

The Type Sanitizer: Free Yourself from `-fno-strict-aliasing`

Hal Finkel
Argonne National Laboratory

2017 LLVM Developers' Meeting

An Example

```
$ cat /tmp/clever.c
#include <stdio.h>
#include <math.h>
```

```
float i_am_clever(unsigned int *i, float *f) {
    if (!isnan(*f))
        *i ^= 1 << 31;
    return *f;
}
```

Do we need to load *f again here?

```
int main() {
    float f = 5;
    f = i_am_clever((unsigned int *) &f, &f);
    printf("%f\n", f);
}
```

An Example

```
$ gcc -o /tmp/c /tmp/clever.c  
$ /tmp/c  
-5.000000  
$ gcc -o /tmp/c /tmp/clever.c -O3  
$ /tmp/c  
5.000000  
$ gcc -o /tmp/c /tmp/clever.c -O3 -fno-strict-aliasing  
$ /tmp/c  
-5.000000
```


```
$ clang -o /tmp/c ~/tmp/clever.c  
$ /tmp/c  
-5.000000  
$ clang -o /tmp/c ~/tmp/clever.c -O3  
$ /tmp/c  
5.000000  
$ clang -o /tmp/c ~/tmp/clever.c -O3 -fno-strict-aliasing  
$ /tmp/c  
-5.000000
```

Clang and GCC are similar in this regard...

An Example

```
$ clang -o /tmp/c /tmp/clever.c -fsanitize=type -g  
$ /tmp/c
```

```
==28127==ERROR: TypeSanitizer: type-aliasing-violation on address 0x7fff042dcf8c (pc 0x000000420ff5 bp 0x7fff042dcd80 sp 0x7fff042dc500 tid 28127)  
READ of size 4 at 0x7fff042dcf8c with type int accesses an existing object of type float  
#0 0x420ff4 in i_am_clever /tmp/clever.c:6:8  
#1 0x421755 in main /tmp/clever.c:12:7  
#2 0x7f13b0ef3c04 in __libc_start_main (/lib64/libc.so.6+0x21c04)  
#3 0x402a14 in _start (/tmp/c+0x402a14)  
  
==28127==ERROR: TypeSanitizer: type-aliasing-violation on address 0x7fff042dcf8c (pc 0x00000042117c bp 0x7fff042dcd80 sp 0x7fff042dc500 tid 28127)  
WRITE of size 4 at 0x7fff042dcf8c with type int accesses an existing object of type float  
#0 0x42117b in i_am_clever /tmp/clever.c:6:8  
#1 0x421755 in main /tmp/clever.c:12:7  
#2 0x7f13b0ef3c04 in __libc_start_main (/lib64/libc.so.6+0x21c04)  
#3 0x402a14 in _start (/tmp/c+0x402a14)  
  
-5.000000
```



```
READ of size 4 at 0x7fff042dcf8c with type int accesses an existing object of type float
```

The Rules

6.10 Lvalues and rvalues

[basic.lval]

⁸ If a program attempts to access the stored value of an object through a glvalue of other than one of the following types the behavior is undefined:⁵⁶

- (8.1) — the dynamic type of the object,
- (8.2) — a cv-qualified version of the dynamic type of the object,
- (8.3) — a type similar (as defined in 7.5) to the dynamic type of the object,
- (8.4) — a type that is the signed or unsigned type corresponding to the dynamic type of the object,
- (8.5) — a type that is the signed or unsigned type corresponding to a cv-qualified version of the dynamic type of the object,
- (8.6) — an aggregate or union type that includes one of the aforementioned types among its elements or non-static data members (including, recursively, an element or non-static data member of a subaggregate or contained union),
- (8.7) — a type that is a (possibly cv-qualified) base class type of the dynamic type of the object,
- (8.8) — a `char`, `unsigned char`, or `std::byte` type.

Clang's vector types are also included in this list.

⁵⁶) The intent of this list is to specify those circumstances in which an object may or may not be aliased.

TBAA Metadata

```
...  
store i32* %i, i32** %i.addr, align 8, !tbaa !3  
...  
%348 = load float*, float** %f.addr, align 8, !tbaa !3  
...  
%409 = load float, float* %348, align 4, !tbaa !7  
...  
store i32 %xor, i32* %870, align 4, !tbaa !9  
...  
!3 = !{!4, !4, i64 0}  
!4 = !{"any pointer", !5, i64 0}  
!5 = !{"omnipotent char", !6, i64 0}  
!6 = !{"Simple C/C++ TBAA"}  
!7 = !{!8, !8, i64 0}  
!8 = !{"float", !5, i64 0}  
!9 = !{!10, !10, i64 0}  
!10 = !{"int", !5, i64 0}
```

All pointers are the same.

The root for C++ code.

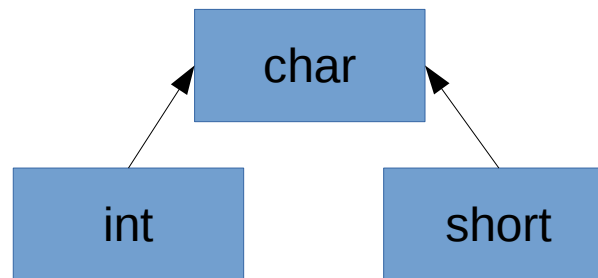
TBAA Metadata

Access Tag: (Base Type, Access Type, Offset)

- For scalar accesses, the base type == access type
- The base/access type is:

(name, member 1 type, offset 1, member 2 type, offset 2, ...)

Scalar types, not just structure types, have the above form. Scalar types don't have members, but they do have parent types...



TBAA Metadata

```
struct Inner {  
  int i;    // offset 0  
  float f; // offset 4  
};
```

```
struct Outer {  
  float f;           // offset 0  
  double d;         // offset 8  
  struct Inner inner_a; // offset 16  
};
```

An access to:

Outer::inner_a::i, Outer @ offset 16

Can alias with...

Inner::i, Inner @ offset 0

Can alias with...

int "@ offset 0"

Can alias with...

char "@ offset 0"

TBAA Metadata

```
struct Inner {  
    int i;    // offset 0  
    float f; // offset 4  
};
```

```
struct Outer {  
    float f;           // offset 0  
    double d;         // offset 8  
    struct Inner inner_a; // offset 16  
};
```

```
!2 = !{!3, !3, i64 0}  
!3 = !{"any pointer", !4, i64 0}  
!4 = !{"omnipotent char", !5, i64 0}  
!5 = !{"Simple C/C++ TBAA"}  
!6 = !{!7, !8, i64 0}  
!7 = !{"Outer", !8, i64 0, !9, i64 8, !10, i64 16}  
!8 = !{"float", !4, i64 0}  
!9 = !{"double", !4, i64 0}  
!10 = !{"Inner", !11, i64 0, !8, i64 4}  
!11 = !{"int", !4, i64 0}  
!12 = !{!7, !11, i64 16}
```

Start here and work backward.

The Type Sanitizer

Clang

- -fsanitize=type
- Always produce TBAAs metadata, even at -O0
- Add type metadata to globals
- Link with the tysan runtime library

LLVM

- Don't use TBAAs metadata for pointer-aliasing analysis
- Instrument access and generate type descriptors
- Disable some “sanitizer unfriendly” optimizations

compiler-rt

- Uses shadow memory to record access types for memory ranges
- Uses TBAAs algorithm at runtime to check access legality
- Reports illegal accesses to the user

Shadow Memory

access descriptor	-1	-2	-3	access descriptor	-1	access descriptor	-1	0	...
-------------------	----	----	----	-------------------	----	-------------------	----	---	-----

4-byte access (scalar) type

2-byte access (scalar) type

Each box above is sizeof(void *) bytes.

Shadow Address = (((Access Address) & __tysan_app_memory_mask) * sizeof(void*)) + __tysan_shadow_memory_address

Descriptors

Access descriptor:

1	base-type desc. ptr.	access-type desc. ptr.	offset
---	----------------------	------------------------	--------

Type descriptor:

2	member count	member desc. ptr.	member offset	...	name
---	--------------	-------------------	---------------	-----	------

- Except for types in anonymous namespaces, use comdat for each descriptor.
- For unnamed types, hash the structure to make a unique name.

Instrumentation

- Reset shadow memory to zero for:
 - `byval` arguments and `allocas` (i.e., new stack allocations)
 - `lifetime_start/lifetime_end`
 - `memset`
- For `memcpy/memmove`, do the same for the corresponding shadow memory
- For a memory access, if the type is unknown (all zeros), set the type in shadow memory. If the type is set, then check that it matches (i.e., that the first shadow memory value is the type descriptor and the remaining values are -1, -2, ...). If it does not match, call the runtime (which may nevertheless determine that the access is legal).

Interceptors

Intercept system functions to...

- Reset the shadow memory to zero (i.e., mark the type as unknown)
 - memset, mmap, malloc, and related functions
- Copy the corresponding shadow memory
 - memcpy, memmove, strdup

Writing interceptors is easy...

```
INTERCEPTOR(int, posix_memalign, void **memptr, uptr alignment, uptr size) {  
    int res = REAL(posix_memalign)(memptr, alignment, size);  
    if (res == 0 && *memptr)  
        tyan_set_type_unknown(*memptr, size);  
    return res;  
}  
...  
INTERCEPT_FUNCTION(posix_memalign);
```

Shadow Memory

Allocate unreserved (i.e., unbacked) pages for the shadow memory based on how each architecture uses its address space...

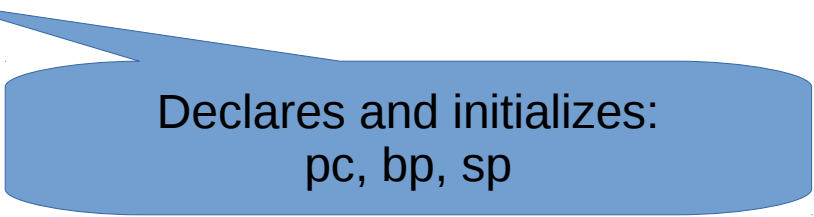
```
#if defined(__x86_64__)
struct Mapping {
    static const uptr kShadowAddr = 0x010000000000ull;
    static const uptr kAppAddr = 0x550000000000ull;
    static const uptr kAppMemMsk = ~0x780000000000ull;
};
...
__tysan_shadow_memory_address = ShadowAddr();
__tysan_app_memory_mask = AppMask();
...
MmapFixedNoReserve(ShadowAddr(), AppAddr() - ShadowAddr());
```

Printing Errors

How do you generate those characteristic sanitizer error messages and stack traces?

First, record information about the caller when you enter the runtime...

```
extern "C" SANITIZER_INTERFACE_ATTRIBUTE
void __tysan_check(void *addr, int size, tysan_type_descriptor *td,
                  int flags) {
    GET_CALLER_PC_BP_SP;
```



Declares and initializes:
pc, bp, sp

Printing Errors

Next, make use of provided functions for printing and the stack trace...

```
Decorator d;  
Printf("%s", d.Warning());  
Report("ERROR: TypeSanitizer: type-aliasing-violation on address %p"  
      " (pc %p bp %p sp %p tid %d)\n",  
      Addr, (void *) pc, (void *) bp, (void *) sp, GetTid());  
Printf("%s", d.End());  
Printf("%s of size %d at %p with type ", AccessStr, Size, Addr);  
...  
if (pc) {  
    BufferedStackTrace ST;  
    ST.Unwind(kStackTraceMax, pc, bp, 0, 0, 0, false);  
    ST.Print();  
} else {  
    Printf("\n");  
}
```

Another Example

```
$ cat /tmp/so.c
#include <stdio.h>
#include <stdlib.h>
```

```
struct X {
    int i;
    int j;
};
```

```
int foo(struct X *p, struct X *q) {
    q->j = 1;
    p->i = 0;
    return q->j;
}
```

```
int main() {
    unsigned char *p = malloc(3 * sizeof(int));
    printf("%i\n", foo((struct X *) (p + sizeof(int)),
                      (struct X *) p));
}
```

WRITE of size 4 at
0x000002712014 with type int (in
X at offset 0) accesses an existing
object of type int (in X at offset 4)

Partial Overlaps

The instrumentation and runtime deals with different overlap cases:

- The current access points to the first byte of the previously-recorded type in memory
- The current access points to the middle of some previously-recorded type in memory
- Not the first byte, but some later bytes, of the current access overlap with some previously-recorded type in memory

READ of size 4 at ... with type float accesses part of an existing object of type long that starts at offset -4

An Experiment

As has been previously identified by others [1], the popular XML parser library Expat, violates type-aliasing rules. Compiling Expat 2.2.0 with the Type Sanitizer and executing the “runtests” program reports 2613 errors, including many like:

“READ of size 8 at ... with type any pointer (in attribute_id at offset 0) accesses an existing object of type any pointer (in <anonymous type> at offset 0)”

“READ of size 4 at ... with type int (in XML_ParserStruct at offset 512) accesses an existing object of type int (in prolog_state at offset 8)”

[1] “Detecting Strict Aliasing Violations in the Wild”
<http://trust-in-soft.com/wp-content/uploads/2017/01/vmcai.pdf>

Future Enhancements

- “Sticky” types for local/global variables (and more) – Some variables have declared types and those types can be set “up front”, and shouldn't be changed later by accesses.
- (Optional) Origin tracking – Currently the stack trace shows the location of the illegal access but not the location of the code that set the type.
- Better handling of unions and arrays – This requires enhancements to the TBAA representation (such enhancements are currently under discussion).

Acknowledgments

- The LLVM community
- ALCF, ANL, and DOE
- ALCF is supported by DOE/SC under contract DE-AC02-06CH11357

<https://reviews.llvm.org/D32199> (Clang)

<https://reviews.llvm.org/D32197> (Runtime)

<https://reviews.llvm.org/D32198> (LLVM)