Optimizing MemorySanitizer

Ideas, Victories, and Dead Ends

Gui Andrade

uninitialized memory

MSAN

What is MemorySanitizer?

MSAN is a compiler pass, it protects you from using uninitialized memory

Uninitialized memory causes nasty bugs!

bugs

vulnerabilities

undefined behavior

MSAN instruments your code with *shadow* memory that tags init status

How does Google use MSAN?

Chromium testing

User-facing memory vulns are a disaster!

Google3 testing

Business-facing memory vulns are a disaster!

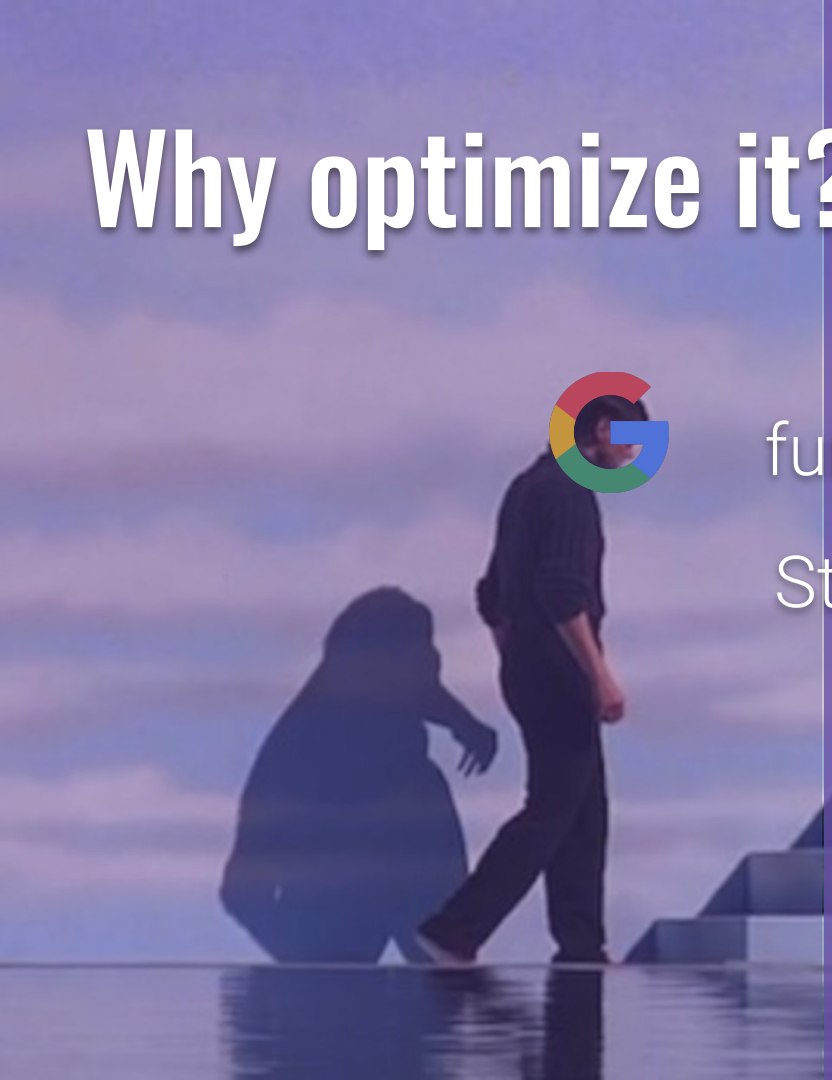
MemorySanitizer

oss-fuzz

OSS-Fuzz

Adaptive checking for bugs in open-source software

Why optimize it? Space



Large binaries get inflated even further with MSAN (typically 3-5x).

Standard x86_64 memory models address code within 2GB region

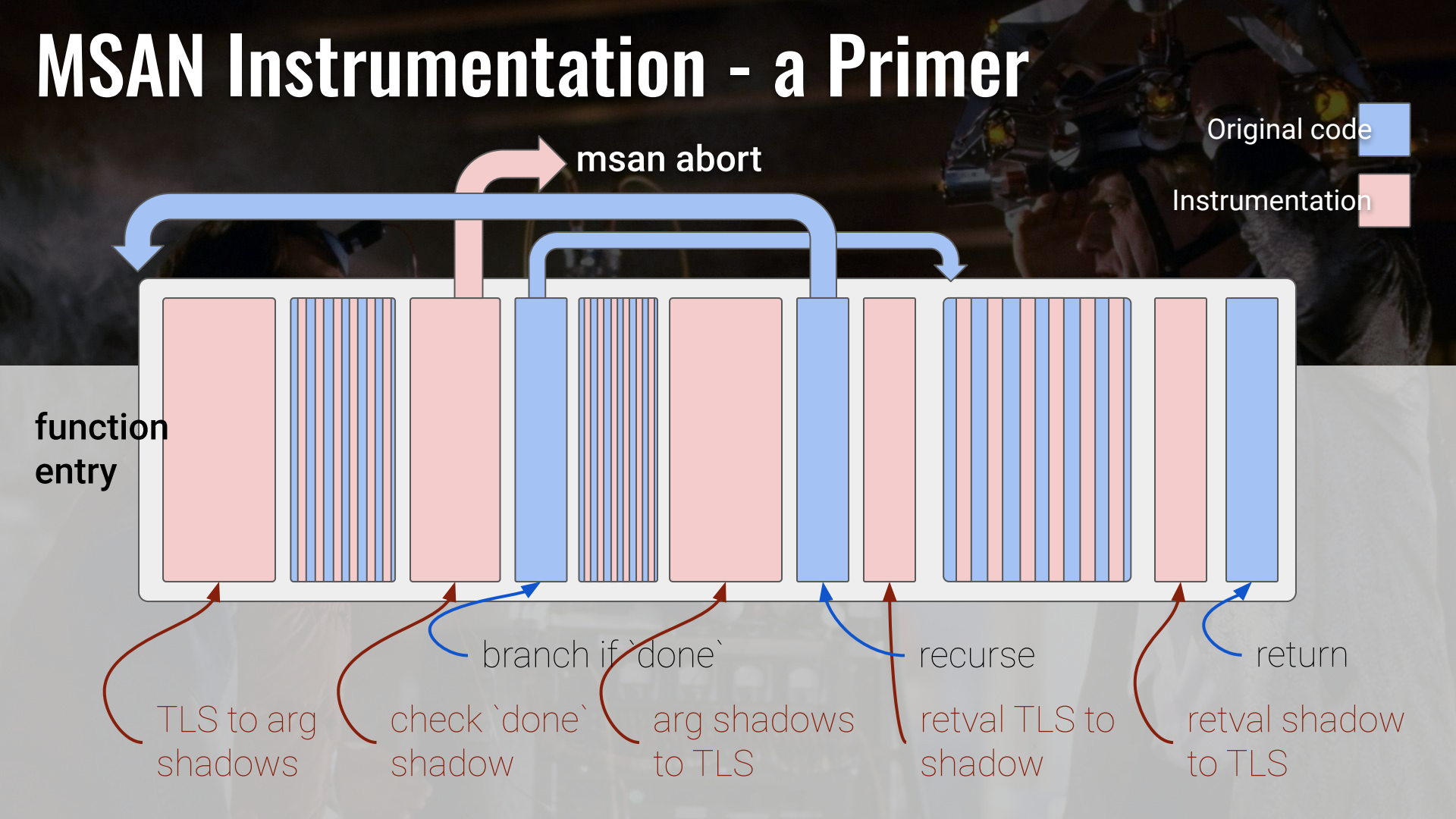
Large Google binaries + MSAN may approach/exceed 2GB

Why optimize it? Runtime

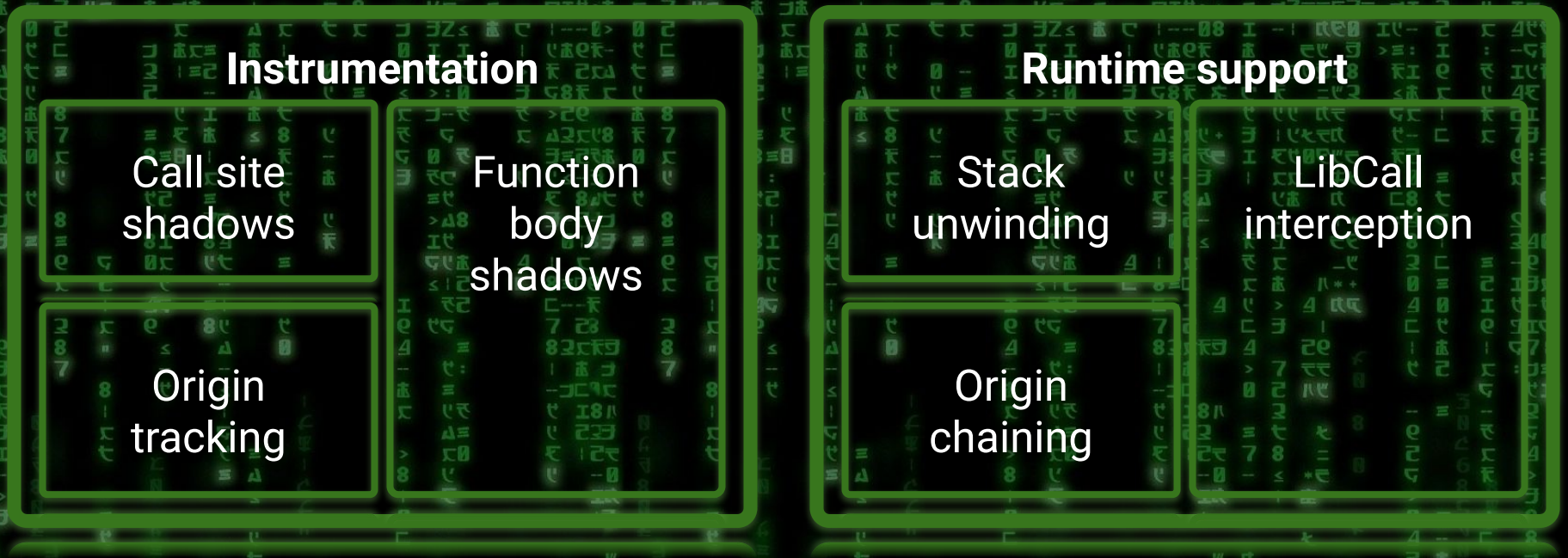
Instrumented Google3 testing costs $$ in compute resources

Instrumented binary runtime directly affects fuzzing ops/second. In the long run, faster binary => more bugs caught!

MSAN Instrumentation - a Primer



Optimizing MSAN = Cutting Overhead



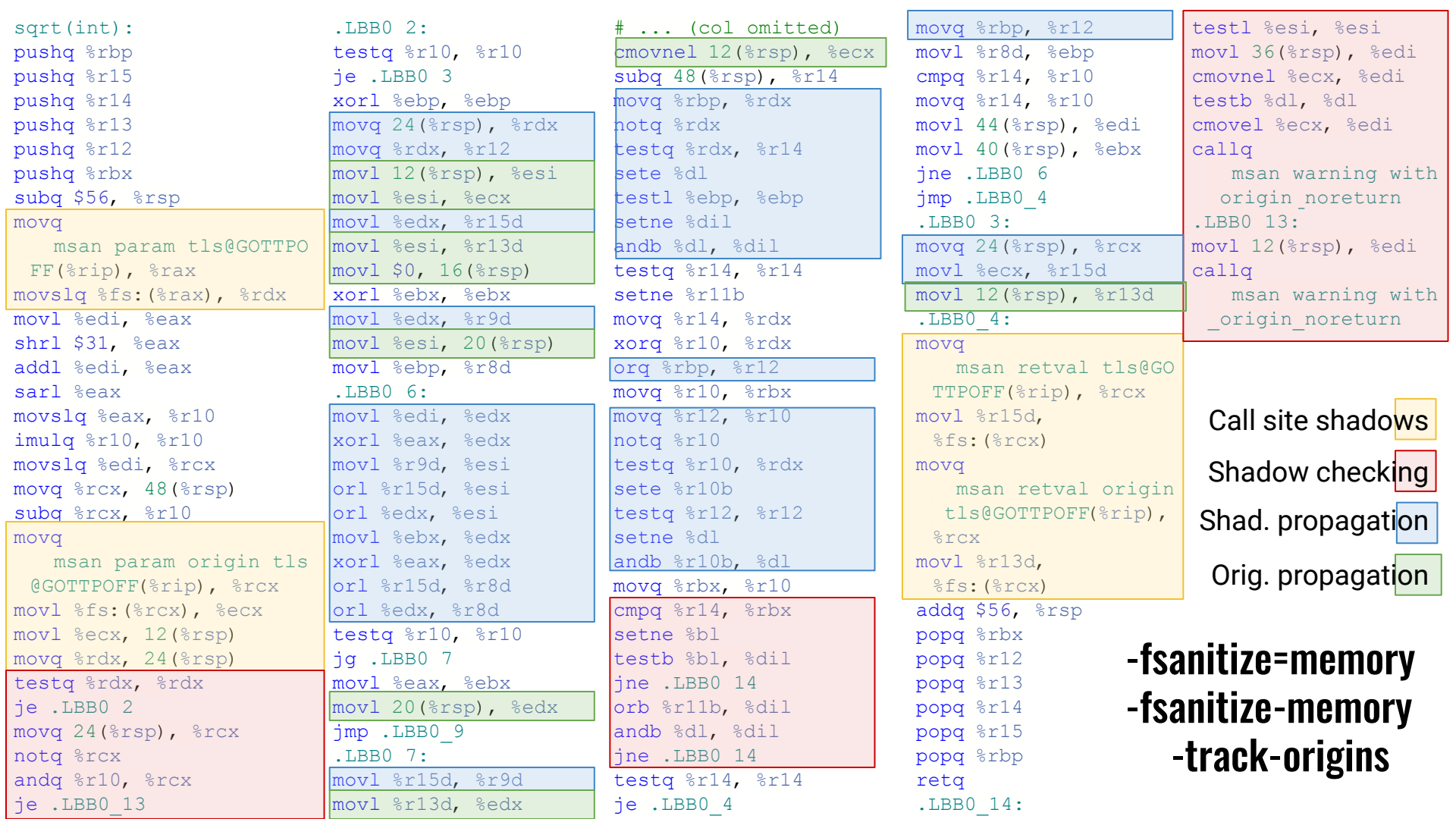
Note: discussing optimizations for MSAN+origin tracking

Case Study - Binary Search Sqrt

```
sqrt(int):
mov eax, edi
shr eax, 31
add eax, edi
sar eax
movsxd rdx, eax
imul rdx, rdx
movsxd r8, edi
sub rdx, r8
je .LBB0_7
xor esi, esi
.LBB0_2:
test rdx, rdx
jg .LBB0_3
mov esi, eax
jmp .LBB0_5
.LBB0_3:
mov edi, eax
.LBB0_5:
mov ecx, edi
sub ecx, esi
mov eax, ecx
shr eax, 31
add eax, ecx
sar eax
add eax, esi
movsxd rcx, eax
imul rcx, rcx
sub rcx, r8
je .LBB0_7
cmp rdx, rcx
mov rdx, rcx
jne .LBB0_2
.LBB0_7:
ret
```

```
int sqrt(int x) {
  int min_r = 0, max_r = x;
  long long old_loss, loss = 0;
  int r;

  #pragma nounroll
  while (true) {
    r = (max_r - min_r) / 2 + min_r;
    old_loss = loss;
    loss = (long long)r * r - x;
    if (!loss || old_loss == loss)
      return r;
    if (loss > 0) {
      max_r = r;
    } else {
      min_r = r;
    }
  }
  return r;
}
```

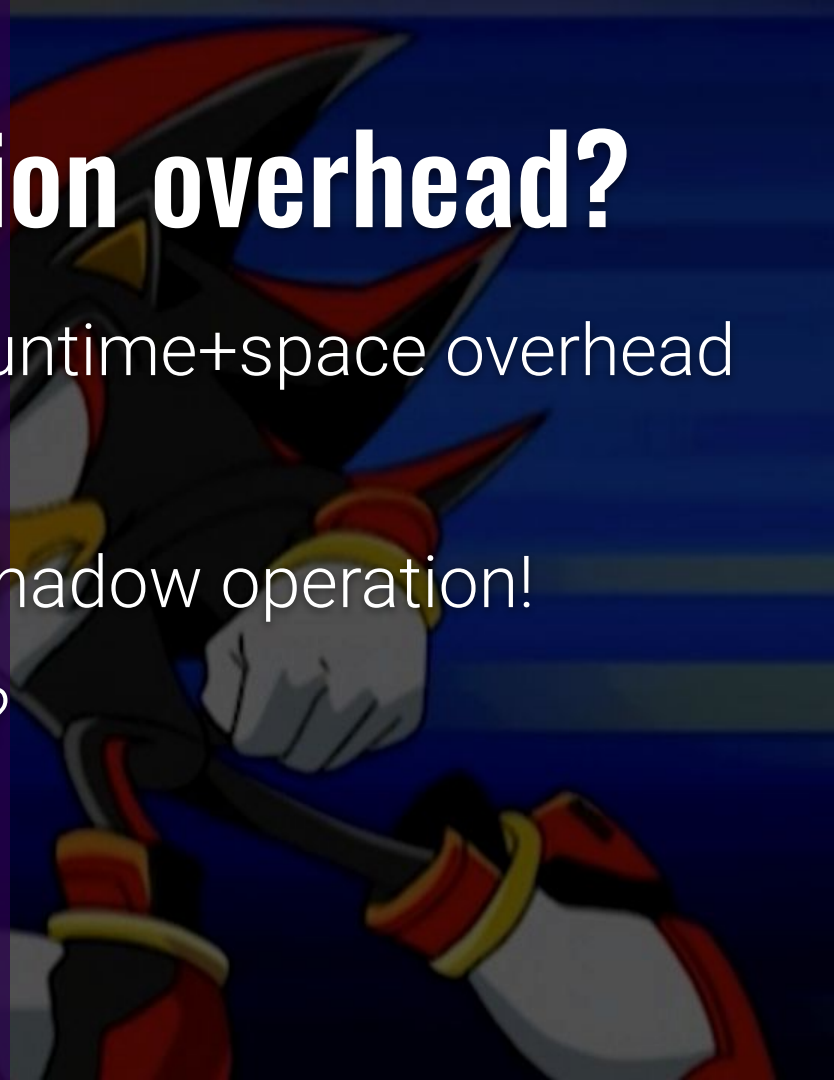

sqrt(int):
pushq %rbp
pushq %r15
pushq %r14
pushq %r13
pushq %r12
pushq %rbx
subq $56, %rsp
movq
msan param tls@GOTTPO
FF(%rip), %rax
movslq %fs:(%rax), %rdx
movl %edi, %eax
shrl $31, %eax
addl %edi, %eax
sarl %eax
movslq %eax, %r10
imulq %r10, %r10
movslq %edi, %rcx
movq %rcx, 48(%rsp)
subq %rcx, %r10
movq
msan param origin tls
@GOTTPOFF(%rip), %rcx
movl %fs:(%rcx), %ecx
movl %ecx, 12(%rsp)
movq %rdx, 24(%rsp)
testq %rdx, %rdx
je .LBB0 2
movq 24(%rsp), %rcx
notq %rcx
andq %r10, %rcx
je .LBB0_13
.LBB0 2:
testq %r10, %r10
je .LBB0 3
xorl %ebp, %ebp
movq 24(%rsp), %rdx
movq %rdx, %r12
movl 12(%rsp), %esi
movl %esi, %ecx
movl %edx, %r15d
movl %esi, %r13d
movl $0, 16(%rsp)
xorl %ebx, %ebx
movl %edx, %r9d
movl %esi, 20(%rsp)
movl %ebp, %r8d
.LBB0 6:
movl %edi, %edx
xorl %eax, %edx
movl %r9d, %esi
orl %r15d, %esi
orl %edx, %esi
movl %ebx, %edx
xorl %eax, %edx
orl %r15d, %r8d
orl %edx, %r8d
testq %r10, %r10
jg .LBB0 7
movl %eax, %ebx
movl 20(%rsp), %edx
jmp .LBB0_9
.LBB0 7:
movl %r15d, %r9d
movl %r13d, %edx
# ... (col omitted)
cmovnel 12(%rsp), %ecx
subq 48(%rsp), %r14
movq %rbp, %rdx
notq %rdx
testq %rdx, %r14
sete %dl
testl %ebp, %ebp
setne %dil
andb %dl, %dil
testq %r14, %r14
setne %r11b
movq %r14, %rdx
xorq %r10, %rdx
orq %rbp, %r12
movq %r10, %rbx
movq %r12, %r10
notq %r10
testq %r10, %rdx
sete %r10b
testq %r12, %r12
setne %dl
andb %r10b, %dl
movq %rbx, %r10
cmpq %r14, %rbx
setne %bl
testb %bl, %dil
jne .LBB0 14
orb %r11b, %dil
andb %dl, %dil
jne .LBB0 14
testq %r14, %r14
je .LBB0_4
movq %rbp, %r12
movl %r8d, %ebp
cmpq %r14, %r10
movq %r14, %r10
movl 44(%rsp), %edi
movl 40(%rsp), %ebx
jne .LBB0 6
jmp .LBB0_4
.LBB0 3:
movq 24(%rsp), %rcx
movl %ecx, %r15d
movl 12(%rsp), %r13d
.LBB0_4:
movq
msan retval tls@GO
TTPOFF(%rip), %rcx
movl %r15d,
%fs:(%rcx)
movq
msan retval origin
tls@GOTTPOFF(%rip),
%rcx
movl %r13d,
%fs:(%rcx)
addq $56, %rsp
popq %rbx
popq %r12
popq %r13
popq %r14
popq %r15
popq %rbp
retq
.LBB0_14:
testl %esi, %esi
movl 36(%rsp), %edi
cmovnel %ecx, %edi
testb %dl, %dl
cmovel %ecx, %edi
callq
msan warning with
origin_noreturn
.LBB0 13:
movl 12(%rsp), %edi
callq
msan warning with
_origin_noreturn
Call site shadows
Shadow checking
Shad. propagation
Orig. propagation
-fsanitize=memory
-fsanitize-memory
-track-origins

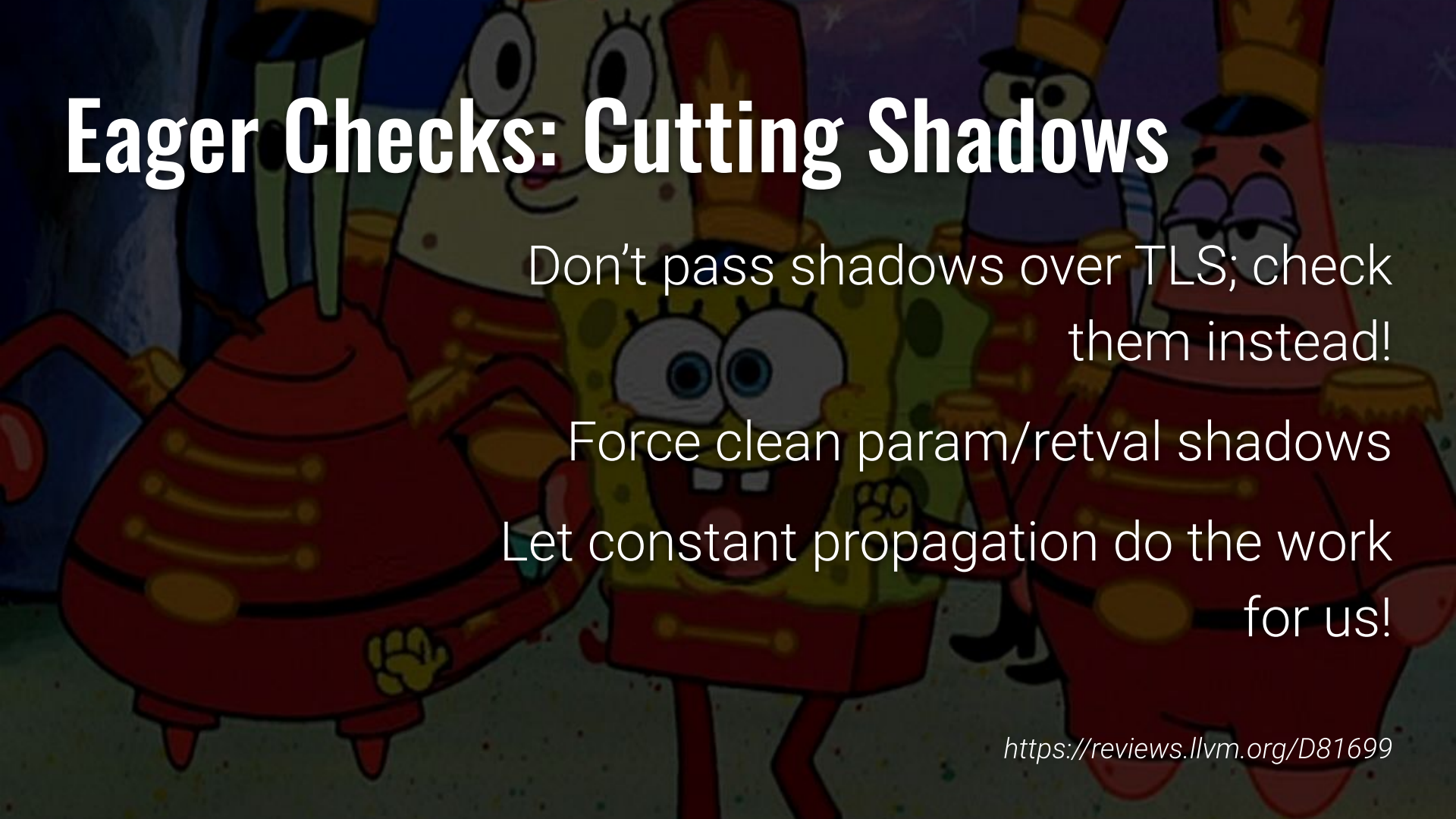
Cutting instrumentation overhead?

For non-trivial functions, most runtime+space overhead comes from shadow prop.

Every operation needs its own shadow operation!

Can we *factor out* shadow prop?





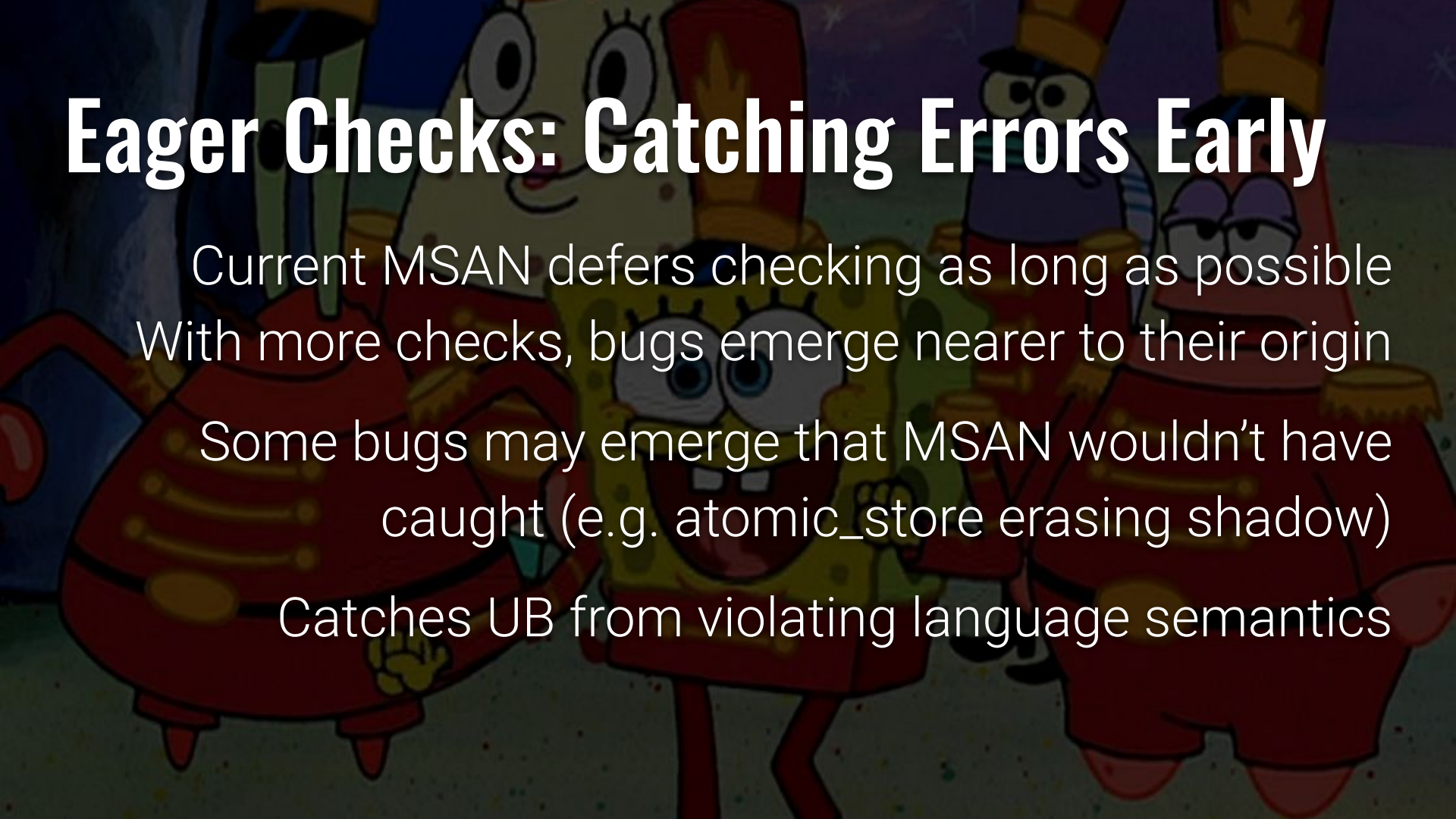
Eager Checks: Cutting Shadows

Don’t pass shadows over TLS; check them instead!

Force clean param/retval shadows

Let constant propagation do the work for us!

https://reviews.llvm.org/D81699



Eager Checks: Catching Errors Early

Current MSAN defers checking as long as possible
With more checks, bugs emerge nearer to their origin

Some bugs may emerge that MSAN wouldn’t have caught (e.g. atomic_store erasing shadow)

Catches UB from violating language semantics

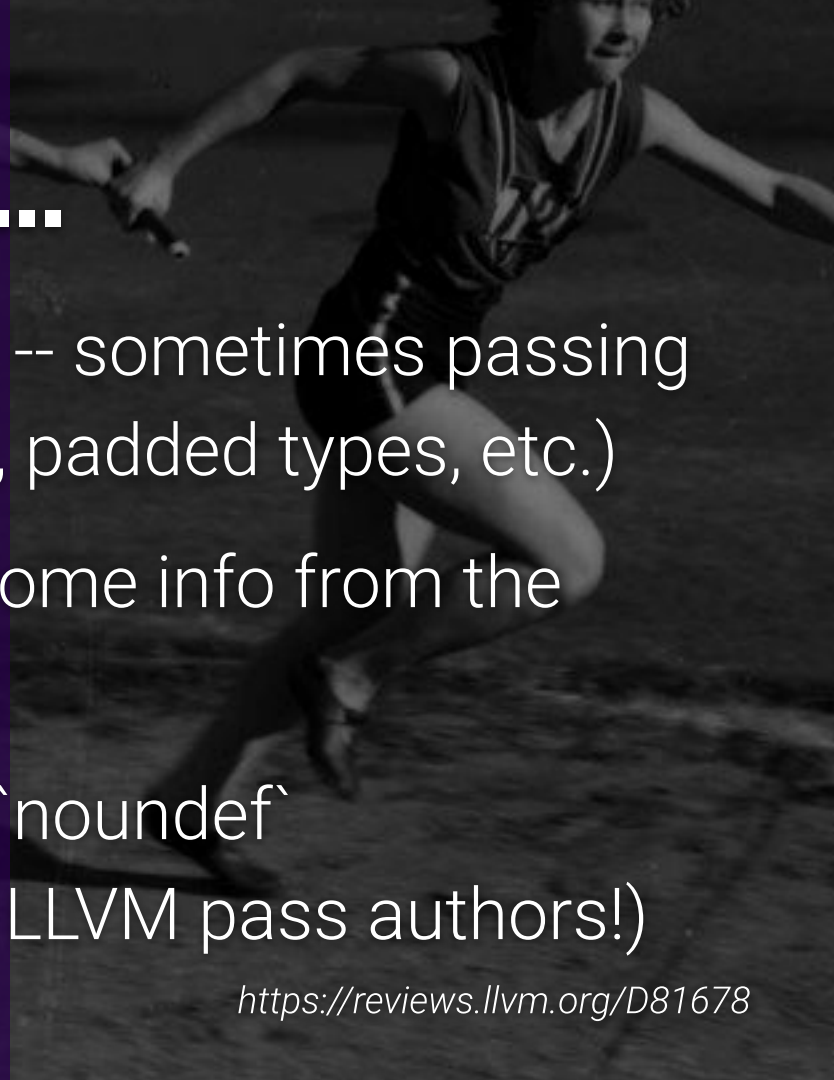
Easier said than done...

Can’t willy nilly check everything -- sometimes passing uninit values is legal! (retval in C, padded types, etc.)

Language-specific rules. Need some info from the frontend!

New LLVM parameter attribute: `noundef`
(actually more broadly useful to LLVM pass authors!)

https://reviews.llvm.org/D81678



Case Study - Binary Search Sqrt

With eager checking:

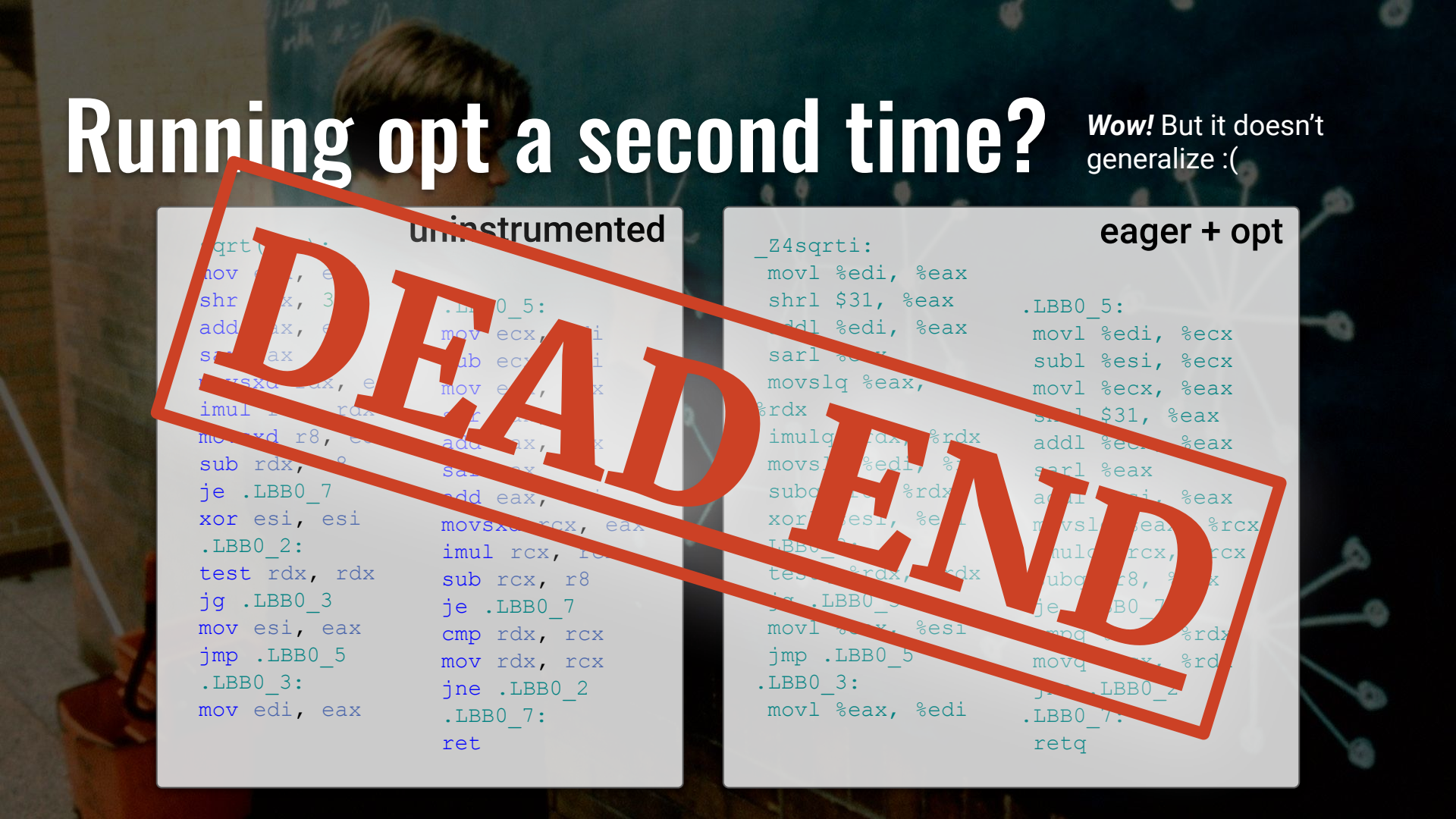
# of instructions: 153 => 132 (-13.7%)
 uninstrumented version: 30 instructions

Object file size: 641B => 545B (-15.0%)
 uninstrumented version: 134B

But! Theoretically all shadow prop should go away...

Running opt a second time?

Wow! But it doesn't generalize :(



Extra wins on the margins #1

MSAN checked shadows with a call to __msan_warning and passed origins over TLS to the runtime

Checks were now extremely common, TLS store insts are bulky => have __msan_warning take origin as a formal parameter instead

Measured 5% space savings on Clang

https://reviews.llvm.org/D81700

Extra wins on the margins? #2

DEAD END

MSAN poisons any local variables on the stack, usually inline (e.g. with memset)

Could force it to poison var + set origin with one rt call (outlining alloca poisoning)

1.5% space savings… 20% increase in runtime 🤔

Extra wins on the margins #3

Large stack variables call __msan_set_alloca_origin to register their stack trace

If variable immediately stored to, this call is redundant

Make MSAN optimize it away?

Variable space savings; 8% for grep, <1% for Clang

https://reviews.llvm.org/D83595

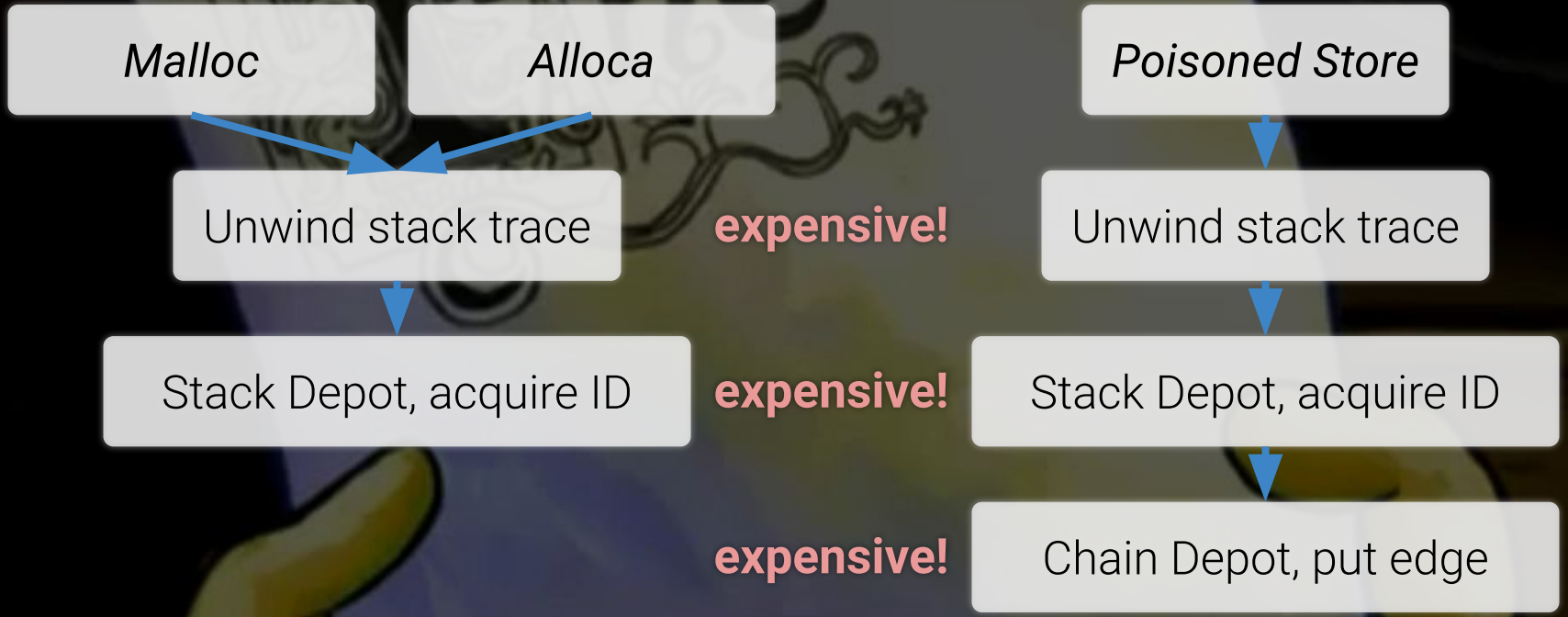
Bookkeeping and performance

MSAN has to track stack traces where uninit memory comes from, and chain together their “origins”

Some programs spend large % of runtime on this bookkeeping (for Clang, ~30%)

% depends on size of program, amount of mallocs done, amount of large allocas done

Bookkeeping process

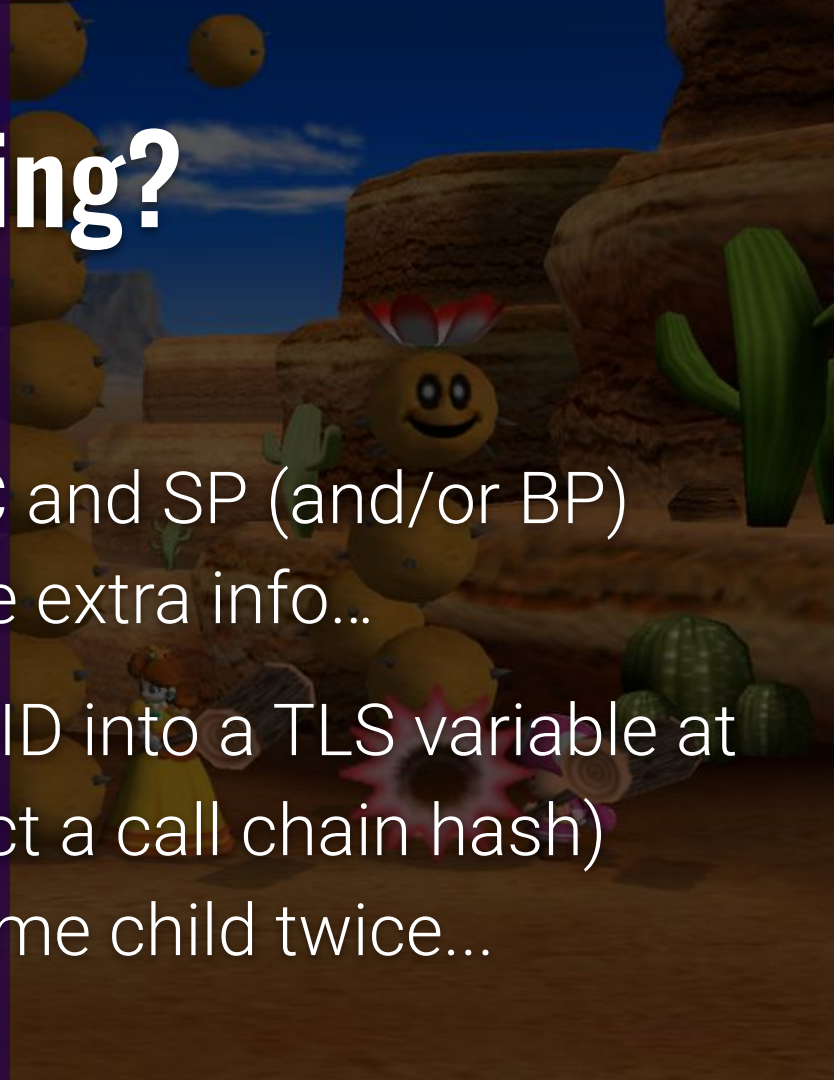


Cached stack unwinding?

What's our *index*?

Idea 1: some combination of PC and SP (and/or BP)
Too many collisions! Need some extra info…

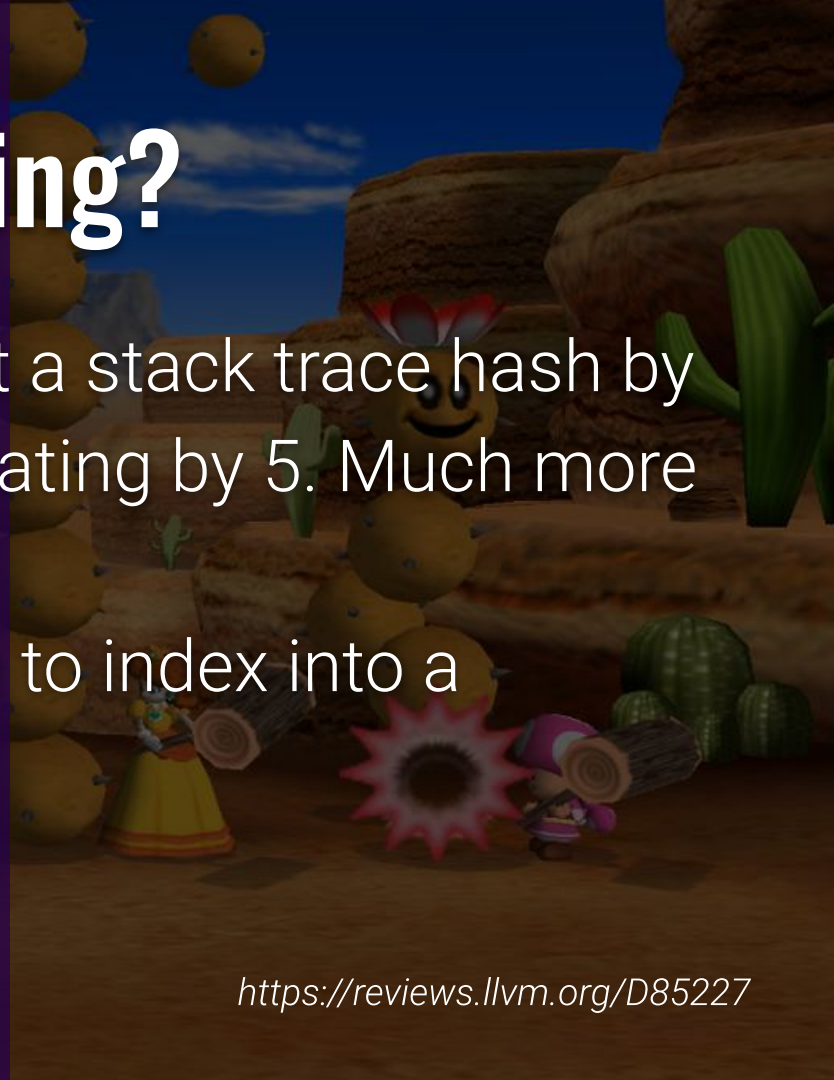
Idea 2: every callee XORs its UUID into a TLS variable at entry, and again at exit (construct a call chain hash)
Collision if same parent calls same child twice...



Cached stack unwinding?

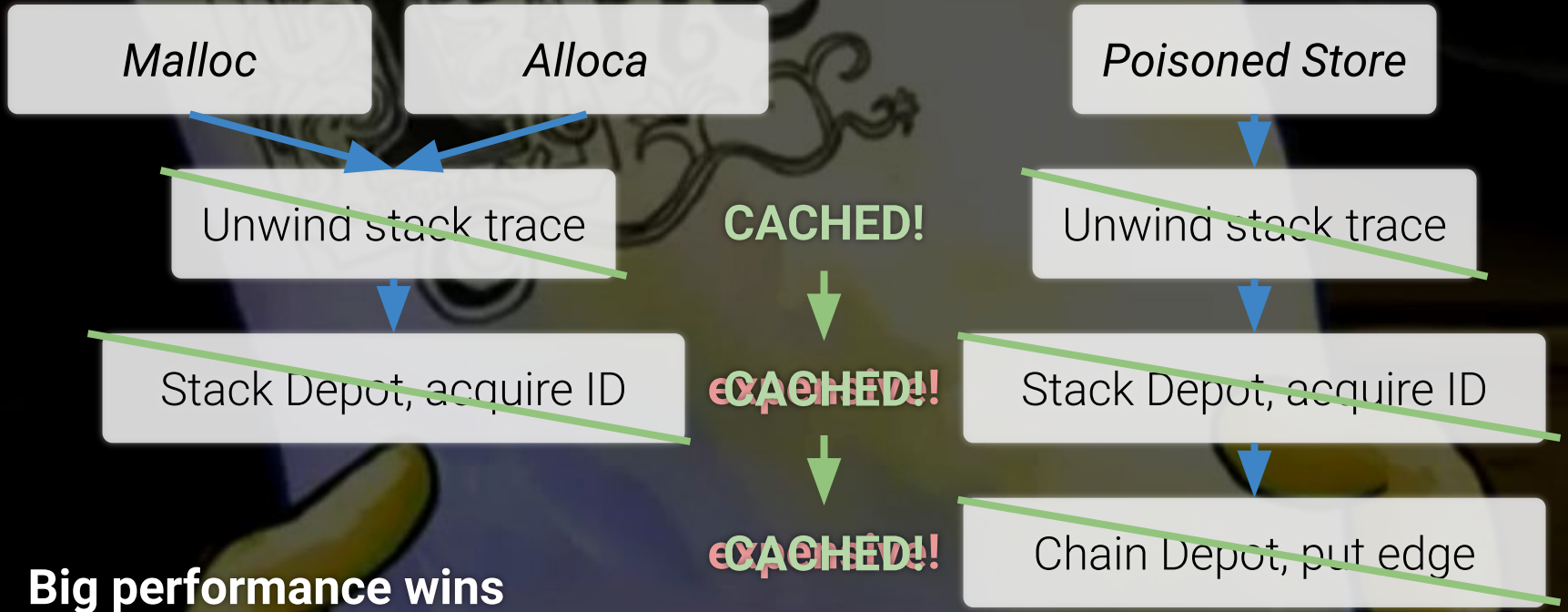
Idea 3: same idea, but construct a stack trace hash by XORing return-addr and then rotating by 5. Much more precise!

Then we can use this *trace hash* to index into a direct-mapped *trace cache*



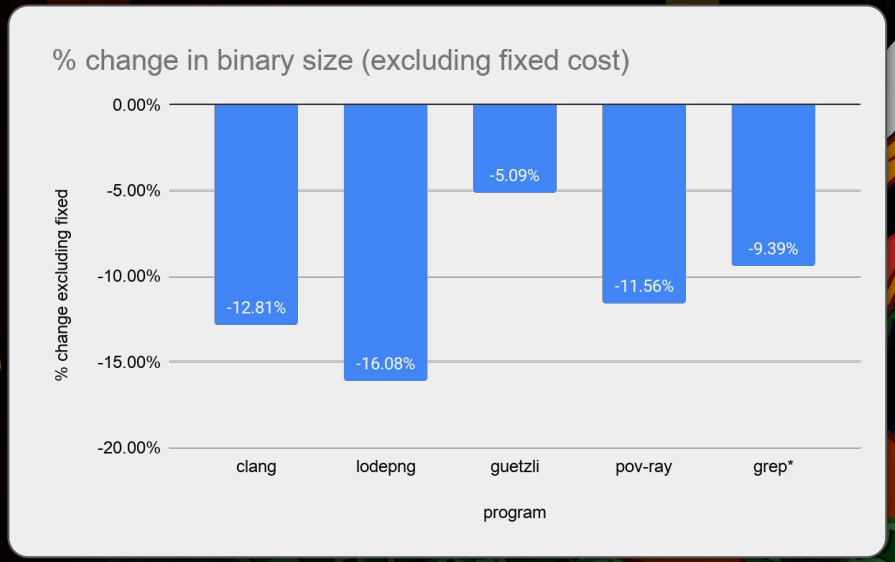
https://reviews.llvm.org/D85227

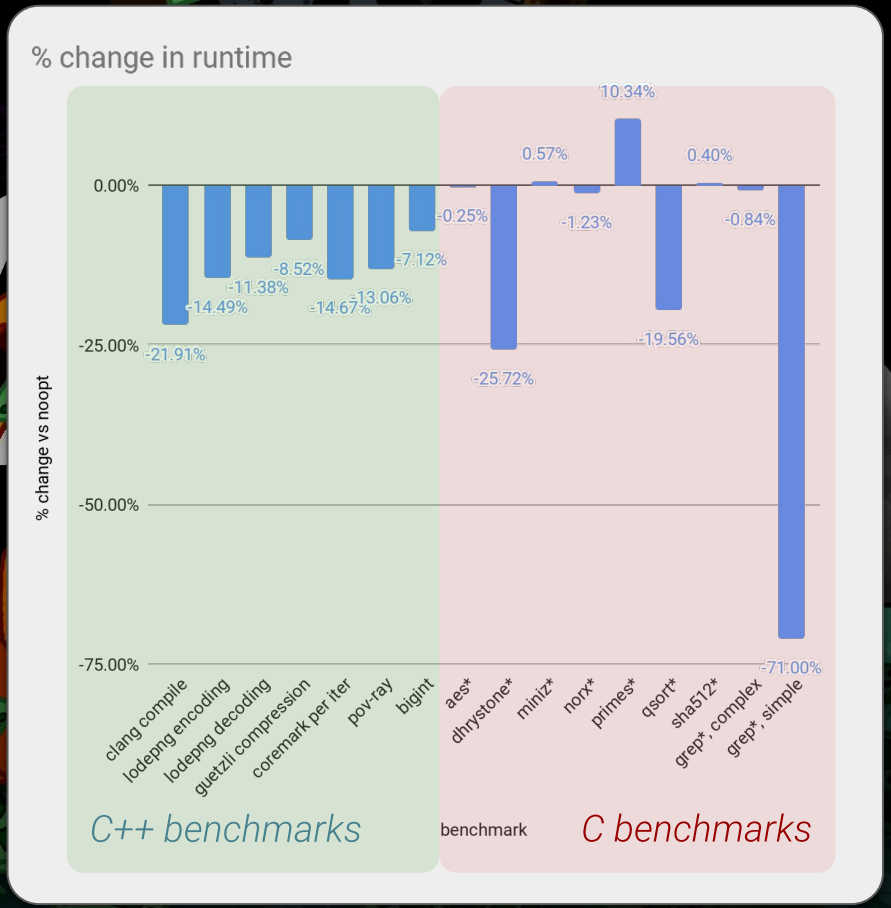
Cached bookkeeping process



Big performance wins for Clang, >17%

Combined Results

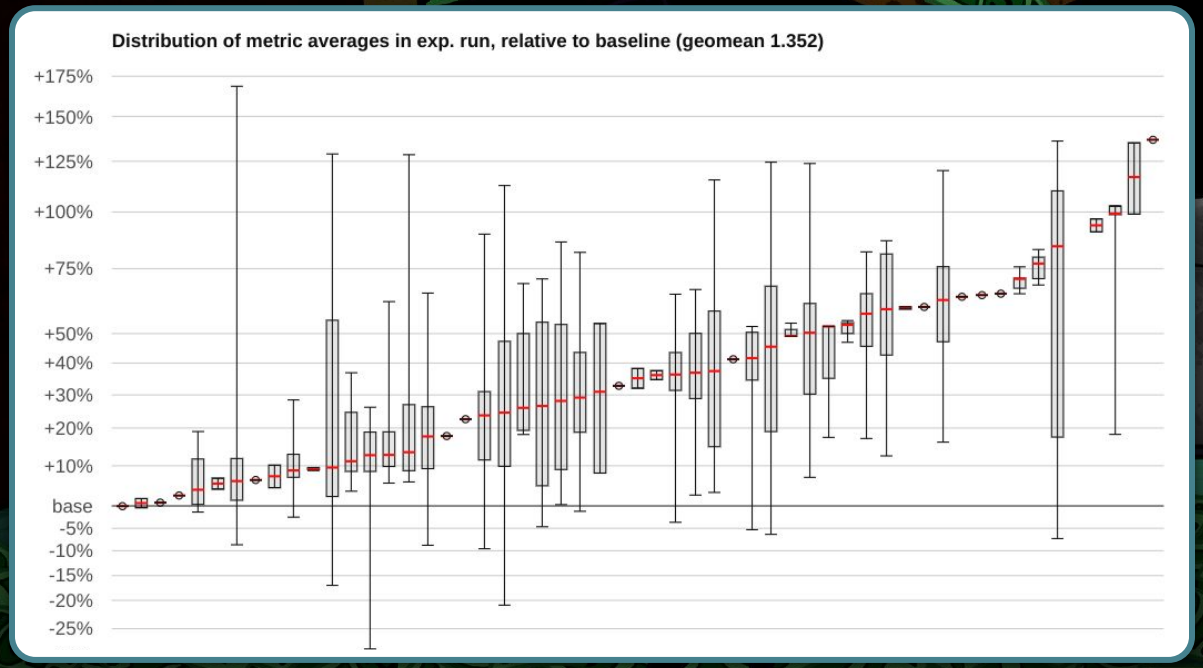




Eager checks most effective in C++

Cached traces best with large programs

Internal benchmarks? Promising!



Execution frequency benchmarks to evaluate compiler releases. Higher=better!

More Google-Specific Results

Chromium binary size

0.749 GB => 0.592 GB. **-21.1%**

Internal RPC benchmark

61k => 75.5k req/sec. **+23.7%**

Internal server binary size...

Reining in Binary Size

Google internal server binary

1.900 GB => 1.687 GB

Moderate savings: **-11.2%**

Enough to keep it comfortably in the 2GB code size limit!

(no origin tracking, as the binary would be too big)

Challenges

Sadly, optimizations don’t necessarily stack

Precise standard compliance for `noundef` rules

Community agreement around `noundef` changes

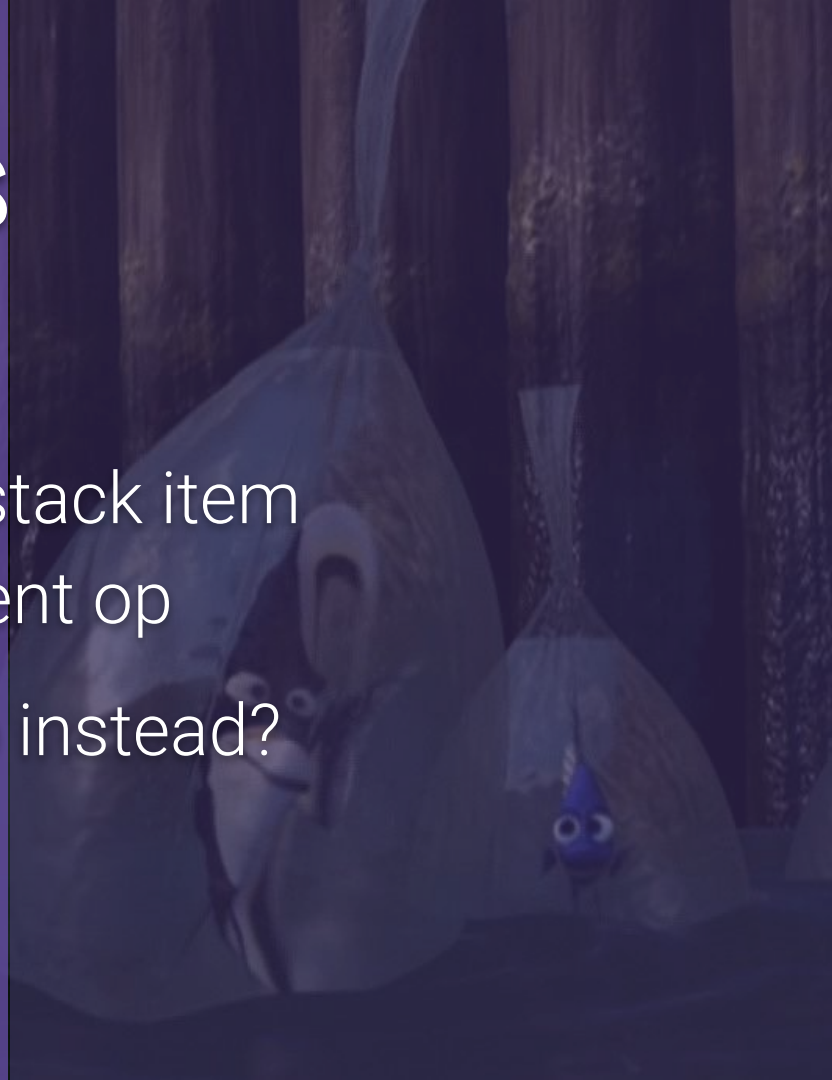
Integrating Clang/MSAN to ad-hoc build systems :(

Next steps, next ideas

Shadow frame pointer

Computing shadow pointer for stack item is a common and space-inefficient op

Reserve a register as shadow fp instead?



Next steps, next ideas

Noundef improvements

Certain classes of types are never `noundef` in the initial implementation (for simplicity)

Structs, coerced types: if these were marked noundef they could be checked in MSAN

Acknowledgements

Hosts **Evgenii Stepanov and Vitaly Buka**
Huge support and guidance!

Richard Smith, Johannes Doerfert, other reviewers
Patience and dedication to correctness

Rest of the dynamic tools team!