byte   255
    .byte   127
    .zero   4
    .quad   6949690704508322118
    .short  0
    .zero   2
    .long   1
```

Figure 4: The x86_64 assembly produced by Clang for global variable `g_884`.

```
@g_884
%4b -1664974207
%1b -34
%fill 3 0
%1b -69
%1b 90
%1b -1
%1b 127
%8b 6949690704508322118
%2b 0
%4b 1
```

Figure 5: The Why assembly produced by ll2w for global variable `g_884`.

5 2.c

2.c was the next program I generated with Csmith. I didn't use C-Reduce on it because I wanted to stress test my compiler.

5.1 Data Section References

Global variables in C can be pointers to other global variables. In x86_64 assembly, this might look like `g_684: .quad g_685`. In Why, the equivalent is `@g_684 %8b g_685`. The first bug revealed in 2.c turned out to be not in my compiler, but in my assembler. It was evaluating expressions and applying relocation in the wrong order, causing it to compute incorrect addresses. I fixed the bug by swapping the order.

5.2 Underflow

In the next bug, the for loop in Figure 6 wasn't terminating. The loop iterator variable was a 32-bit value stored in memory. Because loading a 32-bit value in Why clears the upper 32 bits of the destination register, when the integer was decremented at zero, it didn't change to negative one but instead to $2^{32} - 1$, or about positive 4.3 billion. The fix was to sign extend values shorter than 64 bits in width before doing signed comparisons with them.

```
for (g_2[1] = 28; (g_2[1] > (-25)); g_2[1]--) {
    ...
}
```

Figure 6: The for loop that wasn't terminating.

5.3 Getelementptr

The final bug in 2.c was with my translation of getelementptr instructions. LLVM's getelementptr instruction computes offsets into structs and does pointer arithmetic. Sometimes, this computation has to be done at runtime, such as when one of the indices is a local variable instead of a constant, but usually all the indices are constant and the instruction can be replaced with a single addition at compile time. The part of my compiler that was doing the getelementptr computations was returning a value in bits, but the code elsewhere in the compiler that called it was treating it as a value in bytes, causing invalid offsets to be produced. The fix was to simply divide the computed value by 8. Once this bug was fixed, the Why version of 2.c produced the correct checksum.

6 Conclusion

Csmith was incredibly helpful for finding bugs in my compiler. Compared to C++ code I wrote myself, the Csmith code made it much easier to find bugs. While writing an operating system for Why in C++ with some of the C++ standard library ported (thanks to LLVM's libcxx and libcxxabi projects), I would encounter bugs, but the C++ standard library's functions tend to have many subcalls and stepping through

all the function calls in the debugger was incredibly tedious. Csmith's generated code has more of a flat structure that makes it easy to isolate bugs quickly. Thanks to Csmith, my compiler is now capable of compiling a very large subset of C++ code, evidenced by its ability to compile my operating system.

References

Anderson, Sean Eron (2005). *Bit Twiddling Hacks*. https://graphics.stanford.edu/~seander/bithacks.html.

Regehr, John et al. (June 2012). "Test-Case Reduction for C Compiler Bugs." In: *SIGPLAN Not.* 47.6, pp. 335–346. ISSN: 0362-1340. DOI: 10.1145/2345156.2254104.

Yang, Xuejun et al. (June 2011). "Finding and Understanding Bugs in C Compilers." In: *SIGPLAN Not.* 46.6, pp. 283–294. ISSN: 0362-1340. DOI: 10.1145/1993316.1993532.