

The best way to find bugs  
hiding in your spaghetti is to  
feed it to the sanitizers

İsmail Pazarbaşı <ismail.pazarbasi@gmail.com>  
@ipazarbasi

I'm working as a software engineer at Cisco. I am not working on sanitizers, Clang or any other compiler in my job.



These are a few examples from our product portfolio. My team is making this blue application -- GUI for our products. On top left, you can see picture of what we call "an immersive system". It's a really big and smart system, comes with a desk, which has too many microphones, and universal connectors for sharing laptop screens during meetings. We have additional solutions for that, but I am unsure whether I am allowed to talk about that. These endpoints can recognize whiteboards and faces during meetings, and can track speakers in the room. This enables us to frame the correct region of the meeting. On the left, you can see the touch panel, which shares a great deal of code with GUI. These are for meeting room systems. The bottom one is a cheaper model that can be connected to any screen with HDMI input. Therefore, it can be used in small business meeting rooms or at homes very easily. It's the same GUI that runs on all of these, and more, products. That requires extensive testing, and doing things right. It's exciting, because we are always on the screen; since the first time setup, user interacts directly with our code. We are the bridge between a very complex machine and a person -- or group of people -- who wants to make a video call.

# Why am I presenting sanitizers?

I try to spend a few hours every few weeks to patch clang. Looking into clang bugs, reading mailing lists and submitting patches have been very educational for me, and I enjoy it a lot. A few years ago, I have realized how little I knew about CodeGen. While looking into that, I have met with ubsan, then asan. I didn't know how they worked -- it was all black magic. I saw test cases, and found them very impressive. My curiosity as to "how" they work has become unstoppable. I decided to understand these more in detail, and this is going to be imperfect summary of my learning experience.

*(Tens of) Thousands of lines of code must make sense - not just sequentially, not just within the same translation unit, but in certain permutations.*

*Rough approximation to Kevlin Henney's words*

Kevlin Henney has visited our site, and was talking about how complex source code can be. He's made a comparison with, IIRC, books... volumes of books. He said that books have millions of words, and thousands of sentences, and they make sense when written sequentially. He then said that software is significantly different than this model, because we generally have to make sense even if the execution is non-sequential; we generally execute different pieces of program in different permutations. That can become awfully complicated really quick.

# Bugs in the wild

- Software is intangible
- Murphy's right; anything that can go wrong, will go wrong
- Too many corner cases
- Number/size/complexity of bugs is proportional to complexity of software
- Job security, eh (:



Image source:  
<http://thebreakingtime.typepad.com>

We cannot see, touch, smell, taste or hear software. We can only detect its effects, much like electricity -- although it's fairly easy to "feel" electricity, if one is willing to. Several things can go wrong, even if a small weakness slips in. It's easy to neglect, underestimate or omit checks. We can't release software with such weaknesses, because someone -- either forcefully or by just having a bad luck might hit those cases. Remember Heartbleed? There are too many corner cases in the language, our reasoning about the algorithm, or the runtime. There are infinitely many cases, and correct ones are generally within a smaller subset. Every day, we add more features to our products. This generally requires more code. We make or use more complicated data structures, multiple threads and JITs. All these additions introduce a bug waiting to be born. Last, but not least.. we need bugs for job security!

# Bug hunting

- Compiler diagnostics
- Multiple compilers
- Static analysis tools
- Testing is paramount
- Sanitizers
- Fuzzing!



Source: Fallout Wiki

Compilers can diagnose provably wrong code; such as signed-unsigned comparisons or array out-of-bounds access. In certain cases, it's possible to perform compile-time checks. Testing with multiple compilers is a great practice, because not all compilers diagnose all cases. User might be relying on a compiler bug, or non-standard extension. We'll see a small difference between GCC and Clang shortly. Static analysis tools help discovering many problems before product is even tested! This is awesome! I don't think any sane vendor is shipping software without tests, but testing is paramount, we've got to repeat it. Sanitizers are best to be integrated with tests. I don't think it's a good idea to release instrumented code. Sanitizers will be useless, if you don't test!

# class Insecta;

- First of all, some of them are "features" that are implemented by certain someone who was smoking something funny
- Logic errors in well-formed program
- Problems with graphics or sound, placement of visual elements
- Undefined behavior cases
- Incorrect use of libraries

In the biological classification, bugs we know fall under Insecta class. Among these bugs, we are interested in undefined behavior cases, and incorrect use of libraries. We are not interested in undefined behavior in preprocessor either; only things happen in translation steps 7 and after.

```
1. #define MAC(a,b) a+b
3. int i =
5. #include "main.h" // constains:  MAC( 2,
7. 3
8. );
9.
```

In the biological classification, bugs we know fall under Insecta class. Among these bugs, we are interested in undefined behavior cases, and incorrect use of libraries. We are not interested in undefined behavior in preprocessor either; only things happen in translation steps 7 and after.

# class Insecta;

- First of all, some of them are "features" that are implemented by certain someone who was smoking something funny
- Logic errors in well-formed program
- Problems with graphics or sound, placement of visual elements
- Undefined behavior cases
- Incorrect use of libraries

In the biological classification, bugs we know fall under Insecta class. Among these bugs, we are interested in undefined behavior cases, and incorrect use of libraries. We are not interested in undefined behavior in preprocessor either; only things happen in translation steps 7 and after.

# Cost of bugs

- We spend time. Our time is the most precious, and expensive commodity
- Products delay. Revenue will delay, and decrease, as we spend time on bugs
- Companies lose prestige, and money
- Customers may lose money directly, due to miscalculation in our buggy software
- Sensitive information maybe leaked
- Life-support, avionics, Mars rover, etc. software might misinterpret data, crash, cause loss of lives or burn years of work
- Quick & dirty fixes help bug reproduction

Bugs are not free. In fact, they are really expensive.

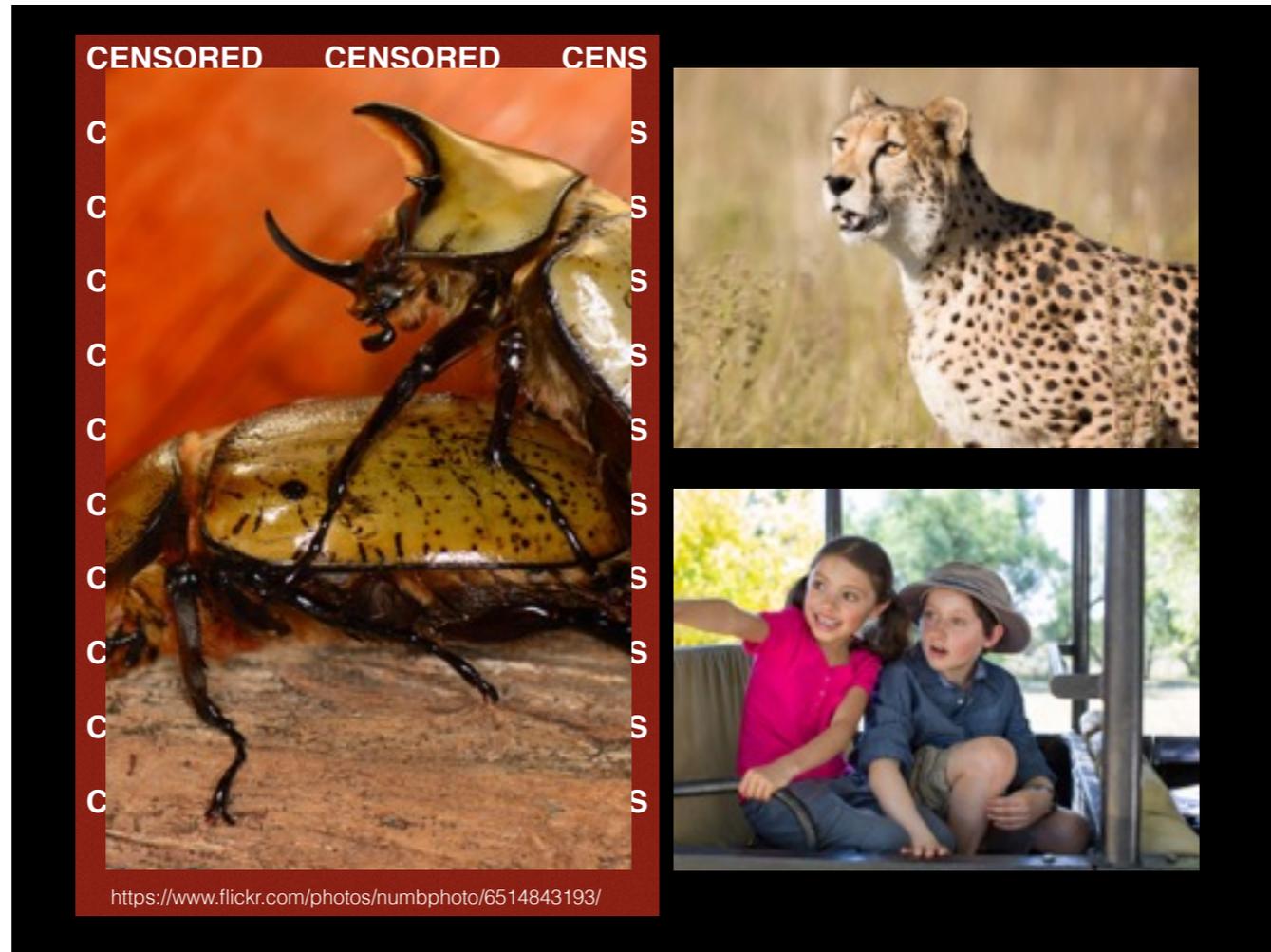


Exhibit A; reproducing bugs. It's bad... it's going to get worse, and everyone will be watching this.

# Memory problems

# Memory problems

- Leaks
- Overwrites
- Use-after-free
- Double-free
- Bounds violation
- Pointer juggling

Leaks are bad, though probably not the worst. Overwrites are really bad; can't be easily traced. Use-after-free is another frequently encountered serious issue. My experience suggests that this happens in C code, and C++ code passing pointers to C++ objects to C callbacks -- but then, C++ callback function is called from C with pointer to destroyed object. Bounds violation is also a frequently encountered problem, and has serious security implications. Pointer juggling, that is making assumptions about pointer arithmetics and relational comparison or making incorrect arithmetics can lead to undefined behavior. This has security implications, as it triggers undefined behavior, and compilers may not generate code or may generate unexpected code -- because, hey, they can!

# Multithreading

# Multithreading

- Ordering is hard, and enforcing it may defeat purpose of having multiple threads
- Hard-to-reproduce Heisenbugs

Surrounding everything with locks is a bad idea. That'll end up with a complicated sequential program. We have to acknowledge that things might happen in different order at each execution of given code. It's generally next to impossible to find a race without a tool in a complicated program, especially when compiled with optimizations.

Static (dis)order

# Static (dis)order

- Order of initialization, and, therefore, destruction, of variables with static storage duration in different translation units is unspecified.
- FAQ says... <URL: <http://isocpp.org/wiki/faq/ctors#static-init-order>>

*The tragedy is that you have a 50%-50% chance of dying. [...] I hear they're hiring down at McDonalds. Enjoy your new job flipping burgers. [...] If you think it's "exciting" to play Russian Roulette with live rounds in half the chambers, you can stop reading here.*

This is not a frequent issue, at least in my experience, but could be hard to trace. This is what C++ FAQ says about static initialization order.

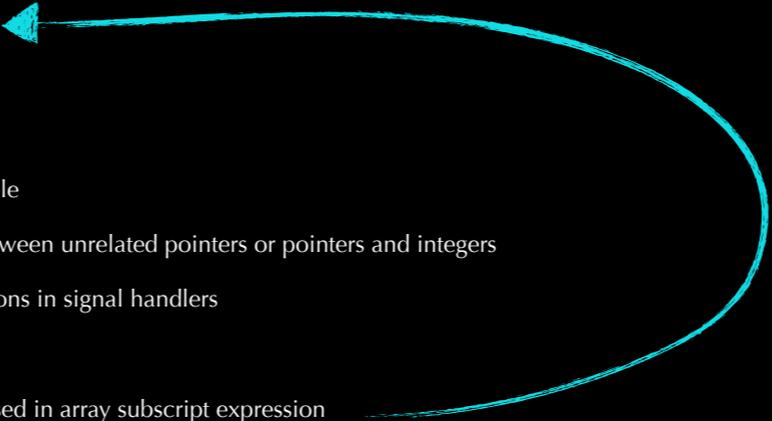
```
root@all-evil% █
```

We all know that... root of all evil is undefined behavior! I thought about finger'ing root@all-evil, but ... this is less offensive. Almost everything we're going to talk about is undefined behavior. Per C and C++ standards, there are several categories as far as behavior is concerned. Unspecified behavior is use of unspecified value or other behavior where standard sets no requirements. Implementation defined behavior is when behavior is left unspecified by the standard, but implementations document it. Undefined behavior is the behavior that is not defined by the standard or explicitly stated that the behavior would be undefined.

```
root@all-evil% undefined-behavior
```

We all know that... root of all evil is undefined behavior! I thought about finger'ing root@all-evil, but ... this is less offensive. Almost everything we're going to talk about is undefined behavior. Per C and C++ standards, there are several categories as far as behavior is concerned. Unspecified behavior is use of unspecified value or other behavior where standard sets no requirements. Implementation defined behavior is when behavior is left unspecified by the standard, but implementations document it. Undefined behavior is the behavior that is not defined by the standard or explicitly stated that the behavior would be undefined.

# *Real* undefined behavior

- Dereferencing null pointer
  - Modifying variable multiple times in one sequence point
  - Data race
  - Array bounds violation
  - Use-after-free, double-free
  - Breaking aliasing rules
  - Use of uninitialized variable
  - Relational comparison between unrelated pointers or pointers and integers
  - Using signal-unsafe functions in signal handlers
  - Signed-integer overflow
  - Signed-integer overflow used in array subscript expression
  - and so on...
- 

This class of undefined behavior is real. These will either crash the program or yield unexpected results when built and run on different platforms with different compilers. It usually doesn't take too long to spot these. Signed integer overflow might mean some piece of code would never be generated.

# Somewhat-undefined behavior

- `static_cast<T*>(0)->foo();` // `foo` is a static member function
  - Highly likely, all compilers will ignore what standard says (evaluate object expression)
- Using unions for type-punning; this is undefined, but everyone uses it, and gets away with it

```
2. union U { int i; float f; } u;  
3. // ...  
4. u.i = 0; // `f` dies here §3.8/p1  
5. u.f = 42.0f; // undefined; f's storage is gone  
6. new (&u.f) float(42.0f); // OK
```

I haven't seen any compiler that evaluated object expression so far. Maybe it's time to define this behavior! The last bullet is interesting. We've discussed this several times, both publicly and at Cisco... this is undefined behavior. The correct way, in C++, is shown at the bottom.

# Security!

I have to underline the fact that every undefined behavior is a potential security vulnerability, a time bomb. Kostya Serebryany has mentioned this at his presentation as well, and I haven't seen many people emphasize this aspect enough. Security is particularly important in these days; after Snowden, and corporate espionage cases. Cisco is connecting the world, and a bit of things orbiting the Earth. I can't disclose where our products are deployed, but everyone knows that they are deployed at many companies, from larger ones to smaller ones. They are also on satellites, universities, and state institutions. Cisco also runs data centers. We take security very seriously.

We've got to do  
something about it!



Test it  
properly!

I don't think anyone is shipping software without properly testing, but just in case, I'd like to emphasize that if you don't test your software, nothing can save you. Your program may not suffer from undefined behavior, but it might have logic error. Having a good test coverage means having a good confidence.

*“Testing software is like dressing up decently before the public. Going out naked is indecent, and embarrassing.”*

Shameless plug. I believe in this, I've made it up. That is what I think about testing.

# Static analysis

Static analysis is important, and shouldn't be ignored.

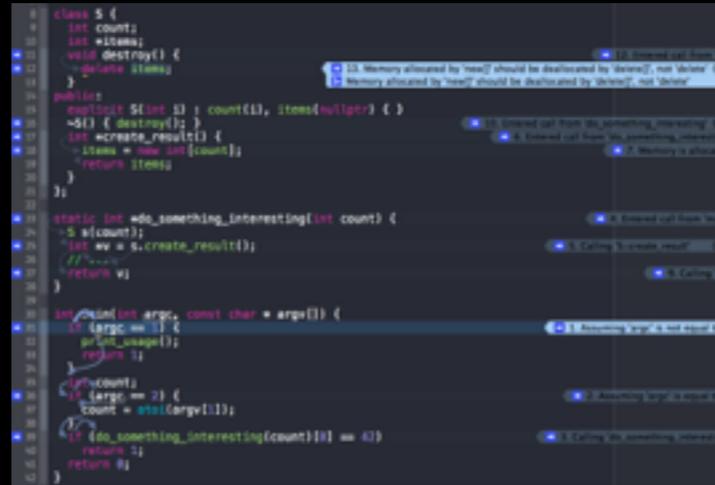
# Static checks/analysis

- It's best to detect problems early - even before testing, when possible
- Compilers generally know more C and C++ than you
- Many static analysis tools are available (both free and commercial)
- We don't need a reason to run static analyzer; it should be run always
- Static analyzers generally do great job at diagnosing issues in single translation unit

One of the take aways is that you might not have test coverage for certain parts of the program. Perhaps you have no test coverage for certain functions, or perhaps certain paths are never taken under test execution. Static analysis helps finding such things, because it will discover these paths, even if the program is never executed.

# Static analysis

- Can perform expensive checks that can't be done during compilation
- Does path sensitive analysis (i.e. bugs that occur only if a certain path of CFG is taken)
- Might catch problems tests didn't cover
- Clang static analyzer is very flexible; new checkers can be added with shared libraries
- Clang repo has a shell script to add your checker to Xcode!



```
1 class S {
2     int count;
3     int *items;
4     void destroy() {
5         delete items;
6     }
7 public:
8     explicit S(int i) : count(i), items(nullptr) {}
9     ~S() { destroy(); }
10    int *create_result() {
11        items = new int[count];
12        return items;
13    }
14 };
15
16 static int *do_something_interesting(int count) {
17     S s(count);
18     int *r = s.create_result();
19     // ...
20     return r;
21 }
22
23 int main(int argc, const char * argv[]) {
24     if (argc == 1) {
25         print_usage();
26         return 1;
27     }
28     int count;
29     if (argc == 2) {
30         count = atoi(argv[1]);
31     }
32     int *r = do_something_interesting(count);
33     return 0;
34 }
```

The screenshot shows several Clang static analyzer warnings in a dark-themed editor. The first warning is on line 5, stating: "Memory allocated by 'new' should be deallocated by 'delete', not 'delete []'". The second warning is on line 11, stating: "Memory allocated by 'new' should be deallocated by 'delete', not 'delete []'". The third warning is on line 25, stating: "Memory allocated by 'new' should be deallocated by 'delete', not 'delete []'". The fourth warning is on line 30, stating: "Memory allocated by 'new' should be deallocated by 'delete', not 'delete []'".

Compiler shouldn't know about C's FILE\* or matching fopen-fclose calls. It's not compiler's job. Even for language constructs, such as dereferencing null pointer, compiler may only issue warning, if it's a clear-cut case in limited scope. Otherwise, these are static analysis jobs, primarily because they are expensive analysis that potentially require path-sensitive analysis and contextual knowledge. Static analyzer has different checks for different cases. Clang's static analyzer will symbolically execute the code, and discovers paths. It will report the problem, and the execution path it has used to reach that conclusion.

# Assertions

- For compile-time sanity checks, `static_assert`
- Always use `assert`
  - LLVM+Clang has 81671\* asserts spread across 2,649,826 LoC\*  $\approx$  every 32 line has 1 asserts

\* Numbers collected in early January, it may include LLDB code as well.

# Reading

- Reading the code is generally futile
- Reading about undefined behavior is not enough, considering number of possible cases
- Not everything you read is authentic (e.g. use of union to convert between float and int)

# Dynamic binary instrumentation

# Dynamic binary instrumentation

- Analyzes program behavior on run-time; doesn't require recompilation
- Disassembles, analyzes, then JITs its IR
- ^~~~~~Very Slow
- It can become prohibitively slow
- Some tools cannot run in multithreaded mode
- Cannot track stack objects
- Doesn't play nice with JITs (e.g. QML)

# Compile-time instrumentation

# GCC Mudflap

- GCC pass
- Instruments some memory accesses
- Runtime holds a tree of tracked locations
- Doesn't have red-zones between adjacent locations
- Doesn't work with DSOs (I didn't try myself)

Mudflap has been implemented as a GCC pass, and was able to instrument most of the interesting pointer operations. The pass was inserting checks around these operations. Mudflap kept its pointer address database as a tree. Unlike sanitizers, mudflap didn't have red-zones between adjacent objects.

# Sanitizers

- Both GCC and Clang include sanitizers\*
- Compiler injects code, runtime library *may* take an action
- Magnitudes of faster than dynamic binary instrumentation

\* clang shipping with Xcode does *not* include sanitizers

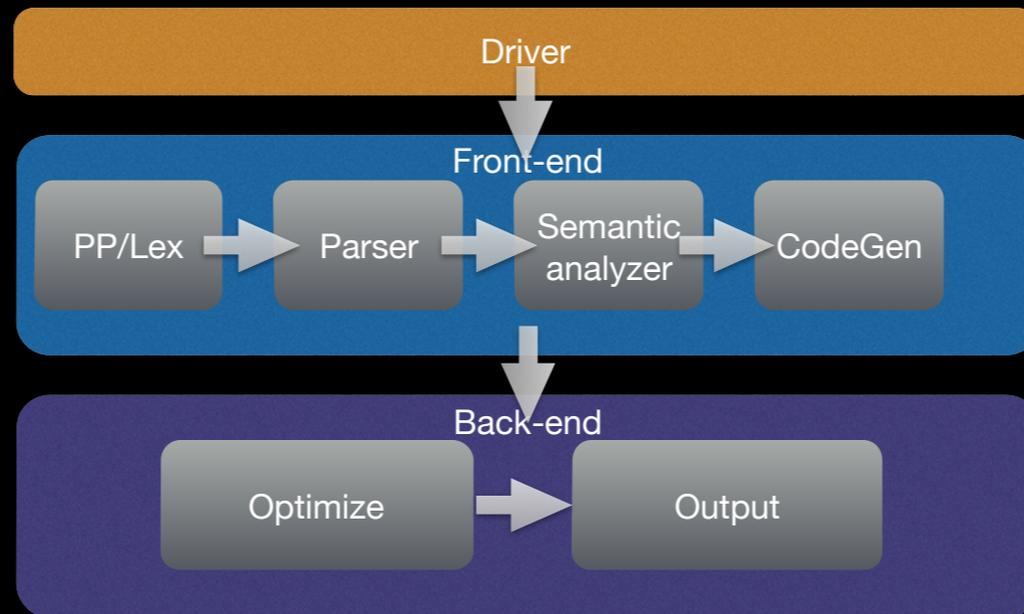
This presentation will not include DataFlow sanitizer or Go language support.

# Sanitizers - Caveats

- Requires recompilation with certain flags
- May require all dependencies to be instrumented with same sanitizer
- Memory footprint increases
- Compilation may slow down by 0-400%
- Execution may slow down by 0.1%-500%
- May not be available for certain targets

Sanitizers are compile-time instrumentations, therefore require code to be compiled again for instrumentation. Certain sanitizers require entire code, plus dependencies to be instrumented with the same sanitizer. Overall memory footprint will increase, except in UBSan. Compilation slows down, but this depends on sanitizer, and the code that is being instrumented. Execution will slow down, because sanitizers will run multiple checks, such as before every load from or write to memory. MSan and TSan will slow down more than others. If you are running tests on your non-x86 target system, some sanitizers may not be available. But maybe that's a good thing, so that you ensure that your code compiles both on target and host CPUs.

# C++ Code To Output



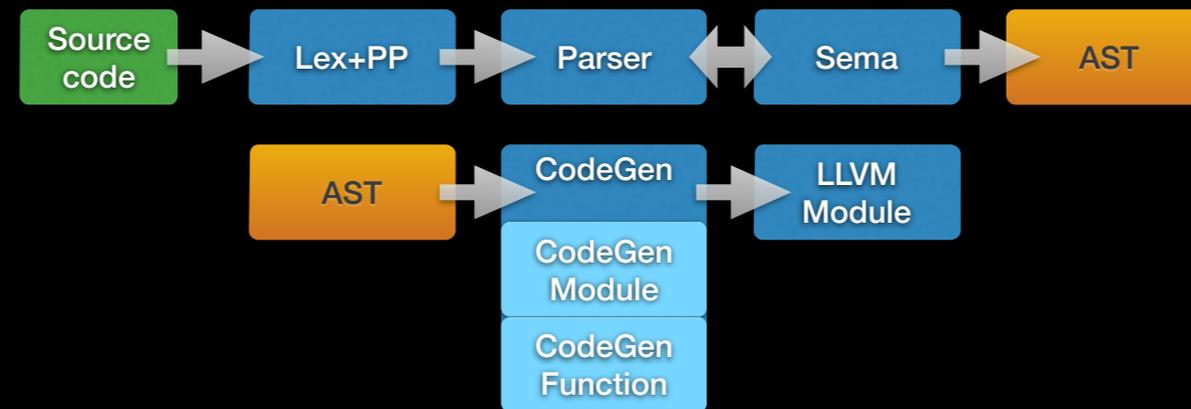
We'll see how C or C++ code is instrumented by Clang. When we run clang executable, what we run is generally the driver. We'll then take brief look into front-end, and back-end.

# Driver

- Invokes language front-end/linker with required parameter list
- Invokes multiple compilation jobs

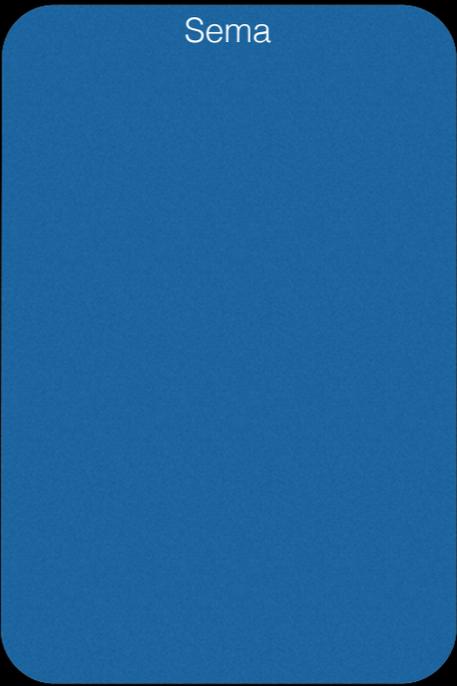
clang executable accepts multiple source files, and a set of flags. That executable is generally driver. I don't know whether people actually use front-end directly, except when writing tests. Driver invokes front-end with set of default include paths, CPU flags, etc. It does this for each input file specified, so that you don't have to repeat all the boring things all over again.

# Front-end



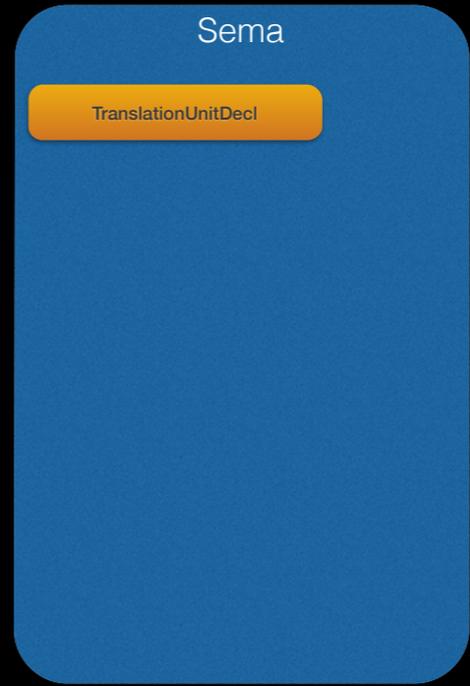
Code is picked up by lexer and preprocessor. This is then tokenized, and passed to parser. Parser analyzes token sequence, and asks Semantic analyzer whether given sequence of tokens make sense. Semantic analyzer creates AST piece-by-piece. Parser sometimes consults semantic analyzer whether given token is a type or template parameter, for example. When a valid AST is constructed, it's picked up by CodeGen, in our case. CodeGen will walk through AST, and create an LLVM module. CodeGen Module is responsible for the module-level IR, while CodeGen Function is generating function-level IR.

```
1.int Global;
2.void *Thread1(void *) {
3.  Global = 42;
4.  return NULL;
5.}
6.int main() {
7.  pthread_t t;
8.  pthread_create(&t, NULL, Thread1, NULL);
9.  ...
10.}
```

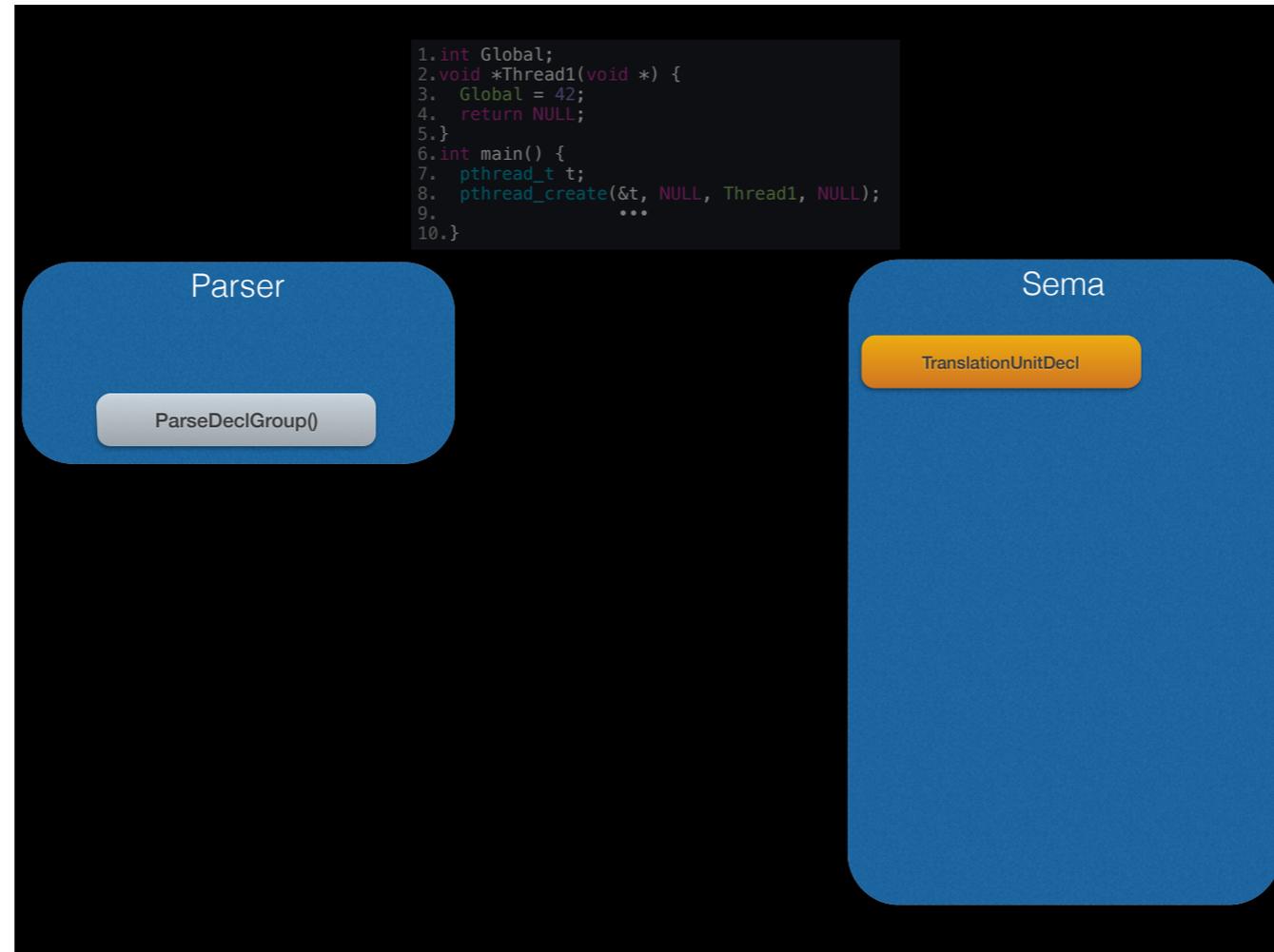


This is a skimmed summary of how clang creates an AST for given C or C++ code. It's imprecise, and skips a lot of steps, but should give the idea. When Sema starts, it creates `ASTContext`, which creates the top-level item -- `TranslationUnitDecl`. Parser takes `tok::kw_int`, and begins parsing a declaration group. There is no initializer, and there is no comma. It then finds a semi, and finishes group, which has only 1 declaration. Sema, then, takes this declaration, and puts it in AST. This process continues for other declarations and expressions.

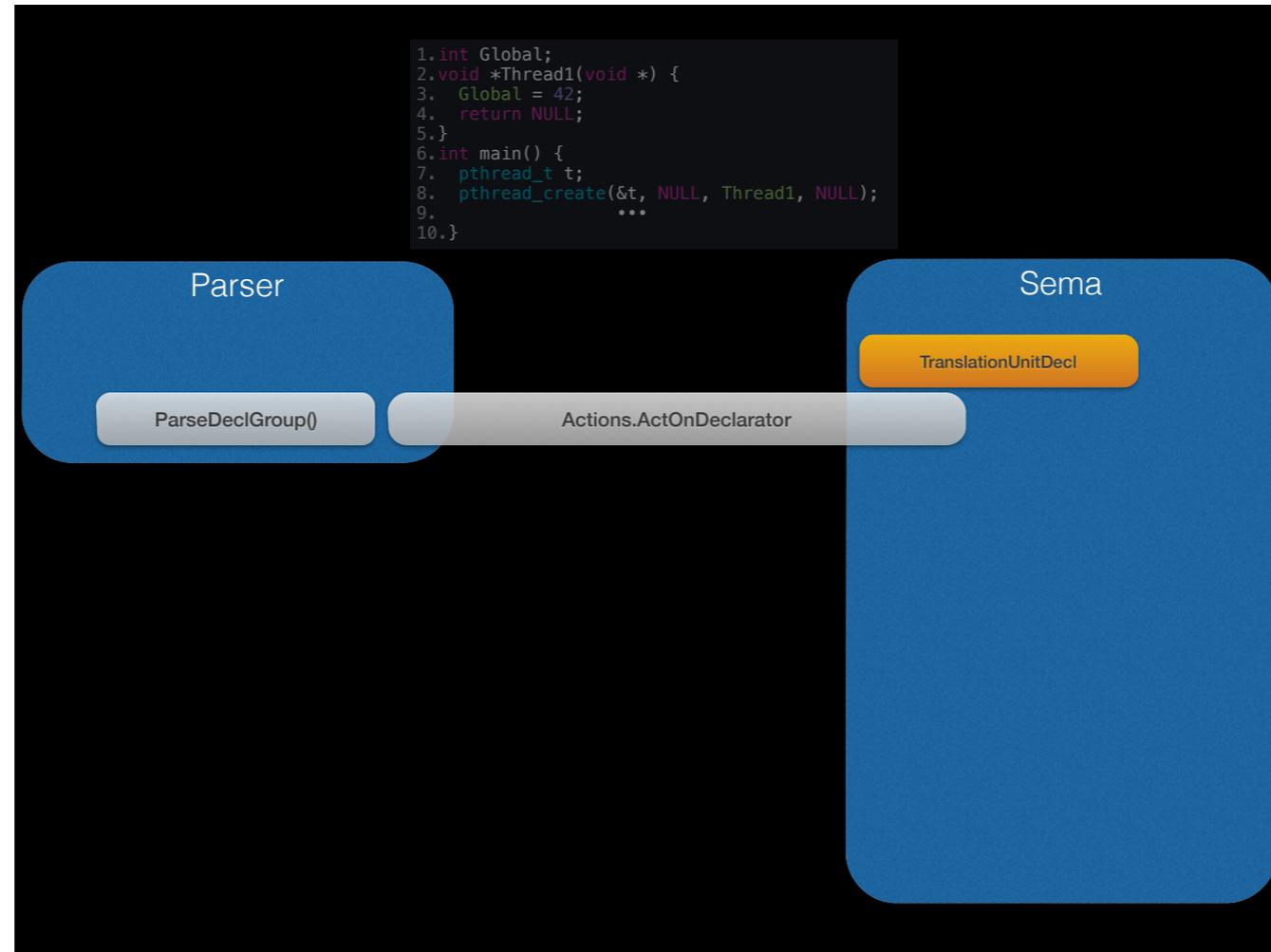
```
1.int Global;
2.void *Thread1(void *) {
3.  Global = 42;
4.  return NULL;
5.}
6.int main() {
7.  pthread_t t;
8.  pthread_create(&t, NULL, Thread1, NULL);
9.  ...
10.}
```



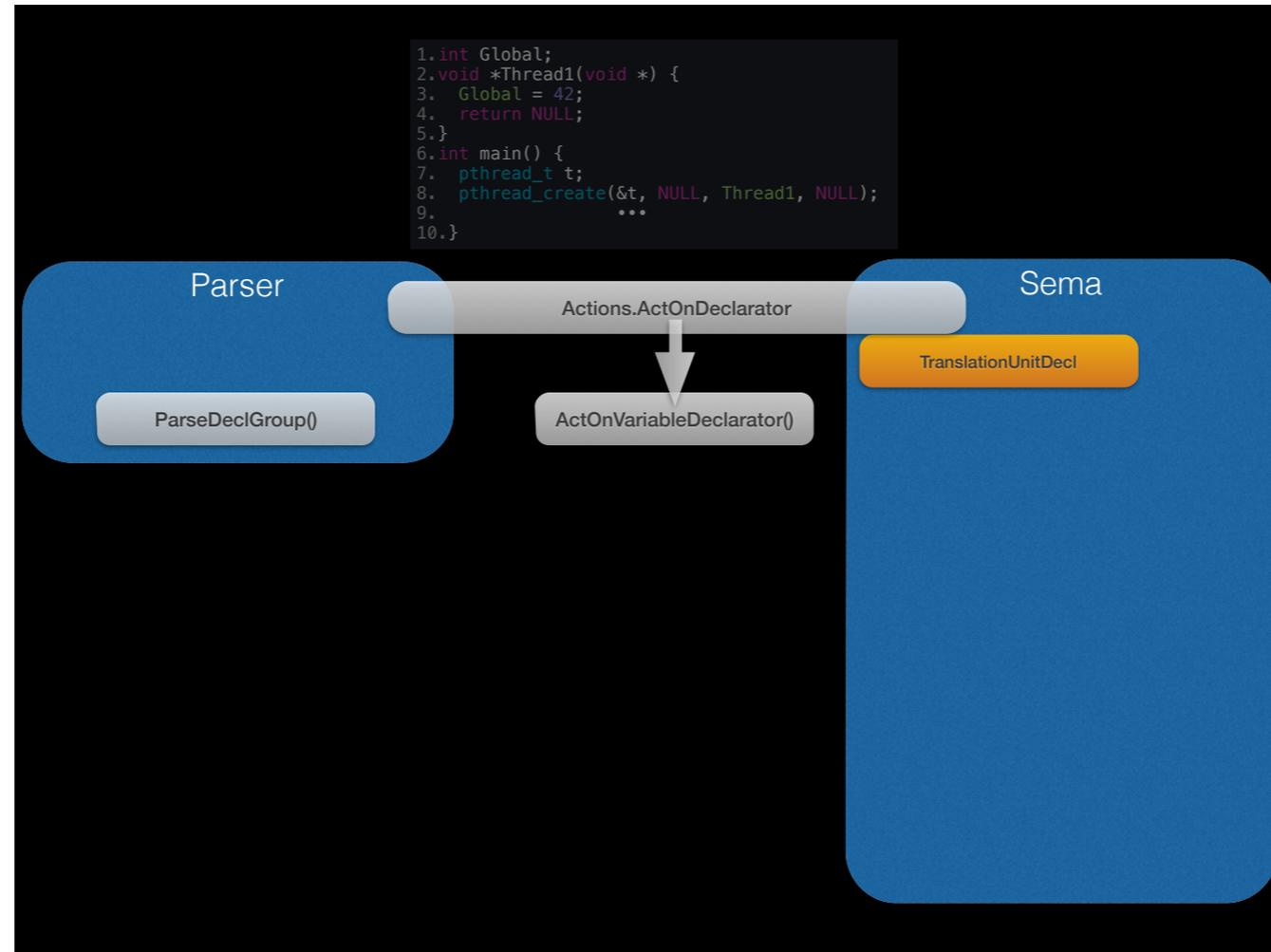
This is a skimmed summary of how clang creates an AST for given C or C++ code. It's imprecise, and skips a lot of steps, but should give the idea. When Sema starts, it creates `ASTContext`, which creates the top-level item -- `TranslationUnitDecl`. Parser takes `tok::kw_int`, and begins parsing a declaration group. There is no initializer, and there is no comma. It then finds a semi, and finishes group, which has only 1 declaration. Sema, then, takes this declaration, and puts it in AST. This process continues for other declarations and expressions.



This is a skimmed summary of how clang creates an AST for given C or C++ code. It's imprecise, and skips a lot of steps, but should give the idea. When Sema starts, it creates `ASTContext`, which creates the top-level item -- `TranslationUnitDecl`. Parser takes `tok::kw_int`, and begins parsing a declaration group. There is no initializer, and there is no comma. It then finds a semi, and finishes group, which has only 1 declaration. Sema, then, takes this declaration, and puts it in AST. This process continues for other declarations and expressions.

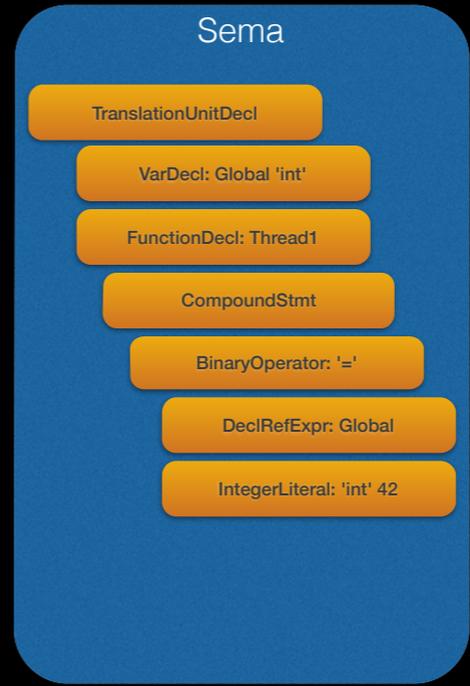


This is a skimmed summary of how clang creates an AST for given C or C++ code. It's imprecise, and skips a lot of steps, but should give the idea. When Sema starts, it creates `ASTContext`, which creates the top-level item -- `TranslationUnitDecl`. Parser takes `tok::kw_int`, and begins parsing a declaration group. There is no initializer, and there is no comma. It then finds a semi, and finishes group, which has only 1 declaration. Sema, then, takes this declaration, and puts it in AST. This process continues for other declarations and expressions.



This is a skimmed summary of how clang creates an AST for given C or C++ code. It's imprecise, and skips a lot of steps, but should give the idea. When Sema starts, it creates `ASTContext`, which creates the top-level item -- `TranslationUnitDecl`. Parser takes `tok::kw_int`, and begins parsing a declaration group. There is no initializer, and there is no comma. It then finds a semi, and finishes group, which has only 1 declaration. Sema, then, takes this declaration, and puts it in AST. This process continues for other declarations and expressions.

```
1.int Global;
2.void *Thread1(void *) {
3.  Global = 42;
4.  return NULL;
5.}
6.int main() {
7.  pthread_t t;
8.  pthread_create(&t, NULL, Thread1, NULL);
9.  ...
10.}
```



This is a skimmed summary of how clang creates an AST for given C or C++ code. It's imprecise, and skips a lot of steps, but should give the idea. When Sema starts, it creates ASTContext, which creates the top-level item -- TranslationUnitDecl. Parser takes tok::kw\_int, and begins parsing a declaration group. There is no initializer, and there is no comma. It then finds a semi, and finishes group, which has only 1 declaration. Sema, then, takes this declaration, and puts it in AST. This process continues for other declarations and expressions.

# Back-end



Backend takes a Module. Backend's function-level passes run on functions in the module. After that, module level passes run on module. Finally, module is sent for output generation -- could be object file, ll, cpp, or an assembly file.

# LLVM IR

- Intermediate Representation in SSA form, generally human-readable

```
%x = 42  
%0 = add i32 %x, 30
```

- Portable
- Can be encoded within bitcode
- Can be compiled to native assembly or object file

# Building blocks in LLVM

- Instructions
- Basic blocks
- Globals
- Functions
- Local variables
- Modules
- Passes

# Instructions

- Higher-level assembly
- Target-independent
- Represents an operation (control flow, arithmetic, memory, bitwise, etc.)

```
%0 = add i32 %x, 30
```

# Basic blocks

- Single-entry — single-exit instruction container
- Used inside function definitions

# Basic blocks

```
define i64 @gcd(i64 %x, i64 %y) {
entry:
  %e = alloca i64, align 8
  %0 = load i64*, %e
  %1 = icmp eq i64 %0, %y
  %2 = icmp eq i64 %x, 0
  br i1 %2, label %if.then, label %if.else

if.then:
  ret i64 %y

if.else:
  %3 = icmp slt i64 %x, %y
  br i1 %3, label %if.then1, label %if.cont

if.cont:
  %4 = call i64 @gcd(i64 %y, i64 %x)
  ret i64 %4
}
```

entry basic-block has single entry, and single exit. It branches to if.then or if.else. if.then exits with return, while if.else branches to if.then1 or if.cont. Most of the code is omitted.

# Globals

- Function declarations
- Other global variables, constants in C code (e.g. string literals), C++ vtables, and aliases

```
@llvm.global_ctors = appending global [1 x { i32, void  
()* }] [{ i32, void (* ) } { i32 0, void (* )  
@__tsan_init }]
```

```
declare void @__tsan_init()
```

# Functions

- Declared in the module
- Have attributes, visibility, and linkage
- Contains basic blocks

```
define i64 @gcd(i64 %x, i64 %y) {  
entry: ...
```

# Local variables

- Local to functions
- Allocated using `alloca` instruction generally at the entry to function

```
define i64 @gcd(i64 %x, i64 %y) {  
entry:  
    %e = alloca i64, align 8
```

# Modules

- Roughly corresponds to translation unit:
  - Type declarations
  - Function declarations/definitions
  - Global variables
- Have constructors and destructors
  - Constructors run during initialization (like constructor of a global instance of a C++ class)
    - Essential for sanitizers

# Module Constructor

```
1. static struct AGlobalClass {  
2.     int i;  
3.     AGlobalClass() : i(42) { }  
4. } global_instance;
```

```
1. __attribute__((constructor))  
2. static void a_module_ctor() {  
4. }
```

These would generate a module constructor. Constructor of AGlobalClass will be placed in module constructors. a\_module\_ctor function will be placed in module constructor, and will be called at the runtime. When they will execute is runtime dependent.

# Example LLVM Module

```
; ModuleID = 'example-module.c'
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.9.0"

%struct.__sFILE = type { ... }
@.str = private unnamed_addr constant [15 x i8] c"Out-of-memory\0A\00", align 1
@.str1 = private unnamed_addr constant [36 x i8] c"Ridiculous string length detected!\0A\00", align 1
; Function Attrs: nounwind ssp uwtable
define i32 @main(i32 %argc, i8** nocapture readonly %argv) #0 {
    %1 = icmp slt i32 %argc, 2          ; if (argc < 2)
    br i1 %1, label %2, label %3

; <label>:2                                ; exit(EXIT_FAILURE);
    tail call void @exit(i32 1) #6
    unreachable

; <label>:3                                ; preds = %0
    %4 = getelementptr inbounds i8** %argv, i64 1 ; size_t l = strlen(argv[1]);
    %5 = load i8** %4, align 8, !tbaa !1
    %6 = tail call i64 @strlen(i8* %5) #7
    %7 = icmp ugt i64 %6, 9223372036854775806 ; if (SSIZE_MAX - 1 < len)
    br i1 %7, label %8, label %11

; <label>:8                                ; fputs(kMsgRidiculousLength, stderr);
    %9 = load %struct.__sFILE** @__stderrp, align 8, !tbaa !1
    %10 = tail call i32 @"%01_fputs"(i8* getelementptr inbounds ([36 x i8]* @.str1, i64 0, i64 0), %struct.__sFILE*
%9) #7
    tail call void @exit(i32 1) #6
    unreachable
```

...

# Clang AST to LLVM Module

```
int Global;  
void *Thread1(void *);  
int main() {  
    pthread_t t;  
    pthread_create(&t, NULL, Thread1, NULL);  
    Global = 43;  
    pthread_join(t, NULL);  
    pthread_mutex_destroy(&g_mut);  
    return Global;  
}
```

VarDecl: Global 'int'

FunctionDecl: Thread1 'void \*(void \*)'

FunctionDecl: main 'int ()'

CompoundStmt

VarDecl: t 'pthread\_t'

CallExpr: 'int' 'pthread\_create'

BinaryOp: 'int' '='

DeclRefExpr: 'int' Global

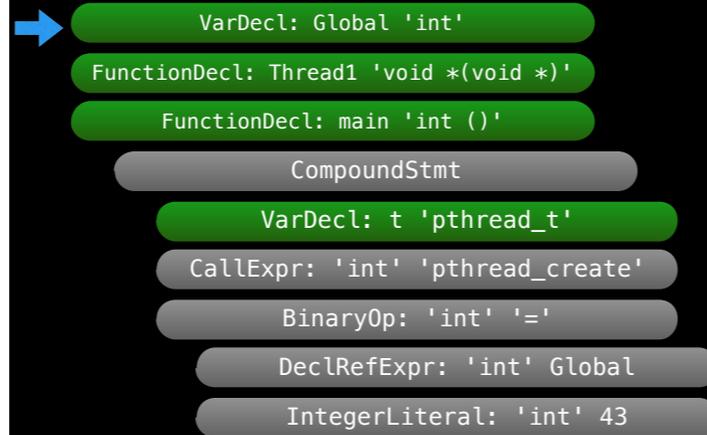
IntegerLiteral: 'int' 43

Thread1 is defined in another translation unit to save slide space. Given this code, AST will look something like this. This is an imprecise overview of how it looks. We will process this AST to generate an LLVM module.

# Clang AST to LLVM Module

- `CodeGenAction` creates `BackendConsumer`
- `BackendConsumer` traverses AST, descending from top-level declarations
- `BackendConsumer` finalizes `Module` construction

# Clang AST to LLVM Module



CodeGen will traverse AST from top-level declarations down, and create an LLVM module. Most of the code is omitted to fit in the slide. Also, this is human readable form, but normally these are represented as objects.

# Clang AST to LLVM Module



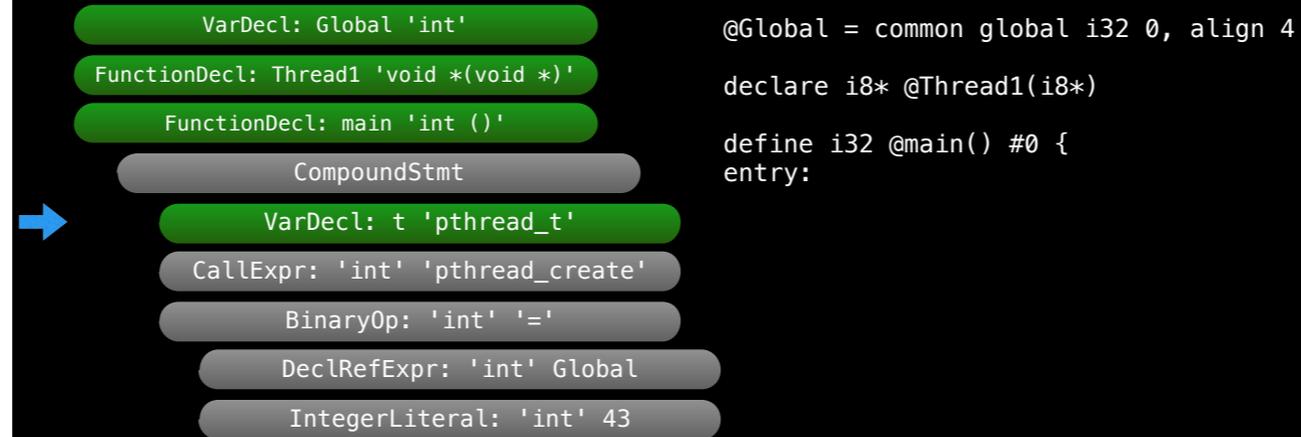
CodeGen will traverse AST from top-level declarations down, and create an LLVM module. Most of the code is omitted to fit in the slide. Also, this is human readable form, but normally these are represented as objects.

# Clang AST to LLVM Module



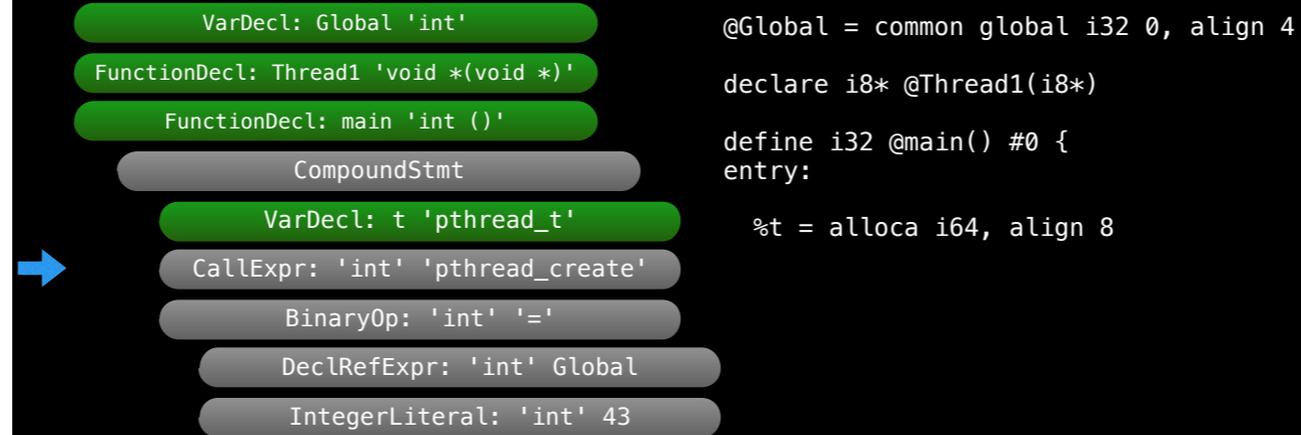
CodeGen will traverse AST from top-level declarations down, and create an LLVM module. Most of the code is omitted to fit in the slide. Also, this is human readable form, but normally these are represented as objects.

# Clang AST to LLVM Module



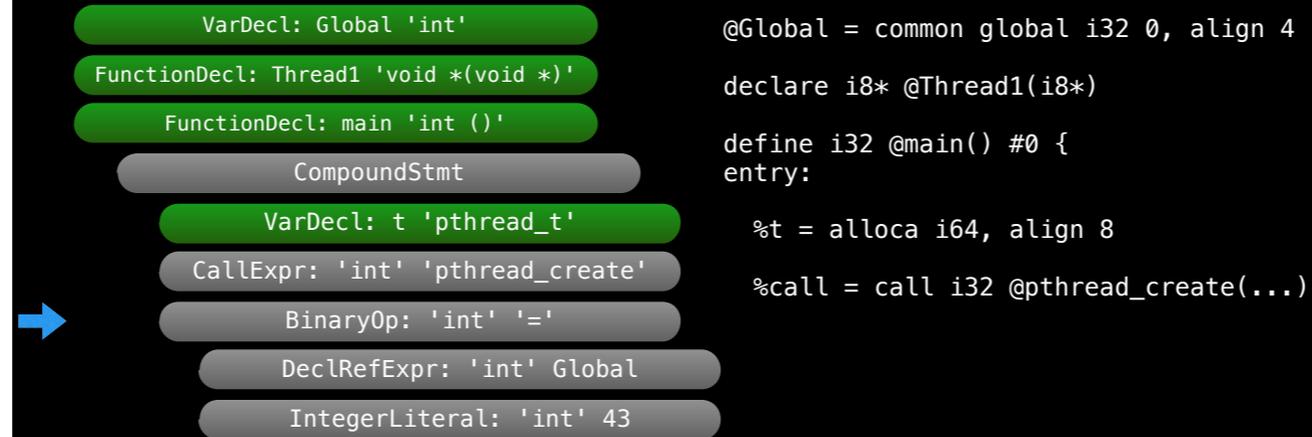
CodeGen will traverse AST from top-level declarations down, and create an LLVM module. Most of the code is omitted to fit in the slide. Also, this is human readable form, but normally these are represented as objects.

# Clang AST to LLVM Module



CodeGen will traverse AST from top-level declarations down, and create an LLVM module. Most of the code is omitted to fit in the slide. Also, this is human readable form, but normally these are represented as objects.

# Clang AST to LLVM Module



CodeGen will traverse AST from top-level declarations down, and create an LLVM module. Most of the code is omitted to fit in the slide. Also, this is human readable form, but normally these are represented as objects.

# Clang AST to LLVM Module



CodeGen will traverse AST from top-level declarations down, and create an LLVM module. Most of the code is omitted to fit in the slide. Also, this is human readable form, but normally these are represented as objects.

# LLVM Passes

- Traverses modules, and functions
- A traversal can:
  - Analyze code
  - Transform code
  - Generate graph, verify, etc.

# GCC vs Clang

- Clang AST contains non-truncated value as-written in the source
- GCC parser truncates overflowing constants, issues warning
- If user ignores overflow warnings, GCC's sanitizers will never see the problem
- GCC's UBSan is equally good as Clang's

Jonathan Wakely said that GCC know about these problems, and are working on a solution.

# Sanitizers

Sanitizers are meant to be used with tests, and generally not in production.



Many readers probably wrinkled their noses when the UN Food and Agriculture Organization recently recommended that eating more insects could help fight world hunger. But one Tokyo-based restaurant has reacted to the news with surprising swiftness. On May 29, an Italian-style café in Kinshicho called Absente will be launching a dish that promises to test the stomachs of even the hardest of diners: locust spaghetti.

Turns out, there is a dish of spaghetti with bugs! Our delicious spaghetti has been invaded by bugs. We can't get rid of them one by one, and we desperately need to deliver this spaghetti to customer! No worries! Our hero bird, no it's not a bird... It's a bug-eating-expert-dragon just came to save the day! He attacks bugs with his might, and burns them down. At the end, we end up with bug-free, completely sanitized -- after applying enough dragon flame -- spaghetti that we can ship from kitchen to table.



Turns out, there is a dish of spaghetti with bugs! Our delicious spaghetti has been invaded by bugs. We can't get rid of them one by one, and we desperately need to deliver this spaghetti to customer! No worries! Our hero bird, no it's not a bird... It's a bug-eating-expert-dragon just came to save the day! He attacks bugs with his might, and burns them down. At the end, we end up with bug-free, completely sanitized -- after applying enough dragon flame -- spaghetti that we can ship from kitchen to table.



Turns out, there is a dish of spaghetti with bugs! Our delicious spaghetti has been invaded by bugs. We can't get rid of them one by one, and we desperately need to deliver this spaghetti to customer! No worries! Our hero bird, no it's not a bird... It's a bug-eating-expert-dragon just came to save the day! He attacks bugs with his might, and burns them down. At the end, we end up with bug-free, completely sanitized -- after applying enough dragon flame -- spaghetti that we can ship from kitchen to table.



Turns out, there is a dish of spaghetti with bugs! Our delicious spaghetti has been invaded by bugs. We can't get rid of them one by one, and we desperately need to deliver this spaghetti to customer! No worries! Our hero bird, no it's not a bird... It's a bug-eating-expert-dragon just came to save the day! He attacks bugs with his might, and burns them down. At the end, we end up with bug-free, completely sanitized -- after applying enough dragon flame -- spaghetti that we can ship from kitchen to table.



Turns out, there is a dish of spaghetti with bugs! Our delicious spaghetti has been invaded by bugs. We can't get rid of them one by one, and we desperately need to deliver this spaghetti to customer! No worries! Our hero bird, no it's not a bird... It's a bug-eating-expert-dragon just came to save the day! He attacks bugs with his might, and burns them down. At the end, we end up with bug-free, completely sanitized -- after applying enough dragon flame -- spaghetti that we can ship from kitchen to table.

# UBSan

- Group of smaller sanitizers
- Operates on AST
- Can work with partially-instrumented code
- Detects broad range of undefined-behavior cases
- Can halt program, print stack trace
- `-fsanitize=undefined-trap -fsanitize=undefined-trap-on-error`  
doesn't emit calls to run-time library

UBSan is slightly different than other sanitizers. It's a group of smaller sanitizers, and it operates on AST. That is, it knows about the undefined behavior in language. There are differences between C and C++ in terms of undefined behavior, where an expression in C might be unspecified, but undefined in C++. UBSan understands these differences as well. UBSan doesn't need to instrument all code, although it's recommended, since undefined behavior occurred before instrumented code might have left the program in an incorrect state. As far as I remember, UBSan has gained many checks during C++11 constexpr work; constant expression context should not have undefined behavior. UBSan doesn't need a runtime; it can emit a trap intrinsic instead of calling runtime function to print a report. This is helpful, especially if you want to run UBSan on an embedded platform, where UBSan runtime isn't ported today. This basically removes dependency to any runtime library, and compiler output can be used directly.

# UBSan - Individual Sanitizers

Alignment	Alignment violations
ArrayBounds	Array bound violations
Bool	Invalid `bool` values (e.g. 194)
Enum	Invalid values for type
FloatCastOverflow	Invalid conversions to/from float
FloatDivideByZero	Float division by 0.0f

\* Is not member of undefined-trap group, and requires runtime library support

Red cells mark non-recoverable kind; execution has to be stopped. As footnote shows, some of them require UBSan runtime library support.

# UBSan - Individual Sanitizers

Function *	Function type mismatch
IntegerDivideByZero	Integer division by 0
NonnullAttribute	Violations of nonnull attribute
Null	Empty/null glvalue
ObjectSize	Storage is not big enough
Return	Missing return statement

\* Is not member of undefined-trap group, and requires runtime library support

Red cells mark non-recoverable kind; execution has to be stopped. As footnote shows, some of them require UBSan runtime library support.

# UBSan - Individual Sanitizers

ReturnsNonnullAttribute	Violation of returns_nonnull attribute
Shift	Shift overflows
SignedIntegerOverflow	Arithmetic operations that might overflow
Unreachable	Executing __builtin_unreachable
VLABound	Variable-length-array bound is >0
Vptr *	Whether vptr is pointing to correct type

\* Is not member of undefined-trap group, and requires runtime library support

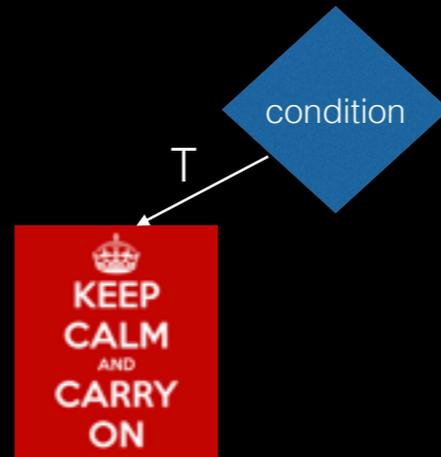
Red cells mark non-recoverable kind; execution has to be stopped. As footnote shows, some of them require UBSan runtime library support.

# UBSan - EmitCheck



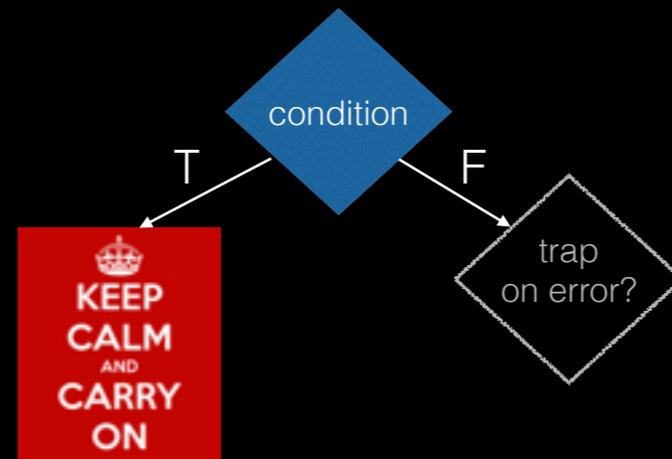
UBSan is all about EmitCheck. UBSan wants to check whether a condition has met or not. To do that, it emits a code that checks the values at runtime. For example, whether  $X > y$ . It then emits a branch instruction. Based on the outcome of this check, it takes either true or false branch. If check result is true, execution continues as written in the program source (example in next slide). For the false branch, how failure should be handled depends on front-end flags. 'trap on error' check is an 'if' statement in front-end code itself; it's not emitted. This is where front-end decides as to whether it should emit a trap intrinsic or call to UBSan runtime library. Outcome of this check is emitted as the false branch of condition check.

# UBSan - EmitCheck



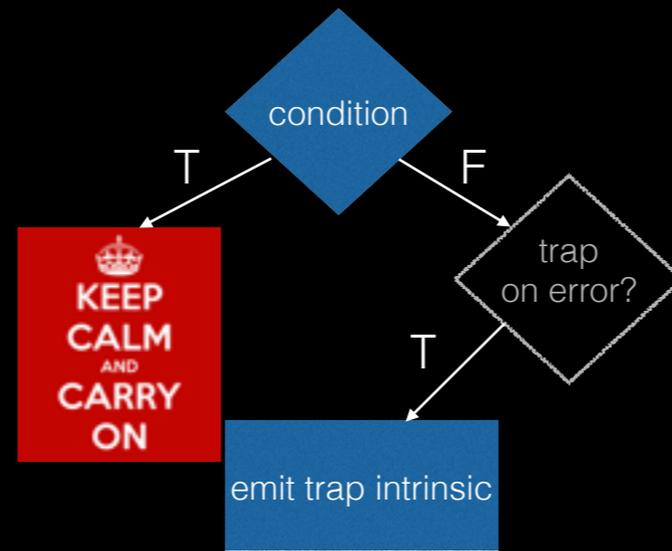
UBSan is all about EmitCheck. UBSan wants to check whether a condition has met or not. To do that, it emits a code that checks the values at runtime. For example, whether  $X$  is  $> y$ . It then emits a branch instruction. Based on the outcome of this check, it takes either true or false branch. If check result is true, execution continues as written in the program source (example in next slide). For the false branch, how failure should be handled depends on front-end flags. 'trap on error' check is an 'if' statement in front-end code itself; it's not emitted. This is where front-end decides as to whether it should emit a trap intrinsic or call to UBSan runtime library. Outcome of this check is emitted as the false branch of condition check.

# UBSan - EmitCheck



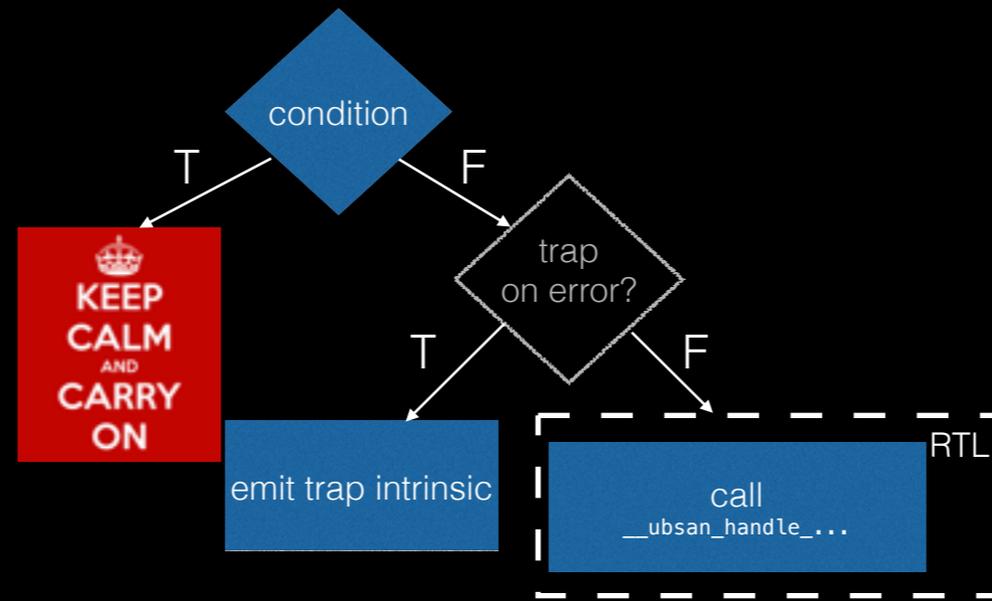
UBSan is all about EmitCheck. UBSan wants to check whether a condition has met or not. To do that, it emits a code that checks the values at runtime. For example, whether  $X > y$ . It then emits a branch instruction. Based on the outcome of this check, it takes either true or false branch. If check result is true, execution continues as written in the program source (example in next slide). For the false branch, how failure should be handled depends on front-end flags. 'trap on error' check is an 'if' statement in front-end code itself; it's not emitted. This is where front-end decides as to whether it should emit a trap intrinsic or call to UBSan runtime library. Outcome of this check is emitted as the false branch of condition check.

# UBSan - EmitCheck



UBSan is all about EmitCheck. UBSan wants to check whether a condition has met or not. To do that, it emits a code that checks the values at runtime. For example, whether  $X > y$ . It then emits a branch instruction. Based on the outcome of this check, it takes either true or false branch. If check result is true, execution continues as written in the program source (example in next slide). For the false branch, how failure should be handled depends on front-end flags. 'trap on error' check is an 'if' statement in front-end code itself; it's not emitted. This is where front-end decides as to whether it should emit a trap intrinsic or call to UBSan runtime library. Outcome of this check is emitted as the false branch of condition check.

# UBSan - EmitCheck



UBSan is all about EmitCheck. UBSan wants to check whether a condition has met or not. To do that, it emits a code that checks the values at runtime. For example, whether  $X > y$ . It then emits a branch instruction. Based on the outcome of this check, it takes either true or false branch. If check result is true, execution continues as written in the program source (example in next slide). For the false branch, how failure should be handled depends on front-end flags. 'trap on error' check is an 'if' statement in front-end code itself; it's not emitted. This is where front-end decides as to whether it should emit a trap intrinsic or call to UBSan runtime library. Outcome of this check is emitted as the false branch of condition check.

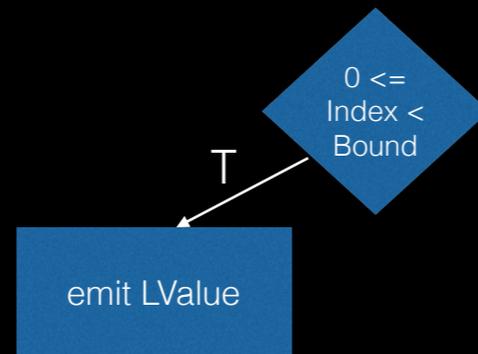
# UBSan - ArrayBounds



$0 \leq$   
Index <  
Bound

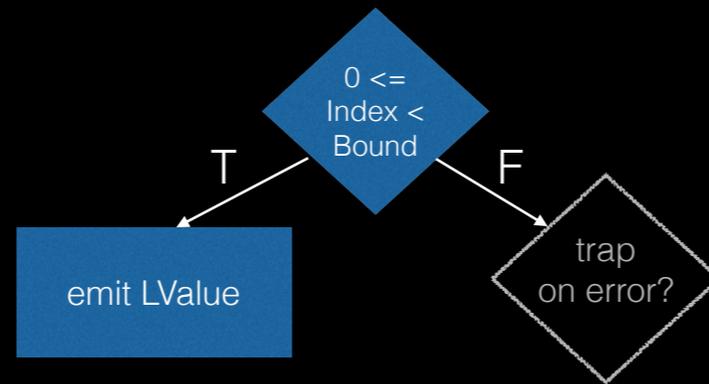
This is a more concrete example of EmitCheck. When CodeGen encounters with array subscript expression -- and when code is instrumented with `ubsan --` it emits a check; "is the index value greater than or equal to zero, and is it less than the array bound". Bound is known at compile time. Index is a runtime value. For true branch of this check, `ubsan` emits whatever it was supposed to emit. It has to decide what should be emitted for the false branch. It checks presence of 'trap on error' frontend flag. If so, it decides to emit a trap intrinsic. If 'trap on error' is not specified, which is default, it emits call to `ubsan` runtime library's `__ubsan_handle_out_of_bounds` function with some information about the location in source. This metadata is also emitted inside the LLVM module.

# UBSan - ArrayBounds



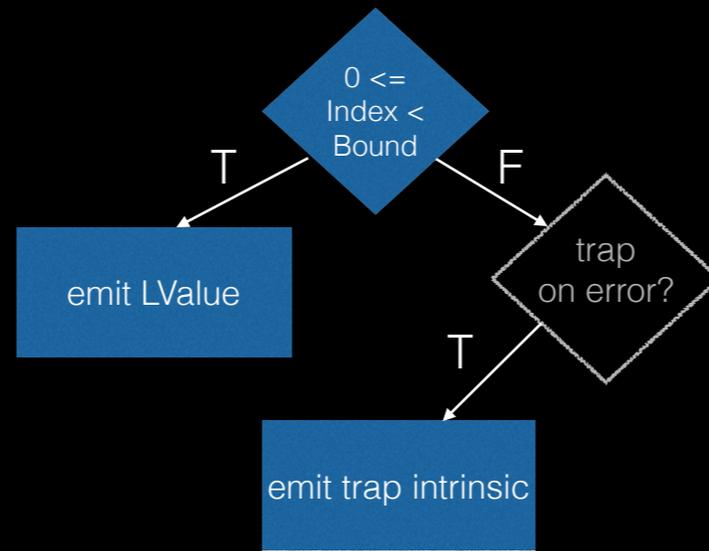
This is a more concrete example of EmitCheck. When CodeGen encounters with array subscript expression -- and when code is instrumented with `ubsan --` it emits a check; "is the index value greater than or equal to zero, and is it less than the array bound". Bound is known at compile time. Index is a runtime value. For true branch of this check, `ubsan` emits whatever it was supposed to emit. It has to decide what should be emitted for the false branch. It checks presence of 'trap on error' frontend flag. If so, it decides to emit a trap intrinsic. If 'trap on error' is not specified, which is default, it emits call to `ubsan` runtime library's `__ubsan_handle_out_of_bounds` function with some information about the location in source. This metadata is also emitted inside the LLVM module.

# UBSan - ArrayBounds



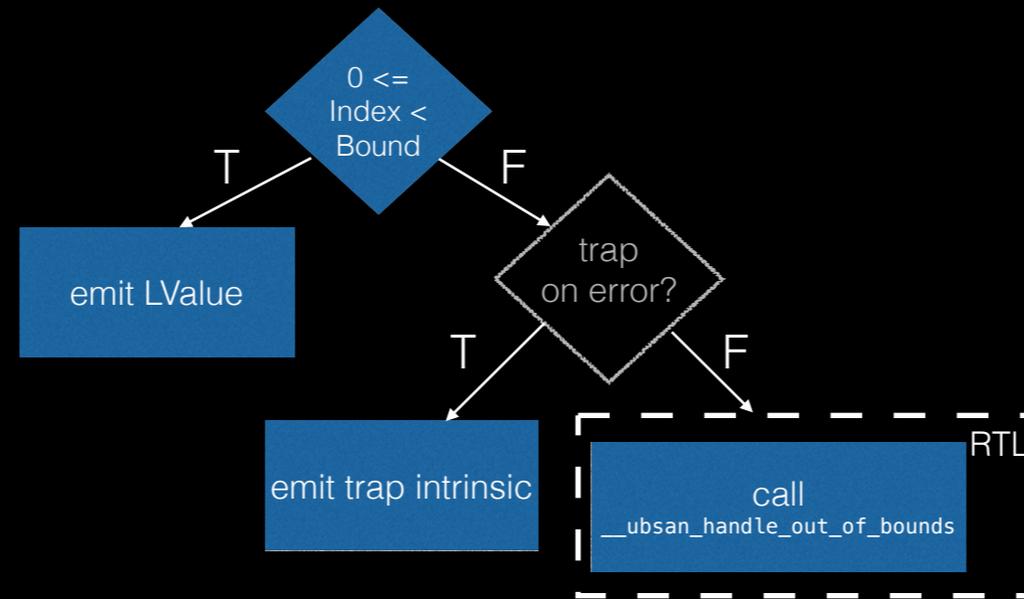
This is a more concrete example of EmitCheck. When CodeGen encounters with array subscript expression -- and when code is instrumented with `ubsan --` it emits a check; "is the index value greater than or equal to zero, and is it less than the array bound". Bound is known at compile time. Index is a runtime value. For true branch of this check, `ubsan` emits whatever it was supposed to emit. It has to decide what should be emitted for the false branch. It checks presence of 'trap on error' frontend flag. If so, it decides to emit a trap intrinsic. If 'trap on error' is not specified, which is default, it emits call to `ubsan` runtime library's `__ubsan_handle_out_of_bounds` function with some information about the location in source. This metadata is also emitted inside the LLVM module.

# UBSan - ArrayBounds



This is a more concrete example of EmitCheck. When CodeGen encounters with array subscript expression -- and when code is instrumented with `ubsan --` it emits a check; "is the index value greater than or equal to zero, and is it less than the array bound". Bound is known at compile time. Index is a runtime value. For true branch of this check, `ubsan` emits whatever it was supposed to emit. It has to decide what should be emitted for the false branch. It checks presence of 'trap on error' frontend flag. If so, it decides to emit a trap intrinsic. If 'trap on error' is not specified, which is default, it emits call to `ubsan` runtime library's `__ubsan_handle_out_of_bounds` function with some information about the location in source. This metadata is also emitted inside the LLVM module.

# UBSan - ArrayBounds



This is a more concrete example of EmitCheck. When CodeGen encounters with array subscript expression -- and when code is instrumented with `ubsan --` it emits a check; "is the index value greater than or equal to zero, and is it less than the array bound". Bound is known at compile time. Index is a runtime value. For true branch of this check, `ubsan` emits whatever it was supposed to emit. It has to decide what should be emitted for the false branch. It checks presence of 'trap on error' frontend flag. If so, it decides to emit a trap intrinsic. If 'trap on error' is not specified, which is default, it emits call to `ubsan` runtime library's `__ubsan_handle_out_of_bounds` function with some information about the location in source. This metadata is also emitted inside the LLVM module.

# UBSan - How does it know?

- It doesn't; someone has to tell it to Clang
- UBSan gained most checks with C++11 constexpr work

AFAIK, UBSan gained most of the checks with C++11 constexpr work, where all/most of the ub cases had to be identified, because constant expressions cannot have undefined behavior.

# UBSan - Example

```
1. float estimate = float();
2. float actual_sequence() { return 7.7; }
3. float calculate_seq_ratio() {
4.     return actual_sequence() / estimate;
5. }
6. int main() {
7.     (void)calculate_seq_ratio();
8.     return 0;
9. }
```

float-div-zero.cpp:4:28: runtime error: division by zero

I won't be showing trophies of sanitizers. This one resembles one of the problems in our code. GUI is responsible for displaying a progress bar at startup. We try to calculate progress ratio before we know enough about our estimates. This ends up dividing a value to 0.0f.

# Instrumentation Workflow

1. Insert an init function to module constructor
2. Instrument IR
  1. Insert inline checks, unless otherwise specified
    1. Emit calls to report functions in run-time library
  2. Emit calls to run-time checks
    1. Call error report functions

Sanitizers -- very briefly -- follow this sequence. A function is inserted to module constructor. Then, sanitizer is called as a function pass. In this pass, it instruments functions. Functions contain basic blocks, and basic blocks contain instructions. Sanitizers visit certain instructions that they find interesting. MSan, for example, is interested almost all instructions, while ASan is more interested in loads, stores. Instrumentation will require checking things, generally, not always, before they happen. These checks can be inserted inline, right before the instruction, or can be inserted as calls to runtime library functions. Inline checks can be faster, but might make code more complicated. Whether checks are implemented in runtime library or inline, except ubsan, reporting is always done by the runtime library.

# Run-time Library Workflow

1. Parse flags (XSAN\_OPTIONS environment variable)
2. Map memory regions
3. Intercept functions
4. Perform checks and/or print reports
  1. Symbolicate frames, and data

I keep saying module constructor and inserting a function there... That is sanitizer's init function. Runtime library implements that init function. Except ubsan, all sanitizers have an init function that setup the environment. Runtime's init function will be called multiple times -- for each module constructor, that is, it will be called at least once for each translation unit in the executable. Generally, sanitizers have a static variable that tells whether it has been initialized before. If not, it's set to 1 or true indicating that the runtime library is ready to operate. Next, sanitizers parse the environment variable or calls the function that return default options. They then do their memory mappings. Next, they intercept functions in C standard library or pthreads. This is the initialization phase. After this point, generally, sanitizer is ready to serve either by only reporting errors or by performing checks and reporting errors, when they occur.

ASan

# ASan

- Dedicated LLVM pass
- Diagnoses addressability issues:
  - Heap
  - Stack
  - Adjacent objects
- Checks can either be emitted directly or as function calls
- Static initialization order, ODR violation, etc.

AddressSanitizer has its own pass. It diagnoses addressability issues on heap, stack, adjacent objects. Until a certain threshold, IIRC default is 7000 instructions, checks will be emitted inline. If module has more than 7000 instructions, ASan will emit calls to checker functions in ASan RTL. It also detects static initialization order, ODR violations.

# ASan instrumentation

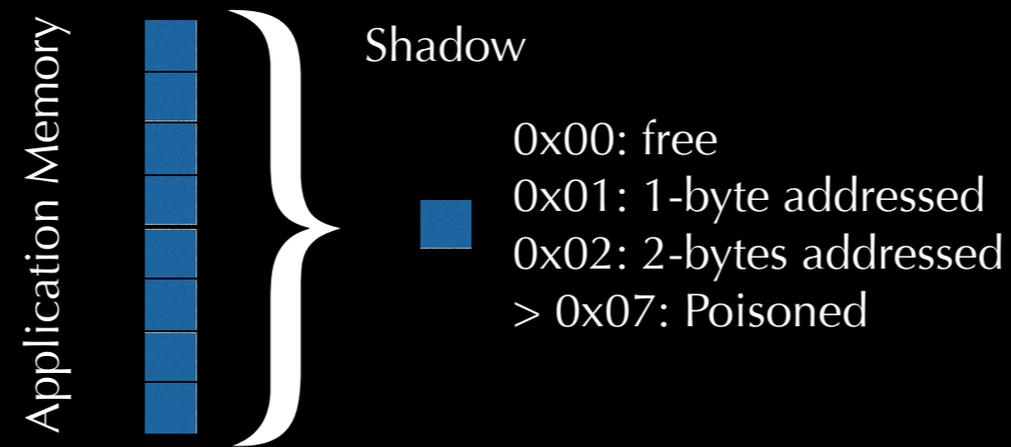
- Shadows variables at runtime
- Interposes allocation/deallocation, copying, and searching functions of C library
- Inserts red-zones around objects
- Poisons red-zones and shadow
- Access to poisoned memory is failure

ASan shadows variables at runtime. That is, it has a smaller representation of memory locations somewhere else in the memory. That somewhere else is called shadow region. ASan RTL intercepts some interesting C standard library functions at runtime. During instrumentation, red-zones will be inserted around objects. Poisoned means not addressable.

# ASan instrumentation steps

1. Declare `loadN/storeN` error callbacks
2. Find interesting `load`, `store`, and function calls
3. Find interesting pointer subtraction/comparison
4. Poison stack
5. Call `__asan_init` from module ctors

# ASan - Shadow & Poison



8-bytes of application memory is represented by 1-byte shadow memory. This byte can have any value between 0 and 255. 0 means entire 8 bytes are addressable. 1 means only 1 byte of this 8 byte block is addressable. Anything greater than 7 is used to represent type of redzone, such as heap-left-red-zone or intra-object-red-zone.

# ASan - Real (!) example

```
#include <stdint>
#include <stdlib>
#include <iostream>

static constexpr unsigned kWidth = 42;
static constexpr unsigned kHeight = 42;
static constexpr unsigned kBpp = 4;
static constexpr unsigned kStride = kWidth * kBpp;
static constexpr size_t kImageSize = kStride * kHeight;

static void do_imageop(uint8_t *dst, const uint8_t *src) {
    for (auto x = 0; x < kWidth; ++x) {
        *dst++ = *src++;
        *dst++ = *src++;
        *dst++ = *src++;
        *dst++ = *src++;
    }
}

int main(int argc, char *argv[]) {
    uint8_t *image1 = (uint8_t*)malloc(kImageSize);
    uint8_t *image2 = (uint8_t*)malloc(kImageSize);
    memset(image1, 0, kImageSize);
    memset(image2, 0, kImageSize);
    uint8_t *src = image1, *dst = image2;

    for (auto line = 0; line < kHeight; ++line,
         src += kStride,
         dst += kStride) {
        do_imageop(dst, src);
    }
    free(image1);
    free(image2);
    return 0;
}
```

One of my colleagues were writing a colorspace converter, IIRC, in C. Like any regular programmer, he began decomposing the problem, and wrote a separate function to this operation for single scanline of image. He then decided this is too boring, and like a boss, he cut the function body, and pasted over the call to function. Program crashed. We took a look at the code, and walked through a bit -- without debugging. Everything looked perfectly sensible. It compiled, not just that, it also linked -- so it must run! We've tried to debug shortly, but then figured something's happening to the pointers. I asked him to sanitize it, and call me back after that. Turns out, there was something wrong. He incremented pointer both in the inner for loop and in the outer one. This was in a function, and changes on pointers dst or src were not visible from outside of the function. When he moved this code inside the loop, names clashed, and pointers went out of bounds. Because pointers went way too out of bounds in his case, he was able to see it really quick. If this was a smaller read or write, it might have slipped.

# ASan - Real (!) example

```
#include <stdint>
#include <stdlib>
#include <iostream>

static constexpr unsigned kWidth = 42;
static constexpr unsigned kHeight = 42;
static constexpr unsigned kBpp = 4;
static constexpr unsigned kStride = kWidth * kBpp;
static constexpr size_t kImageSize = kStride * kHeight;

int main(int argc, char *argv[]) {
    uint8_t *image1 = (uint8_t*)malloc(kImageSize);
    uint8_t *image2 = (uint8_t*)malloc(kImageSize);
    memset(image1, 0, kImageSize);
    memset(image2, 0, kImageSize);
    uint8_t *src = image1, *dst = image2;

    for (auto line = 0; line < kHeight; ++line,
         src += kStride,
         dst += kStride) {
        for (auto x = 0; x < kWidth; ++x) {
            *dst++ = *src++;
            *dst++ = *src++;
            *dst++ = *src++;
            *dst++ = *src++;
        }
    }
    free(image1);
    free(image2);
    return 0;
}
```

One of my colleagues were writing a colorspace converter, IIRC, in C. Like any regular programmer, he began decomposing the problem, and wrote a separate function to this operation for single scanline of image. He then decided this is too boring, and like a boss, he cut the function body, and pasted over the call to function. Program crashed. We took a look at the code, and walked through a bit -- without debugging. Everything looked perfectly sensible. It compiled, not just that, it also linked -- so it must run! We've tried to debug shortly, but then figured something's happening to the pointers. I asked him to sanitize it, and call me back after that. Turns out, there was something wrong. He incremented pointer both in the inner for loop and in the outer one. This was in a function, and changes on pointers `dst` or `src` were not visible from outside of the function. When he moved this code inside the loop, names clashed, and pointers went out of bounds. Because pointers went way too out of bounds in his case, he was able to see it really quick. If this was a smaller read or write, it might have slipped.

# ASan - Real (!) example

```
#include <stdint>
#include <stdlib>
#include <iostream>

static constexpr unsigned kWidth = 42;
static constexpr unsigned kHeight = 42;
static constexpr unsigned kBpp = 4;
static constexpr unsigned kStride = kWidth * kBpp;
static constexpr size_t kImageSize = kStride * kHeight;

int main(int argc, char *argv[]) {
    uint8_t *image1 = (uint8_t*)malloc(kImageSize);
    uint8_t *image2 = (uint8_t*)malloc(kImageSize);
    memset(image1, 0, kImageSize);
    memset(image2, 0, kImageSize);
    uint8_t *src = image1, *dst = image2;

    for (auto line = 0; line < kHeight; ++line,
         src += kStride,
         dst += kStride) {
        for (auto x = 0; x < kWidth; ++x) {
            *dst++ = *src++;
            *dst++ = *src++;
            *dst++ = *src++;
            *dst++ = *src++;
        }
    }
    free(image1);
    free(image2);
    return 0;
}

% ./convert-images image1 image2
=====
==9270==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x6240000dc90 at pc 0x00010ec53be4 bp 0x7fff50fac650
sp 0x7fff50fac630
READ of size 1 at 0x6240000dc90 thread T0
```

One of my colleagues were writing a colorspace converter, IIRC, in C. Like any regular programmer, he began decomposing the problem, and wrote a separate function to this operation for single scanline of image. He then decided this is too boring, and like a boss, he cut the function body, and pasted over the call to function. Program crashed. We took a look at the code, and walked through a bit -- without debugging. Everything looked perfectly sensible. It compiled, not just that, it also linked -- so it must run! We've tried to debug shortly, but then figured something's happening to the pointers. I asked him to sanitize it, and call me back after that. Turns out, there was something wrong. He incremented pointer both in the inner for loop and in the outer one. This was in a function, and changes on pointers `dst` or `src` were not visible from outside of the function. When he moved this code inside the loop, names clashed, and pointers went out of bounds. Because pointers went way too out of bounds in his case, he was able to see it really quick. If this was a smaller read or write, it might have slipped.

# ASan – Step-By-Step

Step-by-step execution of instrumented code. Ch is chunk data, some metadata associated with the location. Pointers go out of bounds. ASan checks the shadow value. It figures that only 4 bytes of this 8 byte region is addressable. See `__asan_load1`.

# ASan – Step-By-Step

```
→ uint8_t *image1 = (uint8_t*)malloc(kImageSize);
   uint8_t *image2 = (uint8_t*)malloc(kImageSize);

   uint8_t *src = image1, *dst = image2;

   for (auto line = 0; line < kHeight; ++line,
        src += kStride, dst += kStride) {
       *dst++ = *src++;
       /* __asan_load1(src);
        tmp = *src;
        ++src;
        __asan_store1(dst);
        *dst = tmp;
        ++dst; */
       *dst++ = *src++;
       *dst++ = *src++;
       *dst++ = *src++;
   }
```

App heap

Shadow memory

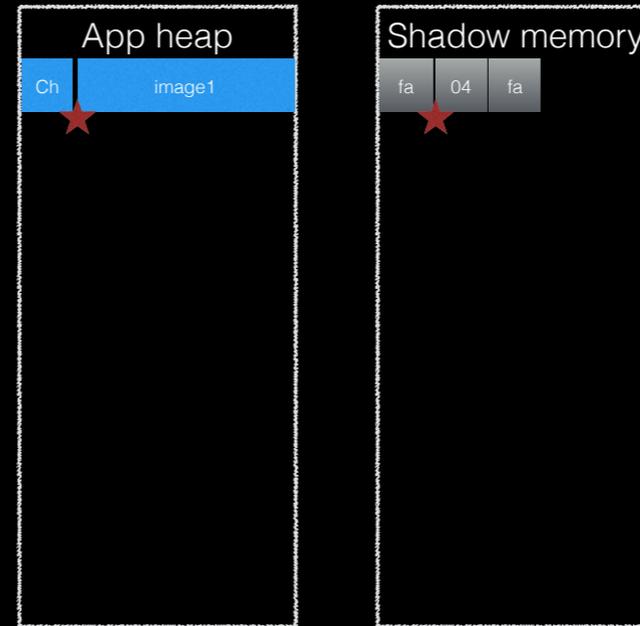
Step-by-step execution of instrumented code. Ch is chunk data, some metadata associated with the location. Pointers go out of bounds. ASan checks the shadow value. It figures that only 4 bytes of this 8 byte region is addressable. See `__asan_load1`.

# ASan – Step-By-Step

```
uint8_t *image1 = (uint8_t*)malloc(kImageSize);
uint8_t *image2 = (uint8_t*)malloc(kImageSize);

uint8_t *src = image1, *dst = image2;

for (auto line = 0; line < kHeight; ++line,
     src += kStride, dst += kStride) {
    *dst++ = *src++;
    /* __asan_load1(src);
     tmp = *src;
     ++src;
     __asan_store1(dst);
     *dst = tmp;
     ++dst; */
    *dst++ = *src++;
    *dst++ = *src++;
    *dst++ = *src++;
}
```



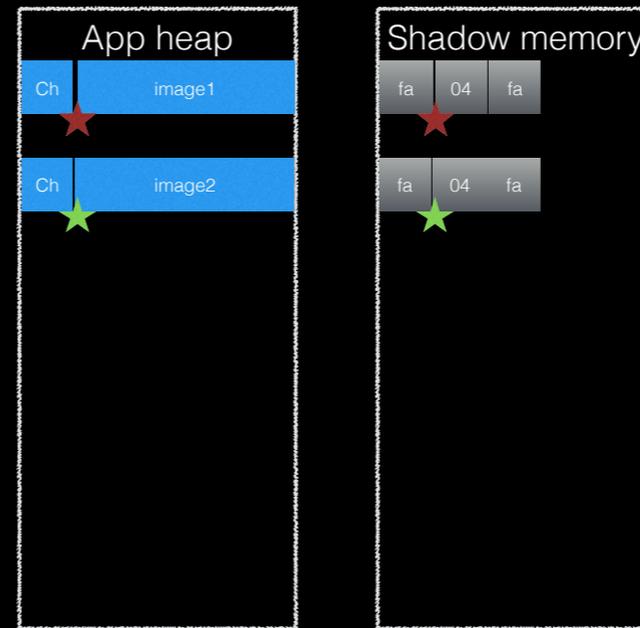
Step-by-step execution of instrumented code. Ch is chunk data, some metadata associated with the location. Pointers go out of bounds. ASan checks the shadow value. It figures that only 4 bytes of this 8 byte region is addressable. See `__asan_load1`.

# ASan – Step-By-Step

```
uint8_t *image1 = (uint8_t*)malloc(kImageSize);
uint8_t *image2 = (uint8_t*)malloc(kImageSize);

uint8_t *src = image1, *dst = image2;

for (auto line = 0; line < kHeight; ++line,
     src += kStride, dst += kStride) {
  *dst++ = *src++;
  /* __asan_load1(src);
   * tmp = *src;
   * ++src;
   * __asan_store1(dst);
   * *dst = tmp;
   * ++dst; */
  *dst++ = *src++;
  *dst++ = *src++;
  *dst++ = *src++;
}
```



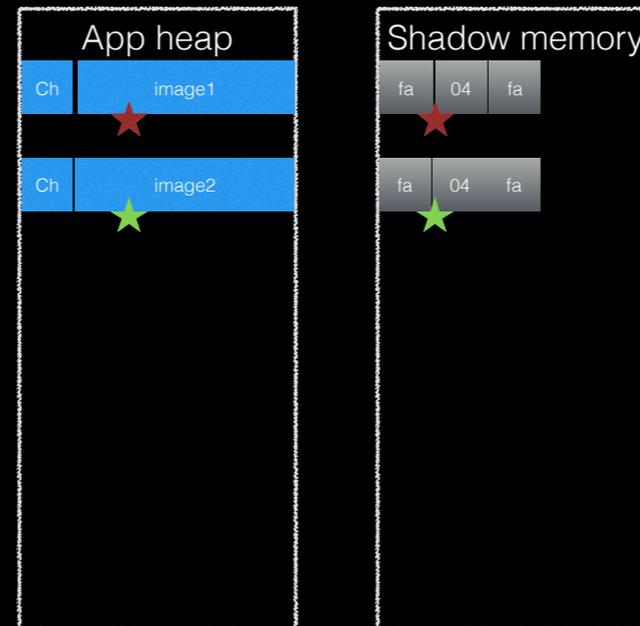
Step-by-step execution of instrumented code. Ch is chunk data, some metadata associated with the location. Pointers go out of bounds. ASan checks the shadow value. It figures that only 4 bytes of this 8 byte region is addressable. See `__asan_load1`.

# ASan – Step-By-Step

```
uint8_t *image1 = (uint8_t*)malloc(kImageSize);
uint8_t *image2 = (uint8_t*)malloc(kImageSize);

uint8_t *src = image1, *dst = image2;

for (auto line = 0; line < kHeight; ++line,
     src += kStride, dst += kStride) {
    *dst++ = *src++;
    /* __asan_load1(src);
     * tmp = *src;
     * ++src;
     * __asan_store1(dst);
     * *dst = tmp;
     * ++dst; */
    *dst++ = *src++;
    *dst++ = *src++;
    *dst++ = *src++;
}
```



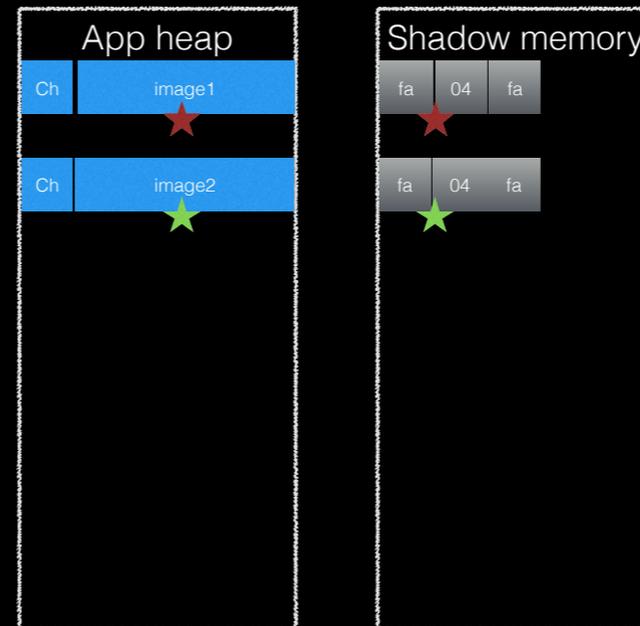
Step-by-step execution of instrumented code. Ch is chunk data, some metadata associated with the location. Pointers go out of bounds. ASan checks the shadow value. It figures that only 4 bytes of this 8 byte region is addressable. See `__asan_load1`.

# ASan – Step-By-Step

```
uint8_t *image1 = (uint8_t*)malloc(kImageSize);
uint8_t *image2 = (uint8_t*)malloc(kImageSize);

uint8_t *src = image1, *dst = image2;

for (auto line = 0; line < kHeight; ++line,
     src += kStride, dst += kStride) {
    *dst++ = *src++;
    /* __asan_load1(src);
     * tmp = *src;
     * ++src;
     * __asan_store1(dst);
     * *dst = tmp;
     * ++dst; */
    *dst++ = *src++;
    *dst++ = *src++;
    *dst++ = *src++;
}
```



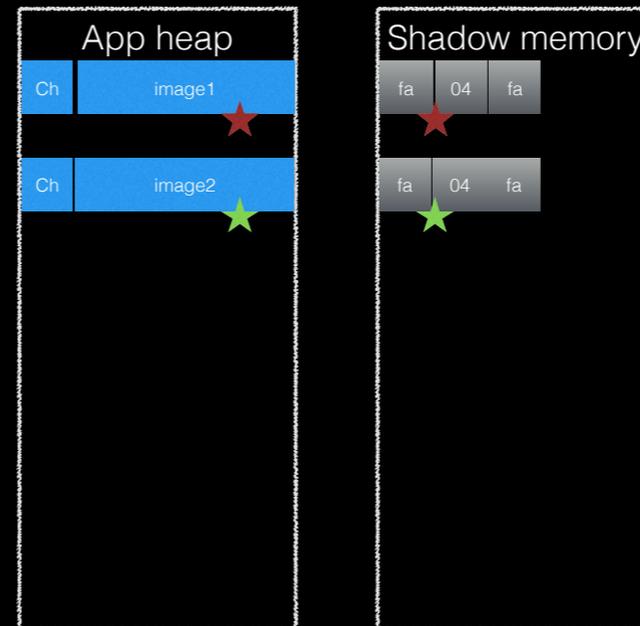
Step-by-step execution of instrumented code. Ch is chunk data, some metadata associated with the location. Pointers go out of bounds. ASan checks the shadow value. It figures that only 4 bytes of this 8 byte region is addressable. See `__asan_load1`.

# ASan – Step-By-Step

```
uint8_t *image1 = (uint8_t*)malloc(kImageSize);
uint8_t *image2 = (uint8_t*)malloc(kImageSize);

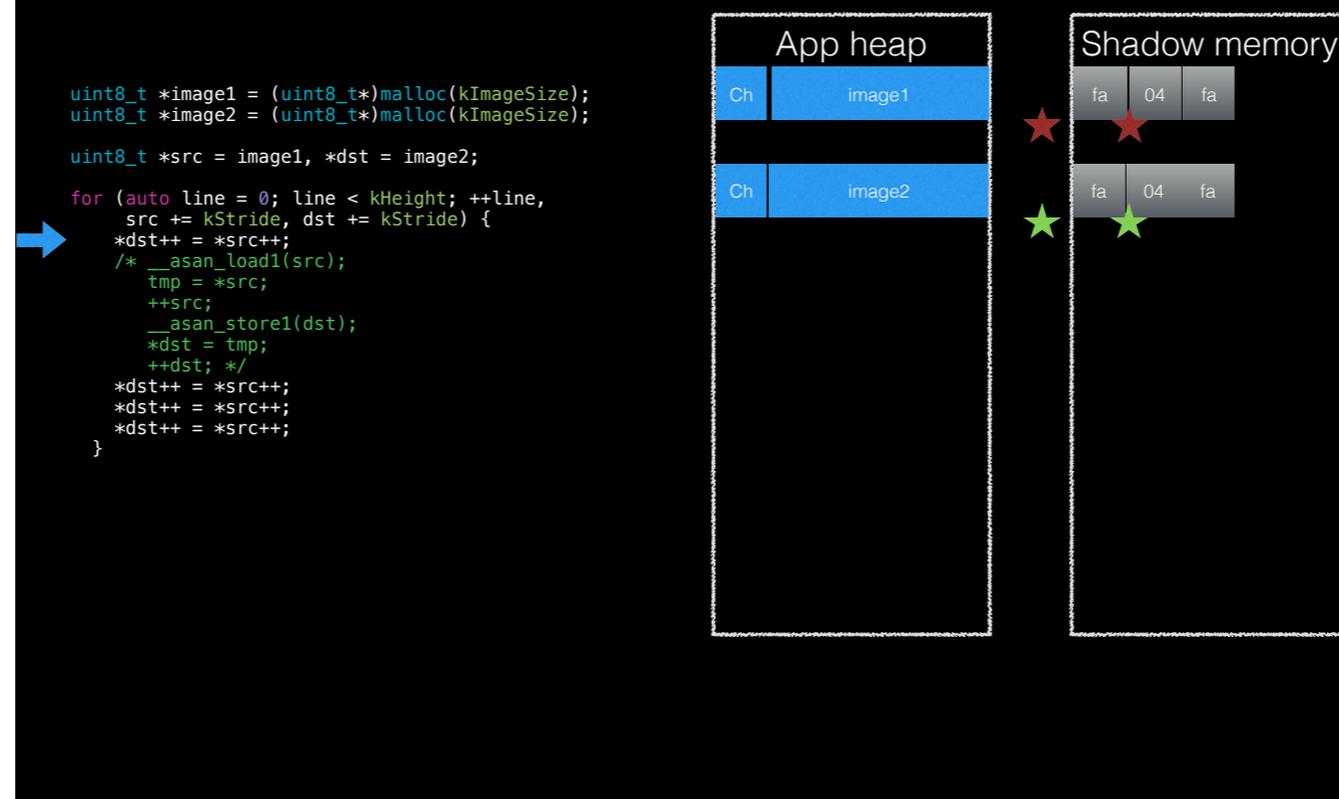
uint8_t *src = image1, *dst = image2;

for (auto line = 0; line < kHeight; ++line,
     src += kStride, dst += kStride) {
    *dst++ = *src++;
    /* __asan_load1(src);
     * tmp = *src;
     * ++src;
     * __asan_store1(dst);
     * *dst = tmp;
     * ++dst; */
    *dst++ = *src++;
    *dst++ = *src++;
    *dst++ = *src++;
}
```



Step-by-step execution of instrumented code. Ch is chunk data, some metadata associated with the location. Pointers go out of bounds. ASan checks the shadow value. It figures that only 4 bytes of this 8 byte region is addressable. See `__asan_load1`.

# ASan – Step-By-Step



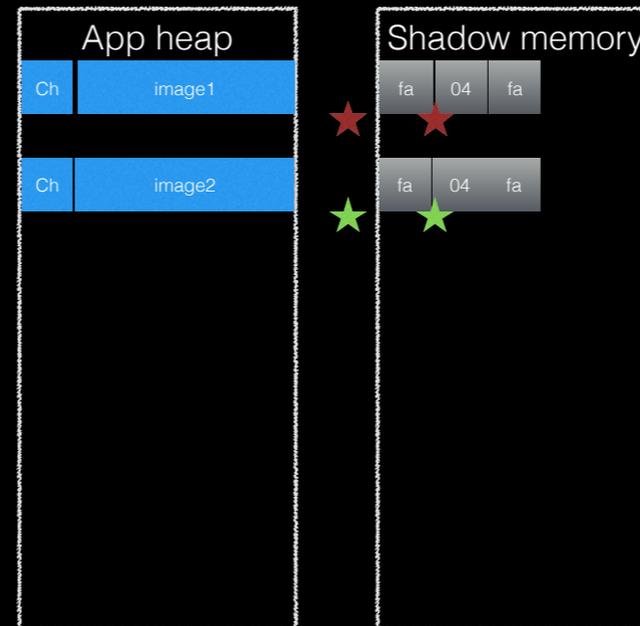
Step-by-step execution of instrumented code. Ch is chunk data, some metadata associated with the location. Pointers go out of bounds. ASan checks the shadow value. It figures that only 4 bytes of this 8 byte region is addressable. See `__asan_load1`.

# ASan – Step-By-Step

```
uint8_t *image1 = (uint8_t*)malloc(kImageSize);
uint8_t *image2 = (uint8_t*)malloc(kImageSize);

uint8_t *src = image1, *dst = image2;

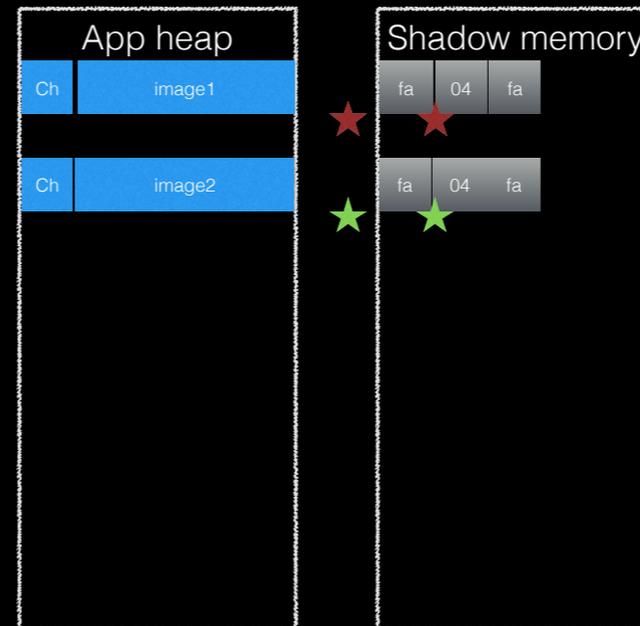
for (auto line = 0; line < kHeight; ++line,
     src += kStride, dst += kStride) {
  *dst++ = *src++;
  /* __asan_load1(src);
   tmp = *src;
   ++src;
   __asan_store1(dst);
   *dst = tmp;
   ++dst; */
  *dst++ = *src++;
  *dst++ = *src++;
  *dst++ = *src++;
}
```



Step-by-step execution of instrumented code. Ch is chunk data, some metadata associated with the location. Pointers go out of bounds. ASan checks the shadow value. It figures that only 4 bytes of this 8 byte region is addressable. See `__asan_load1`.

# ASan – Step-By-Step

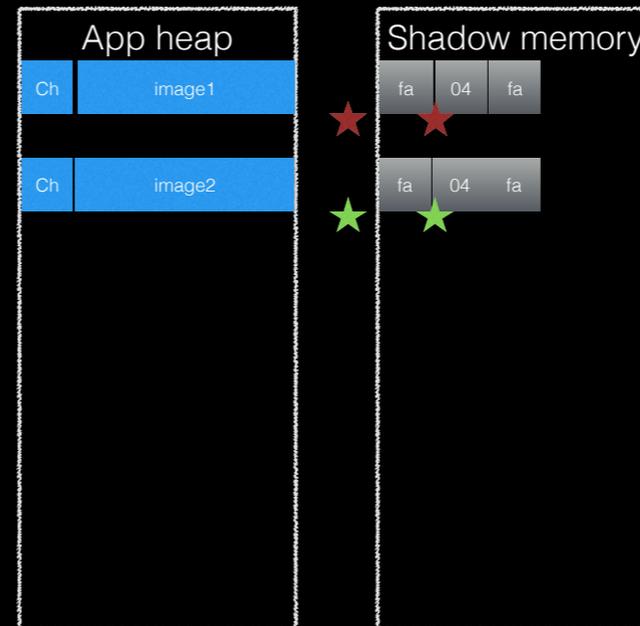
```
void __asan_load1(unsigned long addr) {  
    const unsigned long kLoadSize = 1; // __asan_load1  
    const unsigned long kShadowGranularity = 1ULL << 3;  
    unsigned long ShadowPtr = (addr >> 3) +  
        kDefaultShadowOffset64;  
    unsigned long Shadow = kLoadSize <= kShadowGranularity  
        ? *reinterpret_cast<unsigned char *>(ShadowPtr)  
        : *reinterpret_cast<unsigned short *>(ShadowPtr);  
    if (Shadow && (1 >= kShadowGranularity ||  
        ((signed char)((addr & (kShadowGranularity - 1)) +  
            1 - 1)) >= (signed char)Shadow)) {  
        unsigned long bp = (unsigned long)  
            __builtin_frame_address(0);  
        unsigned long pc = (unsigned long)  
            __builtin_return_address(0);  
        unsigned long local_stack;  
        unsigned long sp = (unsigned long)&local_stack;  
        __asan_report_error(pc, bp, sp, addr,  
            false /*is_write*/,  
            1 /*size*/,  
            0 /*exp_arg*/);  
    }  
}
```



Step-by-step execution of instrumented code. Ch is chunk data, some metadata associated with the location. Pointers go out of bounds. ASan checks the shadow value. It figures that only 4 bytes of this 8 byte region is addressable. See `__asan_load1`.

# ASan – Step-By-Step

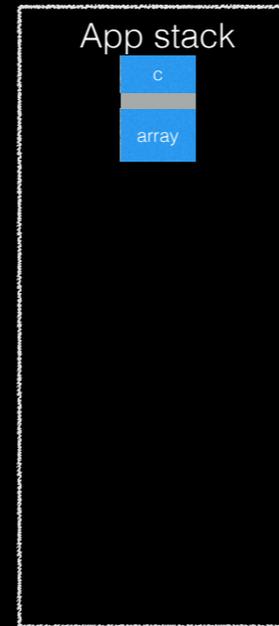
```
void __asan_load1(unsigned long addr) {  
  const unsigned long kLoadSize = 1; // __asan_load1  
  const unsigned long kShadowGranularity = 1ULL << 3;  
  unsigned long ShadowPtr = (addr >> 3) +  
    kDefaultShadowOffset64;  
  unsigned long Shadow = kLoadSize <= kShadowGranularity  
    ? *reinterpret_cast<unsigned char *>(ShadowPtr)  
    : *reinterpret_cast<unsigned short *>(ShadowPtr);  
  if (Shadow && (1 >= kShadowGranularity ||  
    ((signed char)((addr & (kShadowGranularity - 1)) +  
      1 - 1)) >= (signed char)Shadow)) {  
    unsigned long bp = (unsigned long)  
      __builtin_frame_address(0);  
    unsigned long pc = (unsigned long)  
      __builtin_return_address(0);  
    unsigned long local_stack;  
    unsigned long sp = (unsigned long)&local_stack;  
    __asan_report_error(pc, bp, sp, addr,  
      false /*is_write*/,  
      1 /*size*/,  
      0 /*exp_arg*/);  
  }  
}
```



Step-by-step execution of instrumented code. Ch is chunk data, some metadata associated with the location. Pointers go out of bounds. ASan checks the shadow value. It figures that only 4 bytes of this 8 byte region is addressable. See `__asan_load1`.

# ASan Stack Poisoning

```
int f(int a) {  
  char c[11];  
  int array[5];  
  // ..  
}
```



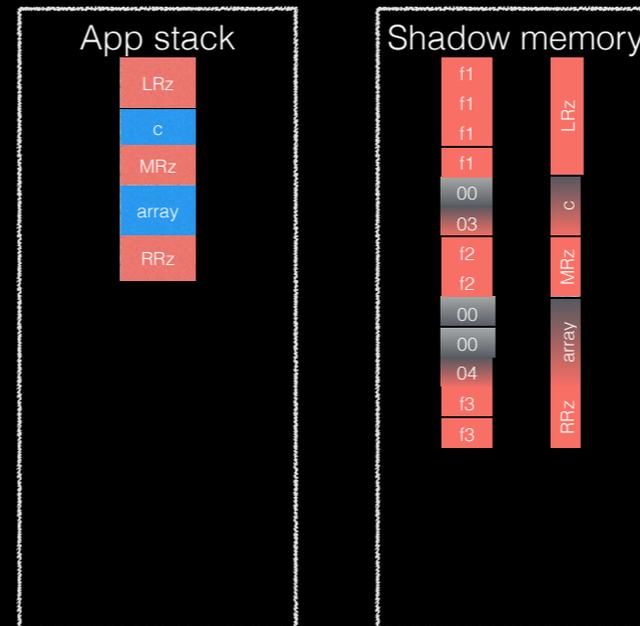
ASan inserts red zones between objects on the stack. This enables it detect overwrites. Left redzone contains some special data. It's 4 pointers wide region. First slot contains a constant value. Second index contains pointer to stack description string, which says "there are 2 objects, first one begins on offset 32, it's 11 bytes, name is 1 byte long, and name is c, second object starts at offset 64, it is 16 bytes long, its name is 5 bytes long, and its name is array". Third slot is program counter, f's address.

# ASan Stack Poisoning

```
int f(int a) {  
    // sizeof(long) * CHAR_BIT / 2  
    char left_redzone[4 * sizeof(void*)];  
    char c[11];  
    char mid_redzone[21];  
    int array[5];  
    char right_redzone[16];  
    // ..  
}
```

## Left red-zone contents

0	0x0000000041b58ab3	Current stack frame magic
1	char *StackDescr	2 32 11 1 c 64 16 5 array
2	PC	f
3	0	



ASan inserts red zones between objects on the stack. This enables it detect overwrites. Left redzone contains some special data. It's 4 pointers wide region. First slot contains a constant value. Second index contains pointer to stack description string, which says "there are 2 objects, first one begins on offset 32, it's 11 bytes, name is 1 byte long, and name is c, second object starts at offset 64, it is 16 bytes long, its name is 5 bytes long, and its name is array". Third slot is program counter, f's address.

# ASan - Container Overflow

```
1. #include <vector>
2. #include <assert.h>
3. typedef long T;
4. int main() {
5.     std::vector<T> v;
6.     v.push_back(0);
7.     v.push_back(1);
8.     v.push_back(2);
9.     assert(v.capacity() >= 4);
10.    assert(v.size() == 3);
11.    T *p = &v[0];
12.    // Here the memory is accessed inside a heap-allocated buffer
13.    // but outside of the region `[v.begin(), v.end())`.
14.    return p[3]; // OOPS.
15. }
```

```
=====  
==8326==ERROR: AddressSanitizer: container-overflow on address 0x6030000d828 at pc 0x000108d79395 bp  
0x7fff56e879c0 sp 0x7fff56e879a0  
READ of size 8 at 0x6030000d828 thread T0  
#0 0x108d79394 in main asan-container-overflow.cpp:14
```

<http://nullcon.net/website/archives/ppt/goa-15/analyzing-chrome-crash-reports-at-scale-by-abhishek-arya.pdf>

ASan has annotation function called by the container when its size changes. It then detects such problems. Marshal Clow noted, I wouldn't know if it wasn't for him, that this is exception safe. That is, if exception is thrown, it can roll back and unpoison regions. Jonathan Wakely also noted that Kostya has submitted patches for libstdc++, but not included yet, probably because it didn't have exception roll back mechanism.

# ASan - Intra-object Red Zone

```
1. class A {
2.     char c[5];
3.     // red-zone after 'c'
4.     int a;
5.     // red-zone after 'a'
6. public:
7.     A() : a(3) {
8.         // __asan_poison_intra_object_redzone(&c)
9.         // __asan_poison_intra_object_redzone(&a)
10.    }
11.    virtual ~A() { }
12.    void set_c(char v, unsigned index) { c[index] = v; }
13.};

15. int main(int argc, char *argv[]) {
16.     A a;
17.     a.set_c('0', argc);
18.     return 0;
19.}
```

```
% clang++ -fsanitize=address -fsanitize-address-field-padding=1 -o intra-object-rz && ./intra-object-rz
```

```
====
==40302==ERROR: AddressSanitizer: intra-object-rz on address 0x7fff54f2c68d at pc 0x00010acd3c22 bp
0x7fff54f2c5f0 sp 0x7fff54f2c5d0
WRITE of size 1 at 0x7fff54f2c68d thread T0
```

ASan can detect overflows between class members as well. This happens by inserting a special function in constructor; `__asan_poison_intra_object_redzone`. Note that this happens in CodeGen, not in AddressSanitizer pass.

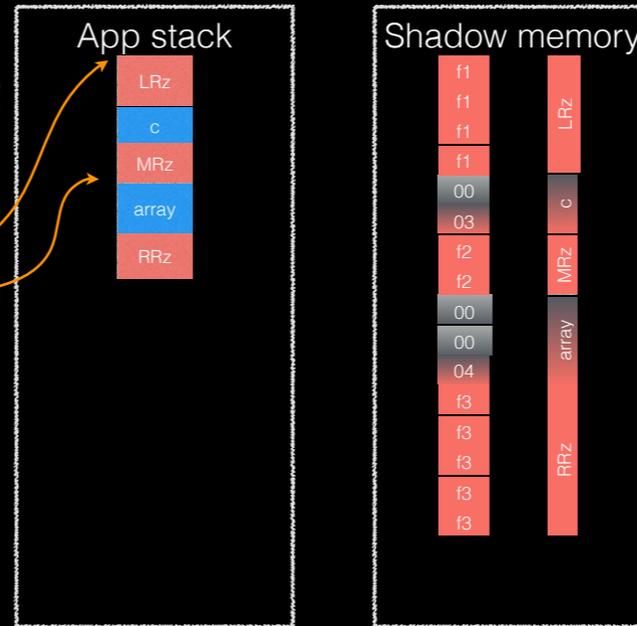
# ASan - Under/overshoot

```
1. int main() {
2.     // char left_redzone[32];
3.     char c[2] = {0};
4.     // char mid_redzone[14]; - alignment of 'i' is 16
5.     int array[2] = {0};
6.     // char right_redzone[24];
7.     char *p = c;
8.     // char right_redzone[8]; - aligns to 8/16
9.     // char right_redzone[32];
10.    p[-33] = '4'; // before left_redzone
11.    p[16] = '0'; // 'array's storage
12.}
```

It is possible to write before or after redzones. In this case, asan's checks will not be able to detect these reads and writes.

# ASan - Under/overshoot

```
1. int main() {  
2.   // char left_redzone[32];  
3.   char c[2] = {0};  
4.   // char mid_redzone[14]; - alignment of 'i' is 16  
5.   int array[2] = {0};  
6.   // char right_redzone[24];  
7.   char *p = c;  
8.   // char right_redzone[8]; - aligns to 8/16  
9.   // char right_redzone[32];  
10.  p[-33] = '4'; // before left_redzone  
11.  p[16] = '0'; // 'array's storage  
12.}
```

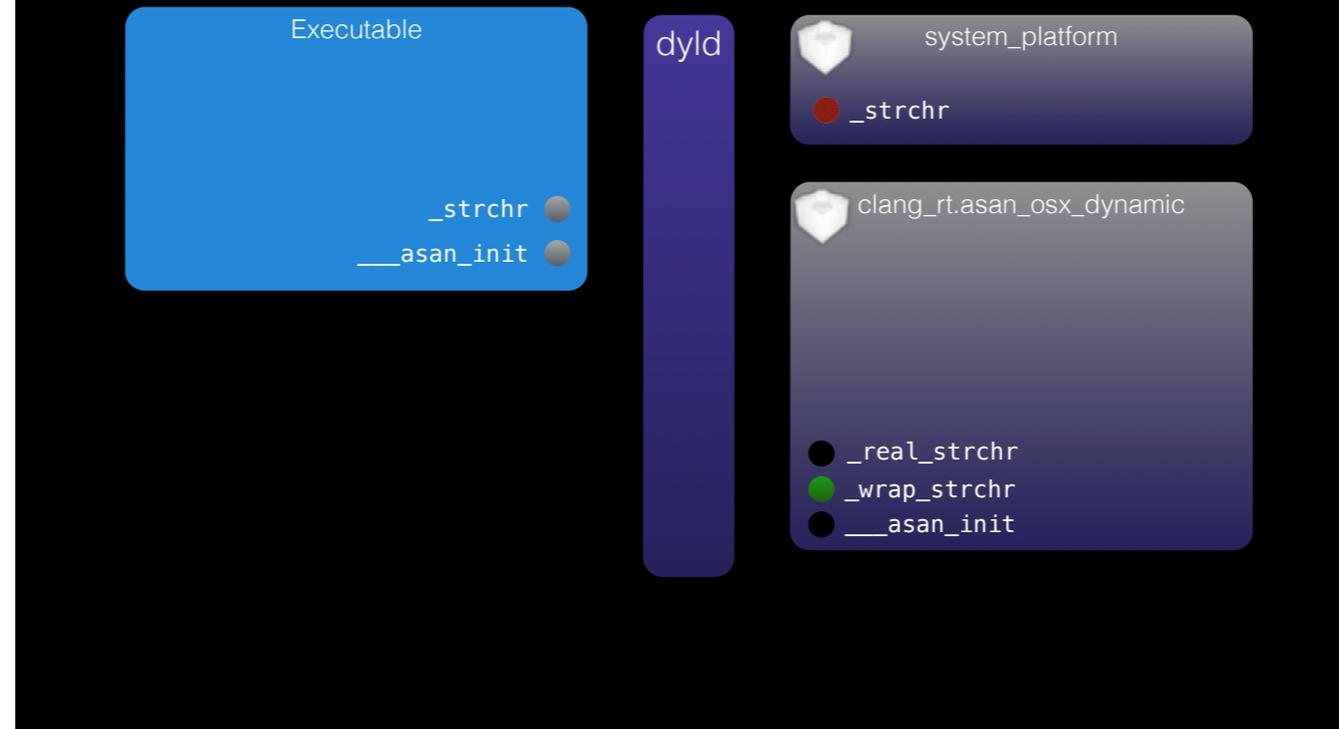


It is possible to write before or after redzones. In this case, asan's checks will not be able to detect these reads and writes.

# ASan - Interposing Functions On Darwin (a.k.a Mac OS X)

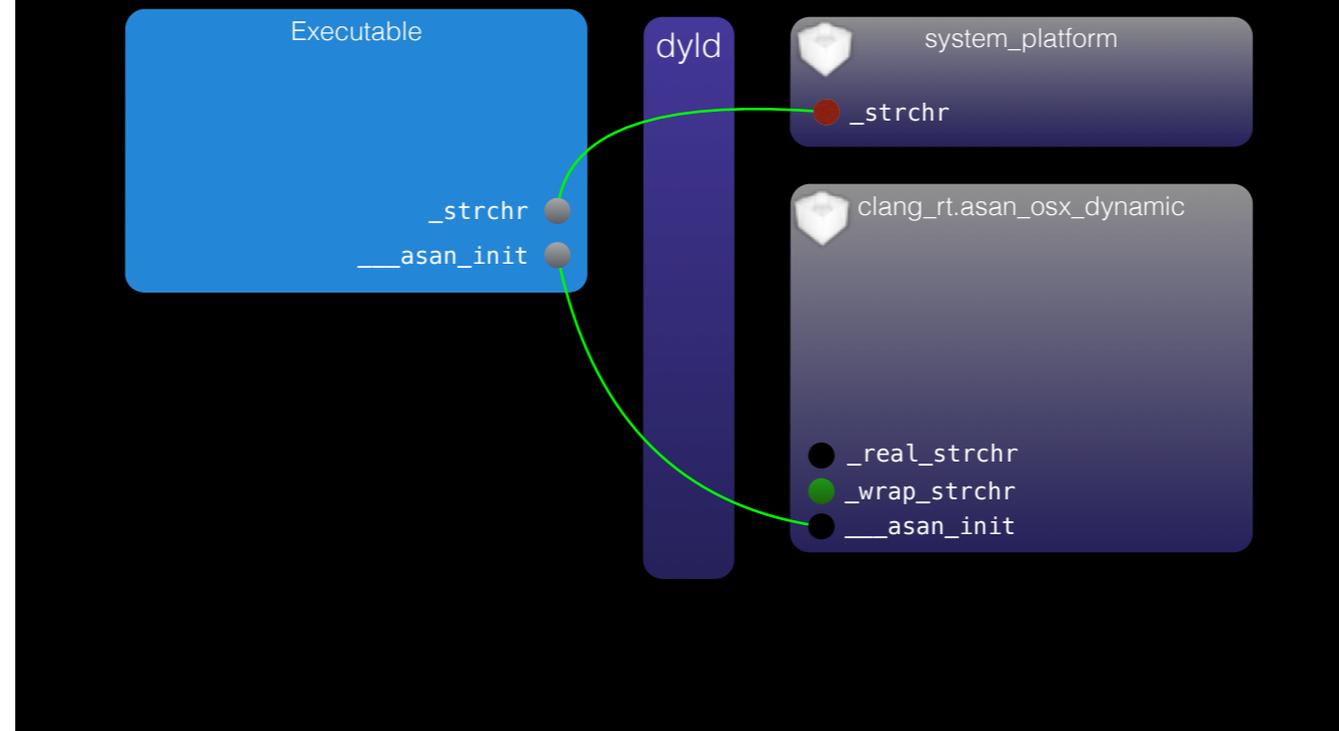
On Darwin, function interposing works slightly different than Linux. Linker reads a special section in Mach-O file named `__DATA,__interpose` to see which functions have replacements. When executable starts, it calls `__asan_init`. Initially, `_strchr` is bound to symbol in `libsystem_platform.dylib`. `__asan_init` checks whether ASan runtime library's name is in environment variable named `DYLD_INSERT_LIBRARIES`. This environment variable tells runtime linker to load given libraries before others. If ASan cannot find its name in `DYLD_INSERT_LIBRARIES` environment variable, it inserts itself in this environment variable, and re-execs the executable with exact arguments provided. So, the only difference between this re-exec and the initial one is presence of ASan runtime library's name in `DYLD_INSERT_LIBRARIES` environment variable. During re-exec, runtime linker finds that ASan runtime library interposes some functions, because it has `__DATA,__interpose` section. Runtime linker then binds, in this example, `strchr` to its the symbol in ASan runtime. When application calls `strchr`, it goes to ASan runtime, not to `system_platform`. ASan might call original function in `system_platform` or might have its own implementation.

# ASan - Interposing Functions On Darwin (a.k.a Mac OS X)



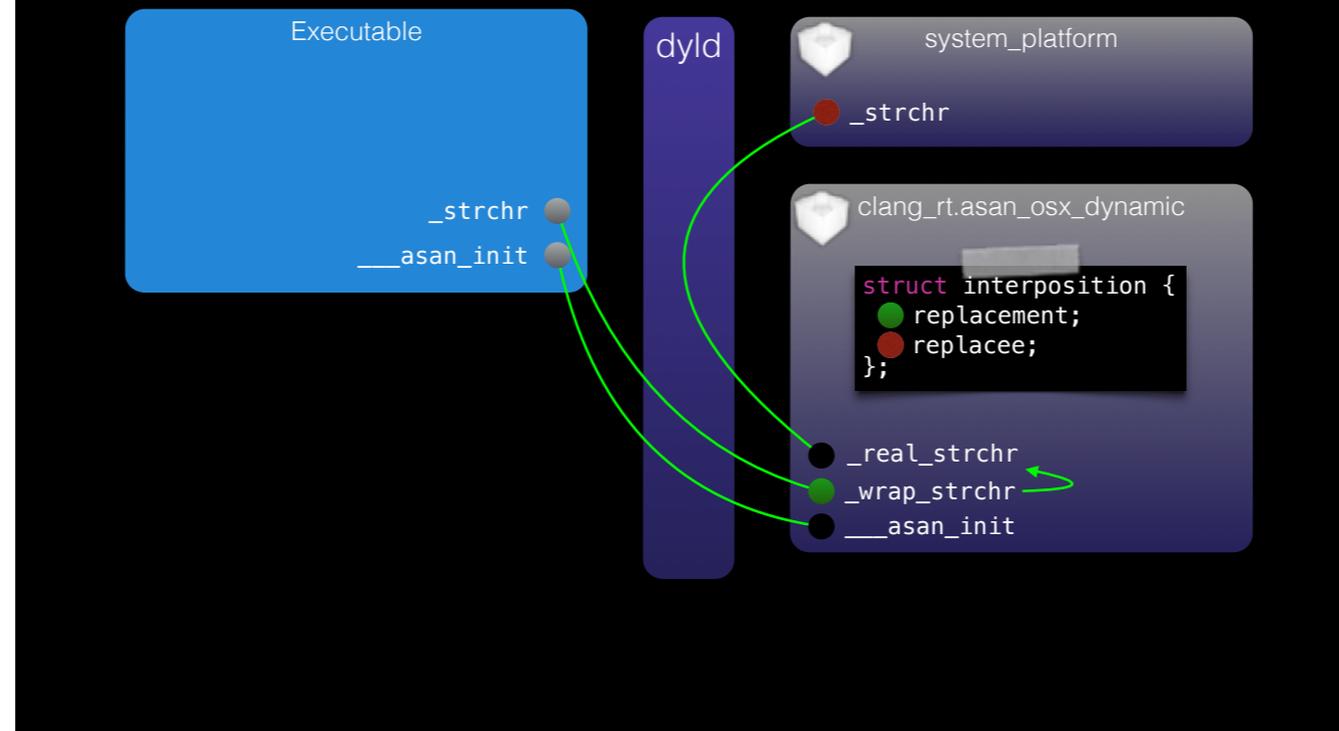
On Darwin, function interposing works slightly different than Linux. Linker reads a special section in Mach-O file named `__DATA,__interpose` to see which functions have replacements. When executable starts, it calls `__asan_init`. Initially, `_strchr` is bound to symbol in `libsystem_platform.dylib`. `__asan_init` checks whether ASan runtime library's name is in environment variable named `DYLD_INSERT_LIBRARIES`. This environment variable tells runtime linker to load given libraries before others. If ASan cannot find its name in `DYLD_INSERT_LIBRARIES` environment variable, it inserts itself in this environment variable, and re-execs the executable with exact arguments provided. So, the only difference between this re-exec and the initial one is presence of ASan runtime library's name in `DYLD_INSERT_LIBRARIES` environment variable. During re-exec, runtime linker finds that ASan runtime library interposes some functions, because it has `__DATA,__interpose` section. Runtime linker then binds, in this example, `strchr` to its the symbol in ASan runtime. When application calls `strchr`, it goes to ASan runtime, not to `system_platform`. ASan might call original function in `system_platform` or might have its own implementation.

# ASan - Interposing Functions On Darwin (a.k.a Mac OS X)



On Darwin, function interposing works slightly different than Linux. Linker reads a special section in Mach-O file named `__DATA,__interpose` to see which functions have replacements. When executable starts, it calls `__asan_init`. Initially, `_strchr` is bound to symbol in `libsystem_platform.dylib`. `__asan_init` checks whether ASan runtime library's name is in environment variable named `DYLD_INSERT_LIBRARIES`. This environment variable tells runtime linker to load given libraries before others. If ASan cannot find its name in `DYLD_INSERT_LIBRARIES` environment variable, it inserts itself in this environment variable, and re-execs the executable with exact arguments provided. So, the only difference between this re-exec and the initial one is presence of ASan runtime library's name in `DYLD_INSERT_LIBRARIES` environment variable. During re-exec, runtime linker finds that ASan runtime library interposes some functions, because it has `__DATA,__interpose` section. Runtime linker then binds, in this example, `strchr` to its the symbol in ASan runtime. When application calls `strchr`, it goes to ASan runtime, not to `system_platform`. ASan might call original function in `system_platform` or might have its own implementation.

# ASan - Interposing Functions On Darwin (a.k.a Mac OS X)



On Darwin, function interposing works slightly different than Linux. Linker reads a special section in Mach-O file named `__DATA,__interpose` to see which functions have replacements. When executable starts, it calls `__asan_init`. Initially, `_strchr` is bound to symbol in `libsystem_platform.dylib`. `__asan_init` checks whether ASan runtime library's name is in environment variable named `DYLD_INSERT_LIBRARIES`. This environment variable tells runtime linker to load given libraries before others. If ASan cannot find its name in `DYLD_INSERT_LIBRARIES` environment variable, it inserts itself in this environment variable, and re-execs the executable with exact arguments provided. So, the only difference between this re-exec and the initial one is presence of ASan runtime library's name in `DYLD_INSERT_LIBRARIES` environment variable. During re-exec, runtime linker finds that ASan runtime library interposes some functions, because it has `__DATA,__interpose` section. Runtime linker then binds, in this example, `strchr` to its the symbol in ASan runtime. When application calls `strchr`, it goes to ASan runtime, not to `system_platform`. ASan might call original function in `system_platform` or might have its own implementation.

LSan

# LSan

- Detects leaks
- No compile-time instrumentation
- Static runtime-only
- Faster than ASan

LSan simply detects leaks, and has no compile-time instrumentation. It's a library. It's faster than ASan, because there is no instrumentation; only allocation and deallocation functions are intercepted.

# LSan ~~Instrumentation~~

- Inserts pointer to `__lsan_init` in ELF `.preinit_array` section
- Pushes `__lsan::DoLeakCheck` to `atexit` callback stack
- `__lsan_init` intercepts libc functions, `operator new`, and `operator delete`
- `__lsan::DoLeakCheck`:
  - Spawn a clone, and suspend all threads
  - Walk through globals, stack ranges, and TLS

I've hacked it, and kind of got it working on Mac OS X within a few hours, but it needed more work. I gave the lowest priority to Mac OS X port of LSan.

# LSan - Simple Example

```
1. int main() {  
2.   int *p = new int(42);  
3.   p = 0;  
4.   return 0;  
5. }
```

=====  
==518==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 4 byte(s) in 1 object(s) allocated from:

#0 0x4245a7 in operator new(unsigned long) llvm-trunk/llvm/projects/compiler-rt/lib/lsan/lsan\_interceptors.cc:154  
#1 0x42526a in main sanitizer-tests/lsan/lsan-basic.cpp:2:12  
#2 0x7f33ee4f17ff in \_\_libc\_start\_main (/usr/lib/libc.so.6+0x207ff)

SUMMARY: LeakSanitizer: 4 byte(s) leaked in 1 allocation(s).

TSan

# TSan

- Diagnoses
  - Data races
  - Dead-locks
  - Thread leaks
  - Mutex misuse
- Happens-before detector
- Does not require annotation
- Supports atomics

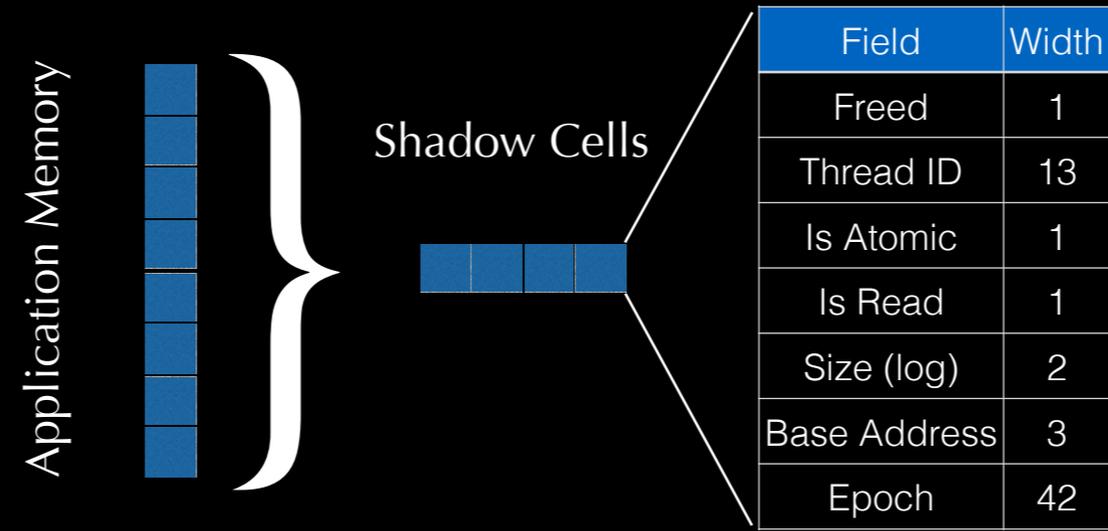
ThreadSanitizer detects all these problems, and a few more. It's a happens-before detector. It does not require annotations, and supports atomics.

# TSan instrumentation

- Traverse atomic memory access, load, store, return, call instructions, and some memory intrinsics (e.g. memcpy)
- Instrument function entry, and exits
- Instrument vtable accesses
- Run-time library interposes pthread and libc functions

It visits loads, stores, function calls and a few more things. Additionally, it instruments vtable access, because there is a hidden race there. Runtime library interposes pthread and C standard library functions.

# TSan – Shadow Memory



TSan has 4 shadow cells, each is 64-bit long. Each 64-bit value holds an information like shown in this table.

# TSan – Happens-Before

- Threads have a basic clock (epoch); incremented at each operation
- Threads also know about other threads' clocks
- Clocks synchronize at certain points (e.g. mutex, or atomic operations)
- According to current thread's clock cache, if the last write to shadow has smaller or equal epoch than the last-known-epoch of the last writer thread, *write happens-before*

Threads have clock. It's actually count of operations. Threads have clock cache. They know about each others' clocks. At certain times, clocks need to be synchronized. What TSan does, basically, is to find whether there is a happens-before relationship between two non-atomic operations, where at least one operation is a write. To do that, threads check their clock caches. Each thread checks 4 shadow cells before memory access. A check tests whether another thread has written to given location "before". Current thread, that is, thread that is reading or writing cell, checks previous write to this shadow cell. If it finds out another thread wrote there -- shadow cell's thread id is different from its thread id -- it searches its own clock cache, finds epoch of that other thread that wrote to this location, and compares that epoch value with the epoch written in the shadow cell. If shadow cell's epoch is greater than the last-known-epoch of writer thread, there is a problem. If shadow cell's epoch is less than the last-known epoch for writer thread, that means threads were synchronized after write this memory location. Let's see with an example.

# TSan – Step-By-Step

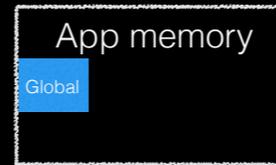
```
1. #include <pthread.h>
2. int Global;
3. void *Thread1(void *x) {
4.     /* __tsan_write4 */
5.     Global = 42;
6.     return x;
7. }
8. int main() {
9.     pthread_t t;
10.    pthread_create(&t, NULL, Thread1, NULL);
11.    /* __tsan_write4 */
12.    Global = 43;
13.    pthread_join(t, NULL);
14.    return Global;
15.}
```

Tables on bottom left represent clock caches. Left one is thread0's clock cache, and right one is clock 1's clock cache. Table on the right is shadow cells. NA values in clock cache mean that the thread is not created yet, so it's missing information. Thread #1 thinks that Thread#0's clock should have been 9500, but figures that Thread #0 has already written to the memory location that it wants to write at 9600 -- Thread #0 and #1 didn't synchronize in between. This is a race condition.

# TSan – Step-By-Step

```

1. #include <pthread.h>
2. int Global;
3. void *Thread1(void *x) {
4.     /* __tsan_write4 */
5.     Global = 42;
6.     return x;
7. }
8. int main() {
9.     pthread_t t;
10.    pthread_create(&t, NULL, Thread1, NULL);
11.    /* __tsan_write4 */
12.    Global = 43;
13.    pthread_join(t, NULL);
14.    return Global;
15.}
    
```



Clocks



Field
Thread ID
R/W
Size (log)
Base Address
Epoch

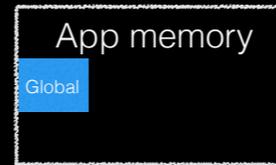
Tables on bottom left represent clock caches. Left one is thread0's clock cache, and right one is clock 1's clock cache. Table on the right is shadow cells. NA values in clock cache mean that the thread is not created yet, so it's missing information. Thread #1 thinks that Thread#0's clock should have been 9500, but figures that Thread #0 has already written to the memory location that it wants to write at 9600 -- Thread #0 and #1 didn't synchronize in between. This is a race condition.

# TSan – Step-By-Step

```

1. #include <pthread.h>
2. int Global;
3. void *Thread1(void *x) {
4.     /* __tsan_write4 */
5.     Global = 42;
6.     return x;
7. }
8. int main() {
9.     pthread_t t;
10.    pthread_create(&t, NULL, Thread1, NULL);
11.    /* __tsan_write4 */
12.    Global = 43;
13.    pthread_join(t, NULL);
14.    return Global;
15.}

```



Clocks

0	1	0	1
8192	NA		

Field
Thread ID
R/W
Size (log)
Base Address
Epoch

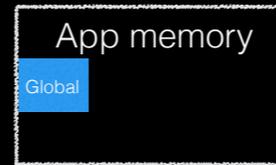
Tables on bottom left represent clock caches. Left one is thread0's clock cache, and right one is clock 1's clock cache. Table on the right is shadow cells. NA values in clock cache mean that the thread is not created yet, so it's missing information. Thread #1 thinks that Thread#0's clock should have been 9500, but figures that Thread #0 has already written to the memory location that it wants to write at 9600 -- Thread #0 and #1 didn't synchronize in between. This is a race condition.

# TSan – Step-By-Step

```

1. #include <pthread.h>
2. int Global;
3. void *Thread1(void *x) {
4.     /* __tsan_write4 */
5.     Global = 42;
6.     return x;
7. }
8. int main() {
9.     pthread_t t;
10.    pthread_create(&t, NULL, Thread1, NULL);
11.    /* __tsan_write4 */
12.    Global = 43;
13.    pthread_join(t, NULL);
14.    return Global;
15.}

```



## Clocks

0	1	0	1
8192	NA	NA	NA
9500	NA	NA	NA

Field
Thread ID
R/W
Size (log)
Base Address
Epoch

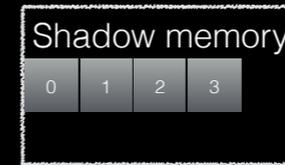
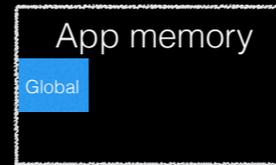
Tables on bottom left represent clock caches. Left one is thread0's clock cache, and right one is clock 1's clock cache. Table on the right is shadow cells. NA values in clock cache mean that the thread is not created yet, so it's missing information. Thread #1 thinks that Thread#0's clock should have been 9500, but figures that Thread #0 has already written to the memory location that it wants to write at 9600 -- Thread #0 and #1 didn't synchronize in between. This is a race condition.

# TSan – Step-By-Step

```

1. #include <pthread.h>
2. int Global;
3. void *Thread1(void *x) {
4.     /* __tsan_write4 */
5.     Global = 42;
6.     return x;
7. }
8. int main() {
9.     pthread_t t;
10.    pthread_create(&t, NULL, Thread1, NULL);
11.    /* __tsan_write4 */
12.    Global = 43;
13.    pthread_join(t, NULL);
14.    return Global;
15.}

```



## Clocks

0	1	0	1
8192	NA	NA	NA
9500	NA	NA	NA
		9500	8192

Field
Thread ID
R/W
Size (log)
Base Address
Epoch

Tables on bottom left represent clock caches. Left one is thread0's clock cache, and right one is clock 1's clock cache. Table on the right is shadow cells. NA values in clock cache mean that the thread is not created yet, so it's missing information. Thread #1 thinks that Thread#0's clock should have been 9500, but figures that Thread #0 has already written to the memory location that it wants to write at 9600 -- Thread #0 and #1 didn't synchronize in between. This is a race condition.

# TSan – Step-By-Step

```

1. #include <pthread.h>
2. int Global;
3. void *Thread1(void *x) {
4.     /* __tsan_write4 */
5.     Global = 42;
6.     return x;
7. }
8. int main() {
9.     pthread_t t;
10.    pthread_create(&t, NULL, Thread1, NULL);
11.    /* __tsan_write4 */
12.    Global = 43;
13.    pthread_join(t, NULL);
14.    return Global;
15.}

```



Field	0
Thread ID	0
R/W	W
Size (log)	2
Base Address	0
Epoch	9600

## Clocks

0	1	0	1
8192	NA	NA	NA
9500	NA	NA	NA
9600	8192	9500	8192

Tables on bottom left represent clock caches. Left one is thread0's clock cache, and right one is clock 1's clock cache. Table on the right is shadow cells. NA values in clock cache mean that the thread is not created yet, so it's missing information. Thread #1 thinks that Thread#0's clock should have been 9500, but figures that Thread #0 has already written to the memory location that it wants to write at 9600 -- Thread #0 and #1 didn't synchronize in between. This is a race condition.

# TSan – Step-By-Step

```

1. #include <pthread.h>
2. int Global;
3. void *Thread1(void *x) {
4.     /* __tsan_write4 */
5.     Global = 42;
6.     return x;
7. }
8. int main() {
9.     pthread_t t;
10.    pthread_create(&t, NULL, Thread1, NULL);
11.    /* __tsan_write4 */
12.    Global = 43;
13.    pthread_join(t, NULL);
14.    return Global;
15.}
    
```



Field	0	1	2	3
Thread ID	0	1		
R/W	W	W		
Size (log)	2	2		
Base Address	0	0		
Epoch	9600	X		

Clocks

0	1	0	1
8192	NA	NA	NA
9500	NA	NA	NA
9600	8192	9500	8192
9600	8192	9500	8193

Tables on bottom left represent clock caches. Left one is thread0's clock cache, and right one is clock 1's clock cache. Table on the right is shadow cells. NA values in clock cache mean that the thread is not created yet, so it's missing information. Thread #1 thinks that Thread#0's clock should have been 9500, but figures that Thread #0 has already written to the memory location that it wants to write at 9600 -- Thread #0 and #1 didn't synchronize in between. This is a race condition.

# TSan – Step-By-Step - Locked

```
1. #include <pthread.h>
2. int Global;
3. pthread_mutex_t g_mut = PTHREAD_MUTEX_INITIALIZER;
4. static void SetGlobal(int v) {
5.     pthread_mutex_lock(&g_mut);
6.     Global = v;
7.     pthread_mutex_unlock(&g_mut);
8. }
9. void *Thread1(void *x) {
10.    SetGlobal(42);
11.    return x;
12. }
13. int main() {
14.    pthread_t t;
15.    pthread_create(&t, NULL, Thread1, NULL);
16.    SetGlobal(43);
17.    pthread_join(t, NULL);
18.    return Global;
19. }
```

Table on bottom left shows line numbers and arbitrary epochs. T0 is line number of thread 0. T1 is line number of thread1. Cx,y shows clock cache, where x represents the thread that owns the cache, and y is other thread. So, C0,0 means thread 0's epoch, while C0,1 is thread1's last-known epoch in thread 0's clock cache. When Thread #0 acquires mutex, it creates a synchronization variable. Clocks are not synchronized yet. When Thread #1 acquires the same mutex, it finds a new synchronization variable to synchronize clocks for. Then, it updates its clock cache by synchronizing with clocks on the synchronization object. That epoch is the epoch when Thread #0 released the mutex.

# TSan – Step-By-Step - Locked

```

1. #include <pthread.h>
2. int Global;
3. pthread_mutex_t g_mut = PTHREAD_MUTEX_INITIALIZER;
4. static void SetGlobal(int v) {
5.     pthread_mutex_lock(&g_mut);
6.     Global = v;
7.     pthread_mutex_unlock(&g_mut);
8. }
9. void *Thread1(void *x) {
10.    SetGlobal(42);
11.    return x;
12. }
13. int main() {
14.    pthread_t t;
15.    pthread_create(&t, NULL, Thread1, NULL);
16.    SetGlobal(43);
17.    pthread_join(t, NULL);
18.    return Global;
19. }

```



T <sub>0</sub>	T <sub>1</sub>	C <sub>0,0</sub>	C <sub>0,1</sub>	C <sub>1,0</sub>	C <sub>1,1</sub>	S
14		8192	NA	NA	NA	NA
16		9500	8192	NA	NA	NA
5	10	9600	8192	9500	8200	&g_mut
6	5	9700	8192	9500	8200	&g_mut
7	5	9750	8192	9500	8200	&g_mut
17	6	9800	8192	9750	8300	NA
18		9850	8350	NA	NA	NA

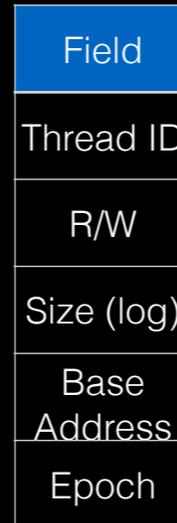
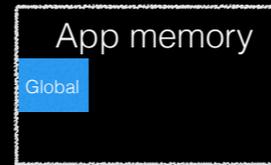
Table on bottom left shows line numbers and arbitrary epochs. T<sub>0</sub> is line number of thread 0. T<sub>1</sub> is line number of thread1. C<sub>x,y</sub> shows clock cache, where x represents the thread that owns the cache, and y is other thread. So, C<sub>0,0</sub> means thread 0's epoch, while C<sub>0,1</sub> is thread1's last-known epoch in thread 0's clock cache. When Thread #0 acquires mutex, it creates a synchronization variable. Clocks are not synchronized yet. When Thread #1 acquires the same mutex, it finds a new synchronization variable to synchronize clocks for. Then, it updates its clock cache by synchronizing with clocks on the synchronization object. That epoch is the epoch when Thread #0 released the mutex.

# TSan – Step-By-Step - Locked

```

1. #include <pthread.h>
2. int Global;
3. pthread_mutex_t g_mut = PTHREAD_MUTEX_INITIALIZER;
4. static void SetGlobal(int v) {
5.     pthread_mutex_lock(&g_mut);
6.     Global = v;
7.     pthread_mutex_unlock(&g_mut);
8. }
9. void *Thread1(void *x) {
10.    SetGlobal(42);
11.    return x;
12. }
13. int main() {
14.    pthread_t t;
15.    pthread_create(&t, NULL, Thread1, NULL);
16.    SetGlobal(43);
17.    pthread_join(t, NULL);
18.    return Global;
19. }

```



T <sub>0</sub>	T <sub>1</sub>	C <sub>0,0</sub>	C <sub>0,1</sub>	C <sub>1,0</sub>	C <sub>1,1</sub>	S
14		8192	NA	NA	NA	NA
16		9500	8192	NA	NA	NA
5	10	9600	8192	9500	8200	&g_mut
6	5	9700	8192	9500	8200	&g_mut
7	5	9750	8192	9500	8200	&g_mut
17	6	9800	8192	9750	8300	NA
18		9850	8350	NA	NA	NA

Table on bottom left shows line numbers and arbitrary epochs. T<sub>0</sub> is line number of thread 0. T<sub>1</sub> is line number of thread 1. C<sub>x,y</sub> shows clock cache, where x represents the thread that owns the cache, and y is other thread. So, C<sub>0,0</sub> means thread 0's epoch, while C<sub>0,1</sub> is thread 1's last-known epoch in thread 0's clock cache. When Thread #0 acquires mutex, it creates a synchronization variable. Clocks are not synchronized yet. When Thread #1 acquires the same mutex, it finds a new synchronization variable to synchronize clocks for. Then, it updates its clock cache by synchronizing with clocks on the synchronization object. That epoch is the epoch when Thread #0 released the mutex.

# TSan – Step-By-Step - Locked

```

1. #include <pthread.h>
2. int Global;
3. pthread_mutex_t g_mut = PTHREAD_MUTEX_INITIALIZER;
4. static void SetGlobal(int v) {
5.     pthread_mutex_lock(&g_mut);
6.     Global = v;
7.     pthread_mutex_unlock(&g_mut);
8. }
9. void *Thread1(void *x) {
10.    SetGlobal(42);
11.    return x;
12. }
13. int main() {
14.    pthread_t t;
15.    pthread_create(&t, NULL, Thread1, NULL);
16.    SetGlobal(43);
17.    pthread_join(t, NULL);
18.    return Global;
19. }
                
```

App memory

Global

Shadow memory

0 1 2 3

Field						
Thread ID						
R/W						
Size (log)						
Base Address						
Epoch						

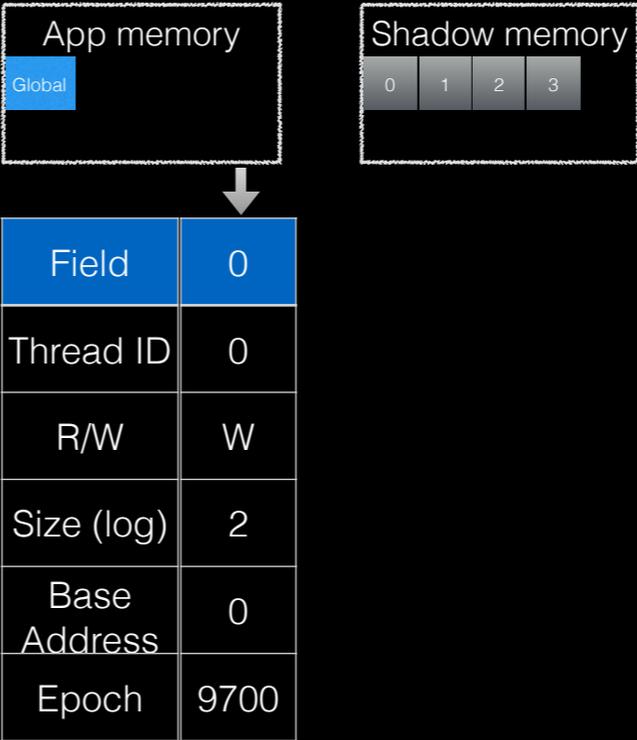
T <sub>0</sub>	T <sub>1</sub>	C <sub>0,0</sub>	C <sub>0,1</sub>	C <sub>1,0</sub>	C <sub>1,1</sub>	S
14		8192	NA	NA	NA	NA
16		9500	8192	NA	NA	NA
5	10	9600	8192	9500	8200	&g_mut
6	5	9700	8192	9500	8200	&g_mut
7	5	9750	8192	9500	8200	&g_mut
17	6	9800	8192	9750	8300	NA
18		9850	8350	NA	NA	NA

Table on bottom left shows line numbers and arbitrary epochs. T<sub>0</sub> is line number of thread 0. T<sub>1</sub> is line number of thread 1. C<sub>x,y</sub> shows clock cache, where x represents the thread that owns the cache, and y is other thread. So, C<sub>0,0</sub> means thread 0's epoch, while C<sub>0,1</sub> is thread 1's last-known epoch in thread 0's clock cache. When Thread #0 acquires mutex, it creates a synchronization variable. Clocks are not synchronized yet. When Thread #1 acquires the same mutex, it finds a new synchronization variable to synchronize clocks for. Then, it updates its clock cache by synchronizing with clocks on the synchronization object. That epoch is the epoch when Thread #0 released the mutex.

# TSan – Step-By-Step - Locked

```

1. #include <pthread.h>
2. int Global;
3. pthread_mutex_t g_mut = PTHREAD_MUTEX_INITIALIZER;
4. static void SetGlobal(int v) {
5.     pthread_mutex_lock(&g_mut);
6.     Global = v;
7.     pthread_mutex_unlock(&g_mut);
8. }
9. void *Thread1(void *x) {
10.    SetGlobal(42);
11.    return x;
12. }
13. int main() {
14.    pthread_t t;
15.    pthread_create(&t, NULL, Thread1, NULL);
16.    SetGlobal(43);
17.    pthread_join(t, NULL);
18.    return Global;
19. }
    
```



T0	T1	C0,0	C0,1	C1,0	C1,1	S
14		8192	NA	NA	NA	NA
16		9500	8192	NA	NA	NA
5	10	9600	8192	9500	8200	&g_mut
6	5	9700	8192	9500	8200	&g_mut
7	5	9750	8192	9500	8200	&g_mut
17	6	9800	8192	9750	8300	NA
18		9850	8350	NA	NA	NA

Table on bottom left shows line numbers and arbitrary epochs. T0 is line number of thread 0. T1 is line number of thread1. Cx,y shows clock cache, where x represents the thread that owns the cache, and y is other thread. So, C0,0 means thread 0's epoch, while C0,1 is thread1's last-known epoch in thread 0's clock cache. When Thread #0 acquires mutex, it creates a synchronization variable. Clocks are not synchronized yet. When Thread #1 acquires the same mutex, it finds a new synchronization variable to synchronize clocks for. Then, it updates its clock cache by synchronizing with clocks on the synchronization object. That epoch is the epoch when Thread #0 released the mutex.

# TSan – Step-By-Step - Locked

```

1. #include <pthread.h>
2. int Global;
3. pthread_mutex_t g_mut = PTHREAD_MUTEX_INITIALIZER;
4. static void SetGlobal(int v) {
5.     pthread_mutex_lock(&g_mut);
6.     Global = v;
7.     pthread_mutex_unlock(&g_mut);
8. }
9. void *Thread1(void *x) {
10.    SetGlobal(42);
11.    return x;
12. }
13. int main() {
14.    pthread_t t;
15.    pthread_create(&t, NULL, Thread1, NULL);
16.    SetGlobal(43);
17.    pthread_join(t, NULL);
18.    return Global;
19. }
    
```

App memory

Global

Shadow memory

0 1 2 3

↓

Field	0
Thread ID	0
R/W	W
Size (log)	2
Base Address	0
Epoch	9700

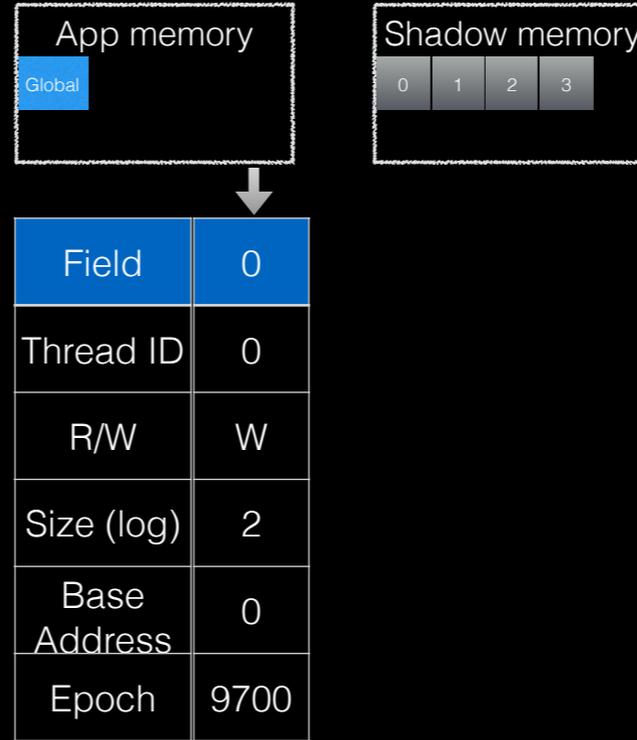
T <sub>0</sub>	T <sub>1</sub>	C <sub>0,0</sub>	C <sub>0,1</sub>	C <sub>1,0</sub>	C <sub>1,1</sub>	S
14		8192	NA	NA	NA	NA
16		9500	8192	NA	NA	NA
5	10	9600	8192	9500	8200	&g_mut
6	5	9700	8192	9500	8200	&g_mut
7	5	9750	8192	9500	8200	&g_mut
17	6	9800	8192	9750	8300	NA
18		9850	8350	NA	NA	NA

Table on bottom left shows line numbers and arbitrary epochs. T<sub>0</sub> is line number of thread 0. T<sub>1</sub> is line number of thread1. C<sub>x,y</sub> shows clock cache, where x represents the thread that owns the cache, and y is other thread. So, C<sub>0,0</sub> means thread 0's epoch, while C<sub>0,1</sub> is thread1's last-known epoch in thread 0's clock cache. When Thread #0 acquires mutex, it creates a synchronization variable. Clocks are not synchronized yet. When Thread #1 acquires the same mutex, it finds a new synchronization variable to synchronize clocks for. Then, it updates its clock cache by synchronizing with clocks on the synchronization object. That epoch is the epoch when Thread #0 released the mutex.

# TSan – Step-By-Step - Locked

```

1. #include <pthread.h>
2. int Global;
3. pthread_mutex_t g_mut = PTHREAD_MUTEX_INITIALIZER;
4. static void SetGlobal(int v) {
5.     pthread_mutex_lock(&g_mut);
6.     Global = v;
7.     pthread_mutex_unlock(&g_mut);
8. }
9. void *Thread1(void *x) {
10.    SetGlobal(42);
11.    return x;
12. }
13. int main() {
14.    pthread_t t;
15.    pthread_create(&t, NULL, Thread1, NULL);
16.    SetGlobal(43);
17.    pthread_join(t, NULL);
18.    return Global;
19. }
    
```



T0	T1	C0,0	C0,1	C1,0	C1,1	S
14		8192	NA	NA	NA	NA
16		9500	8192	NA	NA	NA
5	10	9600	8192	9500	8200	&g_mut
6	5	9700	8192	9500	8200	&g_mut
7	5	9750	8192	9500	8200	&g_mut
17	6	9800	8192	9750	8300	NA
18		9850	8350	NA	NA	NA

Table on bottom left shows line numbers and arbitrary epochs. T0 is line number of thread 0. T1 is line number of thread1. Cx,y shows clock cache, where x represents the thread that owns the cache, and y is other thread. So, C0,0 means thread 0's epoch, while C0,1 is thread1's last-known epoch in thread 0's clock cache. When Thread #0 acquires mutex, it creates a synchronization variable. Clocks are not synchronized yet. When Thread #1 acquires the same mutex, it finds a new synchronization variable to synchronize clocks for. Then, it updates its clock cache by synchronizing with clocks on the synchronization object. That epoch is the epoch when Thread #0 released the mutex.

# TSan – Step-By-Step - Locked

```

1. #include <pthread.h>
2. int Global;
3. pthread_mutex_t g_mut = PTHREAD_MUTEX_INITIALIZER;
4. static void SetGlobal(int v) {
5.     pthread_mutex_lock(&g_mut);
6.     Global = v;
7.     pthread_mutex_unlock(&g_mut);
8. }
9. void *Thread1(void *x) {
10.    SetGlobal(42);
11.    return x;
12. }
13. int main() {
14.    pthread_t t;
15.    pthread_create(&t, NULL, Thread1, NULL);
16.    SetGlobal(43);
17.    pthread_join(t, NULL);
18.    return Global;
19. }
                
```

App memory

Global

Shadow memory

0 1 2 3

↓

Field	0
Thread ID	0
R/W	W
Size (log)	2
Base Address	0
Epoch	9700

T <sub>0</sub>	T <sub>1</sub>	C <sub>0,0</sub>	C <sub>0,1</sub>	C <sub>1,0</sub>	C <sub>1,1</sub>	S
14		8192	NA	NA	NA	NA
16		9500	8192	NA	NA	NA
5	10	9600	8192	9500	8200	&g_mut
6	5	9700	8192	9500	8200	&g_mut
7	5	9750	8192	9500	8200	&g_mut
17	6	9800	8192	9750	8300	NA
18		9850	8350	NA	NA	NA

Table on bottom left shows line numbers and arbitrary epochs. T<sub>0</sub> is line number of thread 0. T<sub>1</sub> is line number of thread1. C<sub>x,y</sub> shows clock cache, where x represents the thread that owns the cache, and y is other thread. So, C<sub>0,0</sub> means thread 0's epoch, while C<sub>0,1</sub> is thread1's last-known epoch in thread 0's clock cache. When Thread #0 acquires mutex, it creates a synchronization variable. Clocks are not synchronized yet. When Thread #1 acquires the same mutex, it finds a new synchronization variable to synchronize clocks for. Then, it updates its clock cache by synchronizing with clocks on the synchronization object. That epoch is the epoch when Thread #0 released the mutex.

# TSan – Step-By-Step - Locked

```

1. #include <pthread.h>
2. int Global;
3. pthread_mutex_t g_mut = PTHREAD_MUTEX_INITIALIZER;
4. static void SetGlobal(int v) {
5.     pthread_mutex_lock(&g_mut);
6.     Global = v;
7.     pthread_mutex_unlock(&g_mut);
8. }
9. void *Thread1(void *x) {
10.    SetGlobal(42);
11.    return x;
12. }
13. int main() {
14.    pthread_t t;
15.    pthread_create(&t, NULL, Thread1, NULL);
16.    SetGlobal(43);
17.    pthread_join(t, NULL);
18.    return Global;
19. }
    
```

App memory

Global

Shadow memory

0 1 2 3

Field	0	1
Thread ID	0	1
R/W	W	W
Size (log)	2	2
Base Address	0	0
Epoch	9700	8300

T <sub>0</sub>	T <sub>1</sub>	C <sub>0,0</sub>	C <sub>0,1</sub>	C <sub>1,0</sub>	C <sub>1,1</sub>	S
14		8192	NA	NA	NA	NA
16		9500	8192	NA	NA	NA
5	10	9600	8192	9500	8200	&g_mut
6	5	9700	8192	9500	8200	&g_mut
7	5	9750	8192	9500	8200	&g_mut
17	6	9800	8192	9750	8300	NA
18		9850	8350	NA	NA	NA

Table on bottom left shows line numbers and arbitrary epochs. T<sub>0</sub> is line number of thread 0. T<sub>1</sub> is line number of thread1. C<sub>x,y</sub> shows clock cache, where x represents the thread that owns the cache, and y is other thread. So, C<sub>0,0</sub> means thread 0's epoch, while C<sub>0,1</sub> is thread1's last-known epoch in thread 0's clock cache. When Thread #0 acquires mutex, it creates a synchronization variable. Clocks are not synchronized yet. When Thread #1 acquires the same mutex, it finds a new synchronization variable to synchronize clocks for. Then, it updates its clock cache by synchronizing with clocks on the synchronization object. That epoch is the epoch when Thread #0 released the mutex.

# TSan – Step-By-Step - Locked

```

1. #include <pthread.h>
2. int Global;
3. pthread_mutex_t g_mut = PTHREAD_MUTEX_INITIALIZER;
4. static void SetGlobal(int v) {
5.     pthread_mutex_lock(&g_mut);
6.     Global = v;
7.     pthread_mutex_unlock(&g_mut);
8. }
9. void *Thread1(void *x) {
10.    SetGlobal(42);
11.    return x;
12. }
13. int main() {
14.    pthread_t t;
15.    pthread_create(&t, NULL, Thread1, NULL);
16.    SetGlobal(43);
17.    pthread_join(t, NULL);
18.    return Global;
19. }
    
```

App memory

Global

Shadow memory

0 1 2 3

Field	0	1
Thread ID	0	1
R/W	W	W
Size (log)	2	2
Base Address	0	0
Epoch	9700	8300

T <sub>0</sub>	T <sub>1</sub>	C <sub>0,0</sub>	C <sub>0,1</sub>	C <sub>1,0</sub>	C <sub>1,1</sub>	S
14		8192	NA	NA	NA	NA
16		9500	8192	NA	NA	NA
5	10	9600	8192	9500	8200	&g_mut
6	5	9700	8192	9500	8200	&g_mut
7	5	9750	8192	9500	8200	&g_mut
17	6	9800	8192	9750	8300	NA
18		9850	8350	NA	NA	NA

Table on bottom left shows line numbers and arbitrary epochs. T<sub>0</sub> is line number of thread 0. T<sub>1</sub> is line number of thread1. C<sub>x,y</sub> shows clock cache, where x represents the thread that owns the cache, and y is other thread. So, C<sub>0,0</sub> means thread 0's epoch, while C<sub>0,1</sub> is thread1's last-known epoch in thread 0's clock cache. When Thread #0 acquires mutex, it creates a synchronization variable. Clocks are not synchronized yet. When Thread #1 acquires the same mutex, it finds a new synchronization variable to synchronize clocks for. Then, it updates its clock cache by synchronizing with clocks on the synchronization object. That epoch is the epoch when Thread #0 released the mutex.

# TSan – Step-By-Step - Locked

```

1. #include <pthread.h>
2. int Global;
3. pthread_mutex_t g_mut = PTHREAD_MUTEX_INITIALIZER;
4. static void SetGlobal(int v) {
5.     pthread_mutex_lock(&g_mut);
6.     Global = v;
7.     pthread_mutex_unlock(&g_mut);
8. }
9. void *Thread1(void *x) {
10.    SetGlobal(42);
11.    return x;
12. }
13. int main() {
14.    pthread_t t;
15.    pthread_create(&t, NULL, Thread1, NULL);
16.    SetGlobal(43);
17.    pthread_join(t, NULL);
18.    return Global;
19. }

```



Field	0	1	2	3
Thread ID	0	1		
R/W	W	W		
Size (log)	2	2		
Base Address	0	0		
Epoch	9700	8300		

T <sub>0</sub>	T <sub>1</sub>	C <sub>0,0</sub>	C <sub>0,1</sub>	C <sub>1,0</sub>	C <sub>1,1</sub>	S
14		8192	NA	NA	NA	NA
16		9500	8192	NA	NA	NA
5	10	9600	8192	9500	8200	&g_mut
6	5	9700	8192	9500	8200	&g_mut
7	5	9750	8192	9500	8200	&g_mut
17	6	9800	8192	9750	8300	NA
18		9850	8350	NA	NA	NA

Table on bottom left shows line numbers and arbitrary epochs. T<sub>0</sub> is line number of thread 0. T<sub>1</sub> is line number of thread1. C<sub>x,y</sub> shows clock cache, where x represents the thread that owns the cache, and y is other thread. So, C<sub>0,0</sub> means thread 0's epoch, while C<sub>0,1</sub> is thread1's last-known epoch in thread 0's clock cache. When Thread #0 acquires mutex, it creates a synchronization variable. Clocks are not synchronized yet. When Thread #1 acquires the same mutex, it finds a new synchronization variable to synchronize clocks for. Then, it updates its clock cache by synchronizing with clocks on the synchronization object. That epoch is the epoch when Thread #0 released the mutex.

# TSan – Step-By-Step - Locked

```

1. #include <pthread.h>
2. int Global;
3. pthread_mutex_t g_mut = PTHREAD_MUTEX_INITIALIZER;
4. static void SetGlobal(int v) {
5.     pthread_mutex_lock(&g_mut);
6.     Global = v;
7.     pthread_mutex_unlock(&g_mut);
8. }
9. void *Thread1(void *x) {
10.    SetGlobal(42);
11.    return x;
12. }
13. int main() {
14.    pthread_t t;
15.    pthread_create(&t, NULL, Thread1, NULL);
16.    SetGlobal(43);
17.    pthread_join(t, NULL);
18.    return Global;
19. }
    
```



Field	0	1	2	3
Thread ID	0	1		
R/W	W	W		
Size (log)	2	2		
Base Address	0	0		
Epoch	9700	8300		

T <sub>0</sub>	T <sub>1</sub>	C <sub>0,0</sub>	C <sub>0,1</sub>	C <sub>1,0</sub>	C <sub>1,1</sub>	S
14		8192	NA	NA	NA	NA
16		9500	8192	NA	NA	NA
5	10	9600	8192	9500	8200	&g_mut
6	5	9700	8192	9500	8200	&g_mut
7	5	9750	8192	9500	8200	&g_mut
17	6	9800	8192	9750	8300	NA
18		9850	8350	NA	NA	NA

Table on bottom left shows line numbers and arbitrary epochs. T<sub>0</sub> is line number of thread 0. T<sub>1</sub> is line number of thread1. C<sub>x,y</sub> shows clock cache, where x represents the thread that owns the cache, and y is other thread. So, C<sub>0,0</sub> means thread 0's epoch, while C<sub>0,1</sub> is thread1's last-known epoch in thread 0's clock cache. When Thread #0 acquires mutex, it creates a synchronization variable. Clocks are not synchronized yet. When Thread #1 acquires the same mutex, it finds a new synchronization variable to synchronize clocks for. Then, it updates its clock cache by synchronizing with clocks on the synchronization object. That epoch is the epoch when Thread #0 released the mutex.

# TSan – vptr Race

```
1. #include <semaphore.h>
2.
3. struct A {
4.     A() { sem_init(&sem_, 0, 0); }
5.     virtual void F() { }
6.     void Done() { sem_post(&sem_); }
7.     virtual ~A() {
8.         sem_wait(&sem_);
9.         sem_destroy(&sem_);
10.    }
11.    sem_t sem_;
12.};

14. struct B : A {
15.     virtual void F() { }
16.     virtual ~B() { }
17.};

20. static A *obj = new B;

22. void *Thread1(void *x) {
23.     obj->F();
24.     obj->Done();
25.     barrier_wait(&barrier);
26.     return NULL;
27.}

29. void *Thread2(void *x) {
30.     barrier_wait(&barrier);
31.     delete obj;
32.     return NULL;
33.}

35. int main() {
36.     barrier_init(&barrier, 2);
37.     pthread_t t[2];
38.     pthread_create(&t[0], NULL, Thread1, NULL);
39.     pthread_create(&t[1], NULL, Thread2, NULL);
40.     pthread_join(t[0], NULL);
41.     pthread_join(t[1], NULL);
42.}
```

```
=====
WARNING: ThreadSanitizer: data race on vptr (ctor/dtor vs virtual call) (pid=98695)
Write of size 8 at 0x7d040000f7f0 by thread T2:
```

In this example, `obj->F()` would want to read vtable of `obj`, whose VTable type is `A`. On the other hand, `Thread2` would want to destroy the object. To do that, it will first call `B`'s deleting destructor. This destructor will call `B`'s base destructor, which will call `A`'s base object destructor. This will write `A`'s vtable offset to vtable. In other words, while deleting an object, it makes a virtual function call. We don't know where the function would end up; in `A`, or in `B`. Because vtable pointer might have been overwritten, and it might point to `A`.

# TSan - Atomic Operations

- TSan understands atomics
- TSan does not understand inline assembly

TSan understands C11 and C++11 atomics. IIRC, it also understands some gcc intrinsics. But it does not understand inline assembly. So, if you build Qt with TSan, there is a good chance it that you will see tons of false-positives.

# TSan - Caveats

- Dependencies better be instrumented with TSan
  - May get away, if no synchronization happens in non-instrumented code
- Memory usage increases
- CPU usage increases
- Cannot instrument inline assembly

TSan generally requires you to compile all dependencies with TSan. But this depends on dependency. If a dependency does not synchronize to write things in your variables, you don't have to instrument it. Otherwise, you will see false-positives. TSan increases memory usage. CPU usage increases more, because tsan checks are expensive.

# Mac OS X

There is a little surprise; I am making TSan on Mac OS X! It's currently working, and 95% of tests run successfully on 64-bit Mac OS X.

# TSan on Mac OS X

- Changed how *all* sanitizers call their respective `_init` functions from module constructors
- TSan re-execs, like ASan, by appending TSan run-time to `DYLD_INSERT_LIBRARIES` environment variable
- TLS access is different on Darwin; needs a little dance -- both at thread startup, and shutdown
- Mac OS X dynamic linker is also using libc++
- Because TLS uses `pthread_set/getspecific`, `cur_thread()` was unavailable when called from a key's destructor (thread exit)
- Some tests are not supported on Darwin, and some are not fixed yet

Except ASan, other sanitizers, and sanitizercoverage are adding their respective init function directly to llvm global module ctors. This doesn't work with Darwin's run-time linker. Dynamic linker expects the initialization functions, that is module ctor, to be defined within the library itself. Besides, it looks a little strange to have different initialization sequence for each sanitizer. The only way to interpose functions on Darwin is to use `DYLD_INSERT_LIBRARIES` environment variable. We check this at startup, like ASan -- in fact, it's exactly the same code. If it doesn't contain tsan runtime library, we add TSan's runtime library to this environment variable, and exec again. TLS is initialized lazily in Darwin. That is, unless you access the address, it's not initialized. This isn't the biggest problem, however. It's using `pthread_get/setspecific` functions. At startup, we register function to dynamic linker that will be called when a TLS variable is allocated or freed. Just in case, we also enumerate TLS storage to discover whether the `cur_thread_placeholder` storage is allocated. At shutdown, the callback we registered for TLS deallocation notification will be called, right before the storage is freed -- so storage is still alive at this point, but should not be accessed directly, because that triggers resurrection of TLS variable, and this causes infinite recursion. So, we check each pointer callback is called for, and try to identify the one that points to `cur_thread_placeholder`. We then tell thread to finalize -- this is normally done from the destructor of a pthread key created at startup. But we cannot access to those thread local variables at this stage. Unfortunately, clock synchronization part always calls `cur_thread`, which will try to access TLS. We cannot allow this on Darwin. So, we pass `cur_thread` from top to whomever needs it. This prevents infinite loop caused by resurrection of TLS storage. There is another problem with this model. During thread shutdown, the order of destruction of pthread\_keys are unspecified. libc++'s `thread_local` values have a destructor function, which are instrumented by TSan. When TSan, if you remember, instruments function entries and exits. This is problematic, because, at thread shutdown destructor function for thread local key is Currently, 95% of tests

MSan

# MSan

- Diagnoses use of uninitialized data
- Significantly faster than dynamic binary instrumentation
- Has its dedicated pass

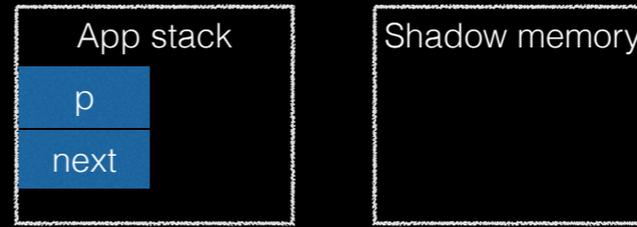
# MSan

- Instruments entire IR
- Application-shadow mapping is bit-to-bit
- Can track origins (i.e. where uninitialized value is obtained from)
- Accessing padding bytes — added due to alignment, is OK
- Can run in multithreaded code
- If misses a write (initialization), might start reporting false-positives
- Can handle `va_args`

MSan instruments more than just loads and stores, it instruments almost everything. Because value might be on register. Registers have no memory address. They cannot have shadow values in memory. Instead, MSan propagates shadow values in TLS to pass shadow information to and from functions.

# MSan Instrumentation

```
1. char *f(char *p) {
2.   // retval_tls[0] = param_tls[0];
3.   return ++p;
4. }
5. int main() {
6.   char *p; // uninitialized
7.   // __msan_poison_stack(&p, sizeof(char *));
8.   char *next =
9.   // __msan_poison_stack(&next, sizeof(char *));
10.  // param_tls[0] = *ShadowPtr(p);
11.  // retval_tls[0] = 0;
12.  // store call result in temporary
13.  f(p);
14.  // *ShadowPtr(next) = retval_tls[0];
15.  // store call result in 'next'
17.  return
18.  // if (*ShadowPtr(next) == 0)
19.  next[0];
20.  // else
21.  // __msan_warning
22. }
```

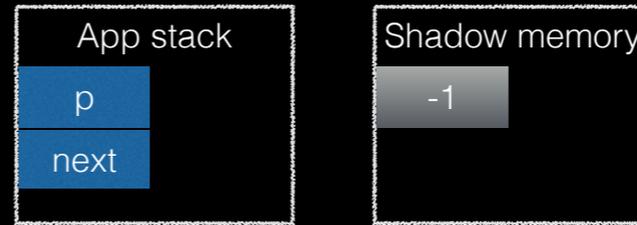


	0	1	...
Parameters			
Return Value			
Temporary			

This is a basic example. Last thing first; if the pointer was initialized, its shadow would be 0, not -1. -1 comes from the fact that uninitialized bits have shadow value of 1, and on 2's complement CPUs, this corresponds to integer representation of -1. MSan instrumentation is pretty complex. This illustrates summary of what's happening. If you see the actual instrumented code, without optimizations, you'd understand what I mean.

# MSan Instrumentation

```
1. char *f(char *p) {
2.   // retval_tls[0] = param_tls[0];
3.   return ++p;
4. }
5. int main() {
6.   char *p; // uninitialized
7.   // __msan_poison_stack(&p, sizeof(char *));
8.   char *next =
9.   // __msan_poison_stack(&next, sizeof(char *));
10.  // param_tls[0] = *ShadowPtr(p);
11.  // retval_tls[0] = 0;
12.  // store call result in temporary
13.  // f(p);
14.  // *ShadowPtr(next) = retval_tls[0];
15.  // store call result in 'next'
17.  return
18.  // if (*ShadowPtr(next) == 0)
19.  next[0];
20.  // else
21.  // __msan_warning
22. }
```

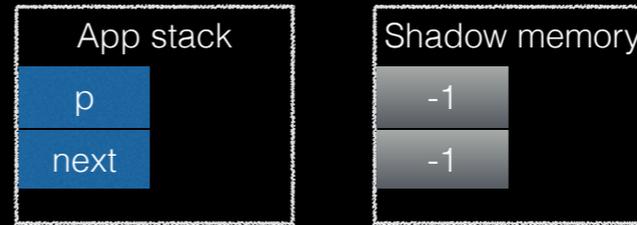


	0	1	...
Parameters			
Return Value			
Temporary			

This is a basic example. Last thing first; if the pointer was initialized, its shadow would be 0, not -1. -1 comes from the fact that uninitialized bits have shadow value of 1, and on 2's complement CPUs, this corresponds to integer representation of -1. MSan instrumentation is pretty complex. This illustrates summary of what's happening. If you see the actual instrumented code, without optimizations, you'd understand what I mean.

# MSan Instrumentation

```
1. char *f(char *p) {
2.   // retval_tls[0] = param_tls[0];
3.   return ++p;
4. }
5. int main() {
6.   char *p; // uninitialized
7.   // __msan_poison_stack(&p, sizeof(char *));
8.   char *next =
9.   // __msan_poison_stack(&next, sizeof(char *));
10.  // param_tls[0] = *ShadowPtr(p);
11.  // retval_tls[0] = 0;
12.  // store call result in temporary
13.  // f(p);
14.  // *ShadowPtr(next) = retval_tls[0];
15.  // store call result in 'next'
17.  return
18.  // if (*ShadowPtr(next) == 0)
19.  next[0];
20.  // else
21.  // __msan_warning
22. }
```

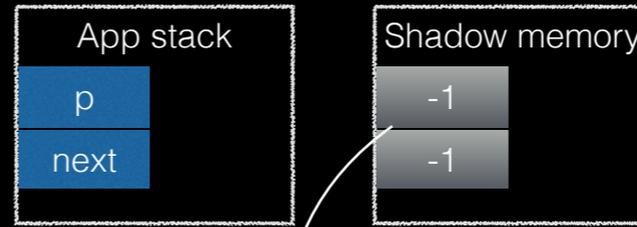


	0	1	...
Parameters			
Return Value			
Temporary			

This is a basic example. Last thing first; if the pointer was initialized, its shadow would be 0, not -1. -1 comes from the fact that uninitialized bits have shadow value of 1, and on 2's complement CPUs, this corresponds to integer representation of -1. MSan instrumentation is pretty complex. This illustrates summary of what's happening. If you see the actual instrumented code, without optimizations, you'd understand what I mean.

# MSan Instrumentation

```
1. char *f(char *p) {
2.   // retval_tls[0] = param_tls[0];
3.   return ++p;
4. }
5. int main() {
6.   char *p; // uninitialized
7.   // __msan_poison_stack(&p, sizeof(char *));
8.   char *next =
9.   // __msan_poison_stack(&next, sizeof(char *));
10.  // param_tls[0] = *ShadowPtr(p);
11.  // retval_tls[0] = 0;
12.  // store call result in temporary
13.  // f(p);
14.  // *ShadowPtr(next) = retval_tls[0];
15.  // store call result in 'next'
17.  return
18.  // if (*ShadowPtr(next) == 0)
19.  next[0];
20.  // else
21.  // __msan_warning
22. }
```

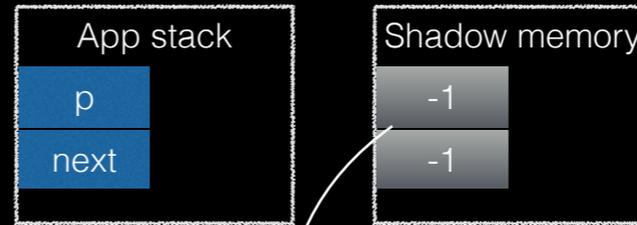


	0	1	...
Parameters	-1		
Return Value			
Temporary			

This is a basic example. Last thing first; if the pointer was initialized, its shadow would be 0, not -1. -1 comes from the fact that uninitialized bits have shadow value of 1, and on 2's complement CPUs, this corresponds to integer representation of -1. MSan instrumentation is pretty complex. This illustrates summary of what's happening. If you see the actual instrumented code, without optimizations, you'd understand what I mean.

# MSan Instrumentation

```
1. char *f(char *p) {  
2.   // retval_tls[0] = param_tls[0];  
3.   return ++p;  
4. }  
5. int main() {  
6.   char *p; // uninitialized  
7.   // __msan_poison_stack(&p, sizeof(char *));  
8.   char *next =  
9.   // __msan_poison_stack(&next, sizeof(char *));  
10.  // param_tls[0] = *ShadowPtr(p);  
11.  // retval_tls[0] = 0;  
12.  // store call result in temporary  
13.  f(p);  
14.  // *ShadowPtr(next) = retval_tls[0];  
15.  // store call result in 'next'  
  
17.  return  
18.  // if (*ShadowPtr(next) == 0)  
19.  next[0];  
20.  // else  
21.  // __msan_warning  
22. }
```



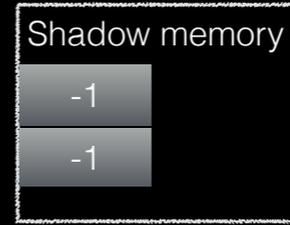
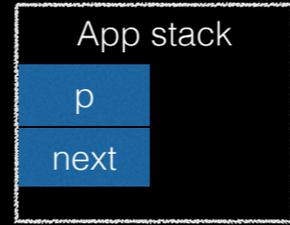
	0	1	...
Parameters	-1		
Return Value	0		
Temporary			

This is a basic example. Last thing first; if the pointer was initialized, its shadow would be 0, not -1. -1 comes from the fact that uninitialized bits have shadow value of 1, and on 2's complement CPUs, this corresponds to integer representation of -1. MSan instrumentation is pretty complex. This illustrates summary of what's happening. If you see the actual instrumented code, without optimizations, you'd understand what I mean.

# MSan Instrumentation



```
1. char *f(char *p) {
2.   // retval_tls[0] = param_tls[0];
3.   return ++p;
4. }
5. int main() {
6.   char *p; // uninitialized
7.   // __msan_poison_stack(&p, sizeof(char *));
8.   char *next =
9.   // __msan_poison_stack(&next, sizeof(char *));
10.  // param_tls[0] = *ShadowPtr(p);
11.  // retval_tls[0] = 0;
12.  // store call result in temporary
13.  // f(p);
14.  // *ShadowPtr(next) = retval_tls[0];
15.  // store call result in 'next'
17.  return
18.  // if (*ShadowPtr(next) == 0)
19.  next[0];
20.  // else
21.  // __msan_warning
22. }
```



	0	1	...
Parameters	-1		
Return Value	0		
Temporary			

This is a basic example. Last thing first; if the pointer was initialized, its shadow would be 0, not -1. -1 comes from the fact that uninitialized bits have shadow value of 1, and on 2's complement CPUs, this corresponds to integer representation of -1. MSan instrumentation is pretty complex. This illustrates summary of what's happening. If you see the actual instrumented code, without optimizations, you'd understand what I mean.

# MSan Instrumentation

```
1. char *f(char *p) {
2.   // retval_tls[0] = param_tls[0];
3.   return ++p;
4. }
5. int main() {
6.   char *p; // uninitialized
7.   // __msan_poison_stack(&p, sizeof(char *));
8.   char *next =
9.   // __msan_poison_stack(&next, sizeof(char *));
10.  // param_tls[0] = *ShadowPtr(p);
11.  // retval_tls[0] = 0;
12.  // store call result in temporary
13.  f(p);
14.  // *ShadowPtr(next) = retval_tls[0];
15.  // store call result in 'next'
17.  return
18.  // if (*ShadowPtr(next) == 0)
19.  next[0];
20.  // else
21.  // __msan_warning
22. }
```

App stack

p
next

Shadow memory

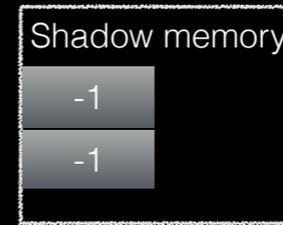
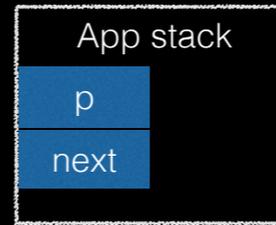
-1
-1

	0	1	...
Parameters	-1		
Return Value	-1		
Temporary			

This is a basic example. Last thing first; if the pointer was initialized, its shadow would be 0, not -1. -1 comes from the fact that uninitialized bits have shadow value of 1, and on 2's complement CPUs, this corresponds to integer representation of -1. MSan instrumentation is pretty complex. This illustrates summary of what's happening. If you see the actual instrumented code, without optimizations, you'd understand what I mean.

# MSan Instrumentation

```
1. char *f(char *p) {
2.   // retval_tls[0] = param_tls[0];
3.   return ++p;
4. }
5. int main() {
6.   char *p; // uninitialized
7.   // __msan_poison_stack(&p, sizeof(char *));
8.   char *next =
9.   // __msan_poison_stack(&next, sizeof(char *));
10.  // param_tls[0] = *ShadowPtr(p);
11.  // retval_tls[0] = 0;
12.  // store call result in temporary
13.  // f(p);
14.  // *ShadowPtr(next) = retval_tls[0];
15.  // store call result in 'next'
17.  return
18.  // if (*ShadowPtr(next) == 0)
19.  next[0];
20.  // else
21.  // __msan_warning
22. }
```

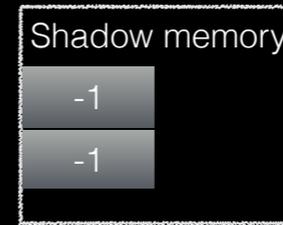
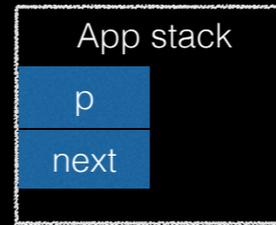


	0	1	...
Parameters	-1		
Return Value	-1		
Temporary	p + 1		

This is a basic example. Last thing first; if the pointer was initialized, its shadow would be 0, not -1. -1 comes from the fact that uninitialized bits have shadow value of 1, and on 2's complement CPUs, this corresponds to integer representation of -1. MSan instrumentation is pretty complex. This illustrates summary of what's happening. If you see the actual instrumented code, without optimizations, you'd understand what I mean.

# MSan Instrumentation

```
1. char *f(char *p) {  
2.   // retval_tls[0] = param_tls[0];  
3.   return ++p;  
4. }  
5. int main() {  
6.   char *p; // uninitialized  
7.   // __msan_poison_stack(&p, sizeof(char *));  
8.   char *next =  
9.   // __msan_poison_stack(&next, sizeof(char *));  
10.  // param_tls[0] = *ShadowPtr(p);  
11.  // retval_tls[0] = 0;  
12.  // store call result in temporary  
13.  f(p);  
14.  // *ShadowPtr(next) = retval_tls[0];  
15.  // store call result in 'next'  
  
17.  return  
18.  // if (*ShadowPtr(next) == 0)  
19.  next[0];  
20.  // else  
21.  // __msan_warning  
22. }
```

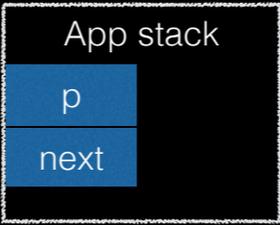


	0	1	...
Parameters	-1		
Return Value	-1		
Temporary			

This is a basic example. Last thing first; if the pointer was initialized, its shadow would be 0, not -1. -1 comes from the fact that uninitialized bits have shadow value of 1, and on 2's complement CPUs, this corresponds to integer representation of -1. MSan instrumentation is pretty complex. This illustrates summary of what's happening. If you see the actual instrumented code, without optimizations, you'd understand what I mean.

# MSan Instrumentation

```
1. char *f(char *p) {  
2.   // retval_tls[0] = param_tls[0];  
3.   return ++p;  
4. }  
5. int main() {  
6.   char *p; // uninitialized  
7.   // __msan_poison_stack(&p, sizeof(char *));  
8.   char *next =  
9.   // __msan_poison_stack(&next, sizeof(char *));  
10.  // param_tls[0] = *ShadowPtr(p);  
11.  // retval_tls[0] = 0;  
12.  // store call result in temporary  
13.  f(p);  
14.  // *ShadowPtr(next) = retval_tls[0];  
15.  // store call result in 'next'  
  
17.  return  
18.  // if (*ShadowPtr(next) == 0)  
19.  next[0];  
20.  // else  
21.  // __msan_warning  
22. }
```



	0	1	...
Parameters	-1		
Return Value	-1		
Temporary			

This is a basic example. Last thing first; if the pointer was initialized, its shadow would be 0, not -1. -1 comes from the fact that uninitialized bits have shadow value of 1, and on 2's complement CPUs, this corresponds to integer representation of -1. MSan instrumentation is pretty complex. This illustrates summary of what's happening. If you see the actual instrumented code, without optimizations, you'd understand what I mean.

# MSan - Caveats

- Dependencies must be instrumented with MSan
- Cannot deal with inline assembly code
- Very complicated instrumentation

Currently doesn't run fully on OSX, but I'm on it, and I see light at the end of tunnel. MSan requires all dependencies to be instrumented with MSan, perhaps except C standard library. libc++ can be built with MSan. If it misses to instrument one write to a variable, it will start reporting false positives.

# Sanitizers & Optimizer

- UBSan checks are emitted before any optimizations
- Other sanitizers operate on partially optimized IR; more optimizations run after them
- MSan re-adds some optimization passes (for anything other than -O0), as it emits a complex instrumentation

lots of the straightforward instrumentation added by ASan is improved by the following passes.

# Sanitizer Feature Check

- Clang feature-testing macro can be used\*

```
#ifndef __has_feature(x)
#define __has_feature(x) (0)
#endif

// currently;
// address_sanitizer, memory_sanitizer,
// thread_sanitizer, and dataflow_sanitizer are valid
#if __has_feature(sanitizer-name)
// ...
#else
// ...
#endif
```

\* UBSan cannot be detected

# Tuning instrumentation

- Additional/expensive instrumentation or instrumentation settings can be adjusted during compilation (-mllvm -asan-xxx=...)
- Sanitizer flags control behavior during run-time (XASAN\_OPTIONS=verbosity=2:color=never:...)
- To disable instrumentation:
  - In code; `__attribute__((no_sanitize_sanitizer-name))`
  - *suppressions* file can be specified to sanitizer at run-time, through environment variable (XASAN\_OPTIONS=suppressions=file)

There is also weak symbols for providing sanitizer default options. IIRC, they have a prototype like this; `extern "C" const char *__asan_default_options()`. Note that on Mac OS X, weak symbols do not work; you need environment variable.

# Multiple Sanitizer Support

Only UBSan can be used with other sanitizers.

LSan+ASan is also a valid combination, but the former is silently ignored in presence of the latter.

After merging UBSan+ASan run-times, UBSan+MSan, and UBSan+TSan became invalid.

UBSan+MSan, and UBSan+TSan will be supported in near future (:

This is slightly complicated. Sometime in February, IIRC, Alexey sent email to llvm and clang mailing lists, saying that he is merging ASan and UBSan runtimes. This meant that UBSan could not be used together with other sanitizers. Because it's not possible to use sanitizers together. Because, they instrument different things, they have different shadow mappings, and so on. I have asked Alexey about this, and he said he is working on UBSan +OtherSanitizers. I have made this slide before his patch. Instead of removing that sentence, I wanted to explain timeline of events.

# Instrumented C++ Standard Library

- Standard libraries do not ship with instrumentation
- In LLVM, compiler-rt checks for presence of libc++, and can build with enabled sanitizers (e.g. msan and tsan)
- Building libstdc++ with sanitizers is trivial

# Availability

	GCC	Clang	asan	msan	tsan	ubsan	lsan
Linux x86_64	● ● ●	● ● ● ● ●					
Linux ARM/Android	●	● ●					
Linux mips64/Android	●	● ● ●					
Mac OS X	●	● ½ ● ●					
iOS Simulator x86	●	● ●					
FreeBSD x86_64	● ?	● ● ● ?					
FreeBSD ARM	●	● ●					
Windows x86_64	?	? ?					

I am unsure about GCC, and am sorry for that. My excuse is very weak, but I'll say it, anyway; I don't have disk space available on my work machine, and at home, I am running a Mac Pro 1,1 with 7GB memory; I didn't want to spend 1 week trying to compile and test GCC. My bad! Similarly, I didn't have access to any Windows machine, and the one I had access to didn't have any toolchain.

Test it  
properly  
with sanitizers!

May the flame-throwing-bug-sanitizing-dragon spits flame at your bugs. But you must ask him, nicely, from your tests!



May the flame-throwing-bug-sanitizing-dragon spits flame at your bugs. But you must ask him, nicely, from your tests!

# Questions