Fuzzing Clang to find ABI Bugs

David Majnemer

What’s in an ABI?

- The size, alignment, etc. of types
- Layout of records, RTTI, virtual tables, etc.
- The decoration of types, functions, etc.
- To generalize: anything that you need N > 1 compilers to agree upon

C++: A complicated language

```
union U {
    int a;
    int b;
};

int U::*x = &U::a; int U::*y = &U::b;
```

Does ‘x’ equal ‘y’ ?

ISO/IEC JTC1 SC22 WG21 N 4141
Date: 2014-09-02
ISO/IEC FDIS 14882
ISO/IEC JTC1 SC22
Secretariat: ANSI

Programming Languages — C++
Langages de programmation — C++

We’ve got a standard

How hard could it be?

“[T]wo pointers to members compare equal if they would refer to the same member of the same most derived object or the same subobject if indirection with a hypothetical object of the associated class type were performed, otherwise they compare unequal.”

No ABI correctly implements this.

Why does any of this matter?

- Data passed across ABI boundaries may be interpreted by another compiler
    - Unpredictable things may happen if two compilers disagree about how to interpret this data
- Subtle bugs can be some of the worst bugs

Finding bugs isn't easy

- ABI implementation techniques may collide with each other in unpredictable ways
  - One compiler permutes field order in structs if the alignment is 16 AND it has an empty virtual base AND it has at least one bitfield member AND …
- Some ABIs are not documented
  - Even if they are, you can't always trust the documentation

What happens if we aren’t proactive

- Let users find our bugs for us
  - This can be demoralizing for users, eroding their trust
  - Altruistic; we must hope that the user will file the bug
  - At best, the user’s time has been spent on something they probably didn’t want to do

Let computers find the bugs

1. Generate some C++
2. Feed it to the compiler
3. Did the compiler die? If so, we have an interesting test case
4. If not, let’s ask another compiler to do the same
5. Compare the output of the two compilers

What we managed to attack

- External name generating (name mangling)
- Virtual table layout
- Thunk generation
- Record layout
- IR generation

In the beginning, there was record layout

- Thought to be high value, low effort to fuzz
- Generate a single TU execution test; expected identical results upon execution
- We want full coverage but without an excessive number of tests

- The plan for version 0.1 of the fuzzer seemed unambitious
  - Generate hierarchies of classes
  - Fill classes with fields
    - Support C scalar types (int, char, etc.)
    - Support bitfields
    - No arrays, pointer to member functions, etc.
    - No virtual methods
    - No pragmas or attributes
  - Dump offsets of fields
    - All classes must have a constructor

First steps…

Let’s generate some hierarchies…

First steps…

```
struct A { };
struct B : virtual A { };
struct C : virtual B, A { };
```

warning C4584: 'C': base-class 'A' is already a base-class of 'B'

```
struct A { };
struct B : virtual A { };
struct C : A, virtual B { };
```

error C2584: 'C': direct base 'A' is inaccessible; already a base of 'B'

First Lesson

- Successful fuzzing requires a model of what good test cases should look like
  - High failure rate can completely cripple the fuzzer
  - Less restrictive is better than more restrictive, you might lose out on test cases otherwise

- Fuzzer 0.1, while quite limited, was wildly successful
- Support for #pragma pack and __declspec(align) was added…

A typical test case

```
struct A {};
struct B {};
#pragma pack(push, 1)
struct C : virtual A,
           virtual B {
};
#pragma pack(pop)
struct D : C {};
```

- alignof(C) == 1, correct
- alignof(D) == 1, wrong!
  - correct answer is 4

It’s like whack-a-mole

```
struct A {};
struct B {};
struct C : virtual A,
           virtual B {
};
#pragma pack(push, 1)
struct D : C {};
#pragma pack(pop)
```

- alignof(C) == 4, correct
- alignof(D) == 4, wrong!
  - correct answer is 1

Testing synthesis of default operators

- Copy constructor IR generation is sophisticated
  - Tries to use memcpy if it’s valid & profitable, otherwise falls back to field-by-field initialization
  - Sophistication comes at a cost: complexity
    - ABI-specific assumptions baked into generic code, resulting in “surprising” IR
  - Fuzz tested by sticking ‘dllexport’ on all classes
    - Forces emission of **all** special member operators

C++ type to LLVM IR type

- We need an IR type for a particular C++ type in different contexts
- Surprisingly leads to different IR types for the same C++ type
  - Increased attack surface

Meet CGRecordLayout

```
union U {
    double x;
    long long y;
};

U u;

%union.U = type { double }

@u = global %union.U zeroinitializer
```

- We asked the compiler to "zero-initialize" u
- First *named* union member is initialized
  - Shocking number of compilers get this wrong
- Code is *relatively* simple, largely powered by AST layout algorithms

Meet ConstStructBuilder

```
union U {
    double x;
    long long y;
};

U u = { .y = 0 };

%union.U = type { double }

@u = global { i64 } { i64 0 }
```

- We asked the compiler to “aggregate-initialize” u
- Can’t use %union.U to initialize, wrong type
    - Anonymous type used instead
- Slavishly builds a new type from scratch
    - Has its own bitfield layout algorithm!

- CGRecordLayout
    - Used for “zero-initialization”
    - “Memory type”, used for loads and stores
- ConstStructBuilder
    - Used for aggregate initialization (C99 designated initializers, C++11 initializer lists)
- This seems complicated, why not let one rule them all?
    - CGRecordLayout is useful, largely reduces the number of new types we need but cannot *always* be used for aggregate initialization
    - ConstStructBuilder can handle aggregate initialization but has no idea how to handle virtual bases, vtordisps, etc.
    - These problems aren’t insurmountable but they aren’t trivial either :(

What about virtual tables?

- Some ABIs have a virtual base table and a virtual function table, others concatenate both into one table
- Virtual function table entries might point to virtual functions or to thunks which then delegate to the actual function body
  - Thunk might adjust the ‘this’ pointer, the returned value or both!
- RTTI data lives in the virtual function table
  - Composed of complex structures which describe inheritance structure, layout, accessibility, etc.

Comparing VTables

- Initial virtual function table comparer was a wrapper around llvm's obj2yaml
  - Worked excellently at first, eventually became a bottleneck
- A dedicated tool was written, llvm-vtabledump
  - More sophisticated: can parse RTTI data, dump virtual base offsets, etc.

```
S::`vftable'[0]: const S::`RTTI Complete Object Locator'
S::`vftable'[4]: public: virtual void * __thiscall S::`destructor'(unsigned int)
S::`vbtable'[0]: -4
S::`vbtable'[4]: 4
S::`RTTI Complete Object Locator'[IsImageRelative]: 0
S::`RTTI Complete Object Locator'[OffsetToTop]: 0
S::`RTTI Complete Object Locator'[VFPtrOffset]: 0
S::`RTTI Base Class Array'[0]: S::`RTTI Base Class Descriptor at (0,-1,0,64)'
```

A typical VTable testcase

```
struct A {
    virtual A *f();
};

struct B : virtual A {
    virtual B *f();
    B() {}
};

struct C : virtual A, B {};
```

- Clang’s vftable for C:
  - A* B::f() [thunk]
- MS’ vftable for C:
  - B* B::f() [thunk]
  - B* B::f()
- Both compilers are wrong!
  - A* B::f() [thunk]
  - B* B::f()

A cute trick used for pure classes

```
struct A {
    virtual A *f() = 0;
};

struct B : virtual A {
    virtual B *f() = 0;
};
```

- Would like to be able to reference virtual function table
- Can’t construct an object of type A or B
- Don’t want to add ctor or dtor, both have ABI implications
- __declspec(dllexport) references the vftable so it may be exported ;)

This approach worked marvelously for RTTI

- RTTI was the first complex component started after the fuzzer was written
  - Feedback loop was created, made it possible to iteratively improve compatibility
- Zero known bugs in RTTI as of this talk

Virtual tables don’t seem so hard, what’s the big deal?

- It turns out the other compiler has bugs (*cue gasps*)
  - Develop heuristics to determine when clang is correct and they are incorrect
  - We hope we didn’t miss any interesting cases :(
- **Non-virtual** *overloads* can have an effect on virtual table contents

String literals

- Some ABIs mangle their string literals
    - Wait, seriously?
        - Yeah, that way they merge across translation units

Examples

- “hello!” turns into “??_C@_06GANFPHOD@hello?$CB?$AA@“
- L“hello!” turns into “??_C@_1O@IMICCIOB@?$AAh?$AAe?$AAl?$AAl?$AAo?$AA?$CB?$AA?$AA@”
- Wonderful, right?

Custom fuzzer written

- I *thought* I was on the right track but I wanted to be sure, this was easily tested with a purpose-built fuzzer

```
// <char-type> ::= 0   # char
//             ::= 1   # wchar_t
//             ::= ??? # char16_t/char32_t will need a mangling too...
//
// <literal-length> ::= <non-negative integer>  # the length of the literal
//
// <encoded-crc>    ::= <hex digit>+ @          # crc of the literal including
//                                              # null-terminator
//
// <encoded-string> ::= <simple character>           # uninteresting character
//                  ::= '?$' <hex digit> <hex digit> # these two nibbles
//                                                   # encode the byte for the
//                                                   # character
//                  ::= '?' [a-z]                    # \xe1 - \xfa
//                  ::= '?' [A-Z]                    # \xc1 - \xda
//                  ::= '?' [0-9]                    # [,/\:. \n\t'-]
//
// <literal> ::= '??_C@_' <char-type> <literal-length> <encoded-crc>
//               <encoded-string> '@'
```

Is this approach effective?

98 MS ABI bugs found since the fuzzer was written:

17748 17750 17761 17767 17768 17772 17816 **18021** 18022 **18024**
18025 18026 18027 18035 **18039** 18118 **18167 18168 18169 18170**
18172 18173 18175 **18186 18215 18216** 18248 **18264 18278 18433**
18434 18435 18436 18437 18444 18464 18467 18474 18476 18479
18617 18618 18675 **18692 18694 18702 18826** 18844 18845 18880
18902 18917 18951 18967 19012 19025 19066 19104 19172 **19180**
19181 19240 19361 19398 **19399 19407 19408 19413 19414 19487**
19505 19506 19733 20017 20047 20221 **20315** 20343 20351 20418
20444 20464 20477 20479 20653 20719 20897 20947 21022 21164
21031 21046 21060 21061 21062 21064 21073 21074

Bug numbers in bold are bugs found by superfuzz.

Thanks!