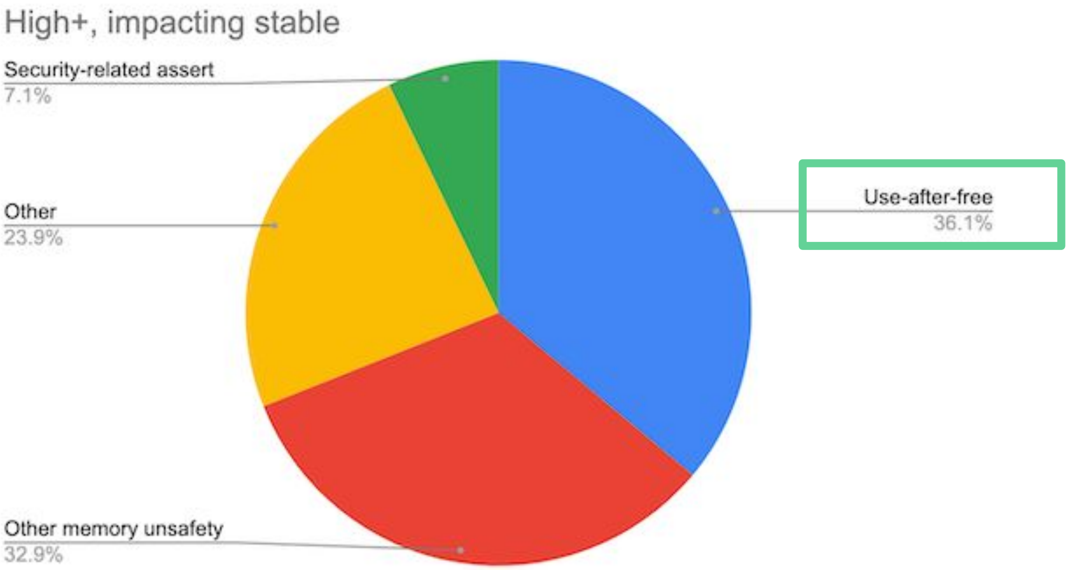
Lifetime Safety in Clang

LLVM Developers' Meeting 2025
Utkarsh Saxena (Google)
29 Oct, 2025

Temporal Memory Safety

It is undefined behaviour to access a memory after it has been deallocated or "freed".

Temporal Safety: Impact



Analysis based on *912* high or critical severity security bugs since 2015 *(in Chromium project).*
https://www.chromium.org/Home/chromium-security/memory-safety/

Temporal Safety: Examples

Use-after-free

```
int foo() {
    int* p;
    {
        std::unique_ptr<int> x = std::make_unique<int>(5);
        p = x.get();
    }  // 'x' destructed here.
    std::cout << *p; // use-after-free.
}
```

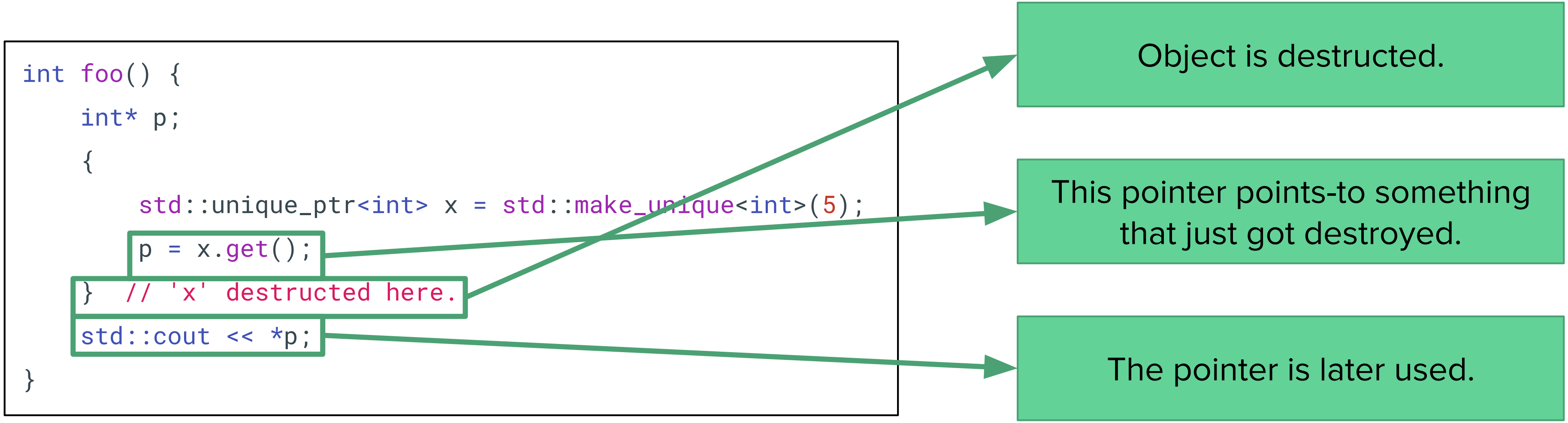
Lifetime Safety

An alias-based analysis

An intuitive path towards incremental compile-time temporal safety.

Programmer’s intuition

How does a programmer reason about Temporal Safety ?



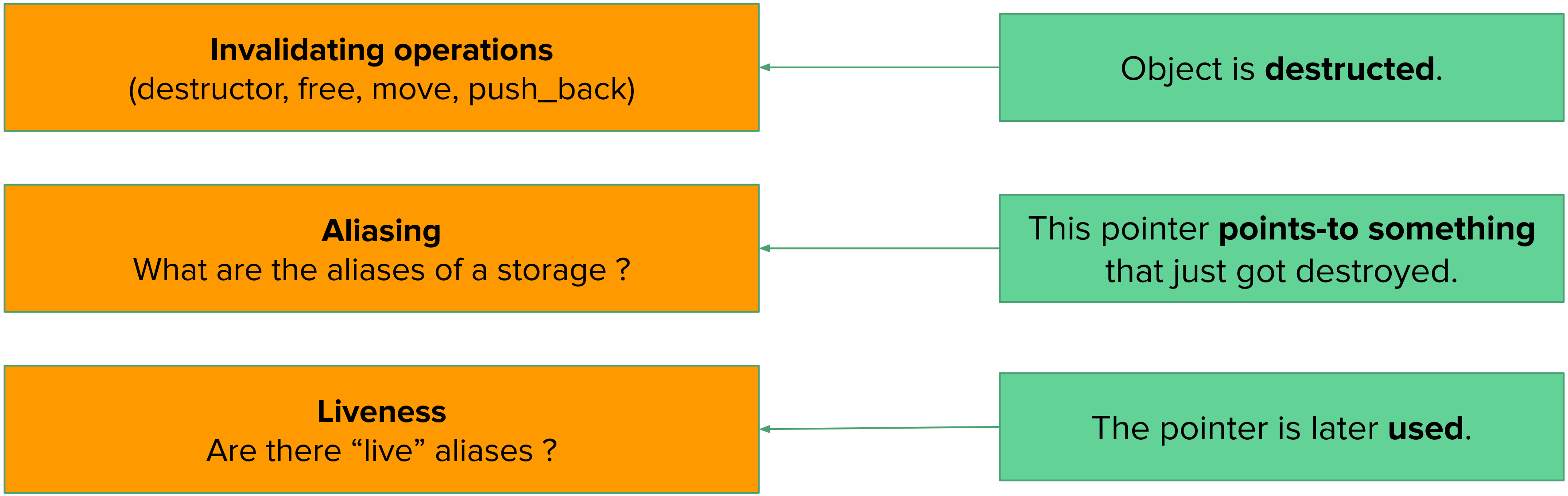
```
int foo() {
    int* p;
    {
        std::unique_ptr<int> x = std::make_unique<int>(5);
        p = x.get();
    }  // 'x' destructed here.
    std::cout << *p;
}
```

Object is destructed.

This pointer points-to something that just got destroyed.

The pointer is later used.

Programmer’s intuition



Invalidating operations
(destructor, free, move, push_back)

Object is **destructed**.

Aliasing
What are the aliases of a storage ?

This pointer **points-to something** that just got destroyed.

Liveness
Are there “live” aliases ?

The pointer is later **used**.

Invalidating operations

```
// Destructors.
{
    unique_ptr<int> x = make_unique<int>(5);
}  // 'x' destructed here.
```

Hidden behind **abstractions**: `push_back`, `clear()`

```
std::vector<int> v = {1, 2, 3, 4};
auto it = v.find(1);
v.push_back(5); // invalidates 'it'.
```

Aliasing

```
int foo() {
    int* p;
    {
        std::unique_ptr<int> x = std::make_unique<int>(5);
        p = x.get();
    }
    return *p;
}
```

Aliasing

```
int foo() {
    int* p;
    {
        std::unique_ptr<int> x = std::make_unique<int>(5);
        int *q = x.get();
        p = q;
    }
    return *p;
}
```

Aliasing

```
int foo() {
    int* p;
    {
        std::unique_ptr<int> x = std::make_unique<int>(5);
        int *q = x.get();
        int *r = q;
        int *s = r;
        int *t = s;
        p = s;
    }
    return *p;
}
```

Liveness

```
int foo() {
    int* p;
    {
        std::unique_ptr<int> x = std::make_unique<int>(5);
        p = x.get();
    }
    std::cout << *p;
}
```

Liveness

```
int foo() {
    int* p;
    {
        std::unique_ptr<int> x = std::make_unique<int>(5);
        p = x.get();
    }
    std::unique_ptr<int> y = std::make_unique<int>(42);
    p = y.get();
    std::cout << *p;
}
```

Not alive

Kills the previous value

Lifetime model

Loans

Represents the **act of borrowing** from a specific **memory location**.

Defined by

- **Where** it is created (the borrow site)
- **What** memory is borrowed

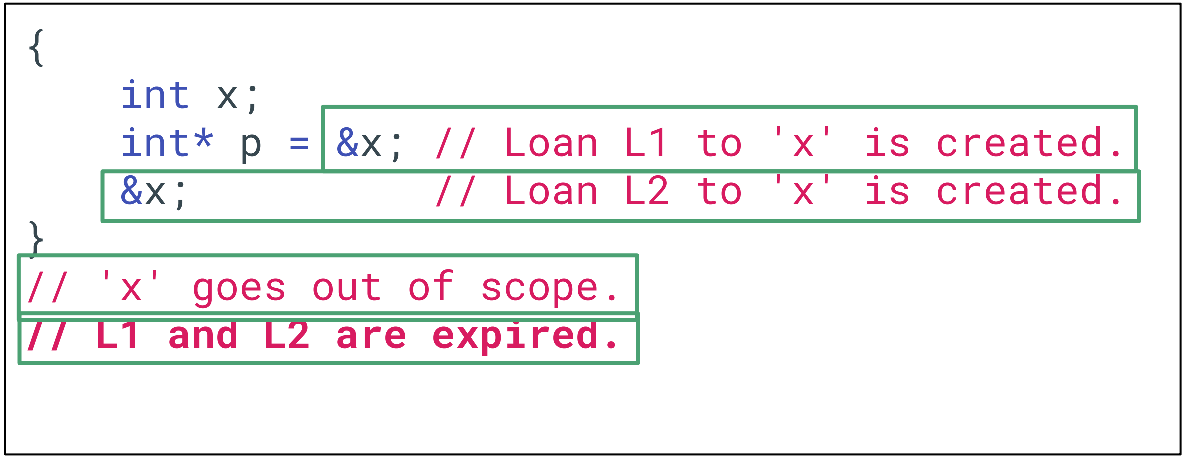
```
int x;

int* p = &x;
      // Loan L1 to 'x' is created.
```

Loans Expirations

Represents memory **invalidations**.

When a storage is invalidated, all loans to it expires.



```
{
    int x;
    int* p = &x; // Loan L1 to 'x' is created.
    &x;          // Loan L2 to 'x' is created.
}
// 'x' goes out of scope.
// L1 and L2 are expired.
```

Origins

Represents **aliasing**.

- Symbolic identifier associated with **pointer-like types**.
- **Set of all loans** that an entity can hold.

```
int* p; // int* ^O1
```

```
{
    int x;
    int* p; // int* ^O1
    p = &x; // Loan L1 to 'x'.
            // ^O1 = {L1}
}
// 'x' goes out of scope.
// L1 is expired.
```

Flow-sensitivity and subtyping rules

Represents the **flow-sensitive** nature of **aliasing.**

- Flow-sensitive subtyping rules
- Implies a **subset** constraint

```
int* p; // int* ^O1
int* q; // int* ^O2
q = p;  // O2 ← O1
```

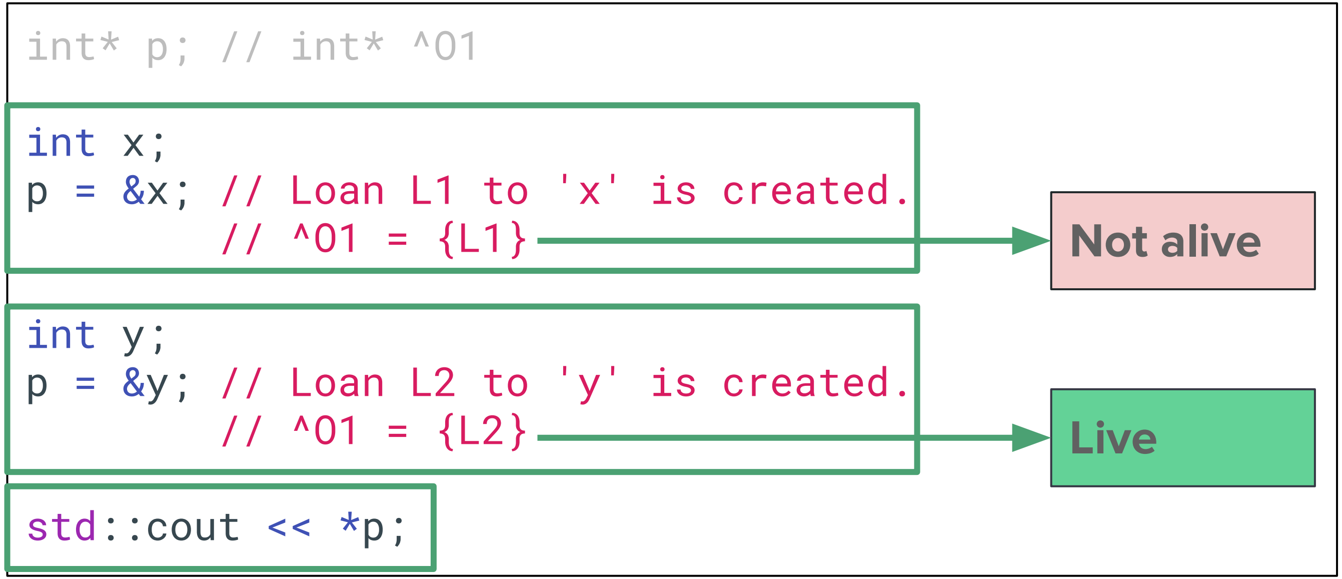
```
int* p;     // int* ^O1
int* q;     // int* ^O2

int x = 42;
p = &x;     // Loan L1 to 'x'
            // O1 = {L1}.

q = p;      // O2 ← O1 = {L1}
```

Live Origins

- “Is this value later used ?”



```
int* p; // int* ^01

int x;
p = &x; // Loan L1 to 'x' is created.
        // ^01 = {L1}

int y;
p = &y; // Loan L2 to 'y' is created.
        // ^01 = {L2}

std::cout << *p;
```

Not alive

Live

The Lifetime Policy

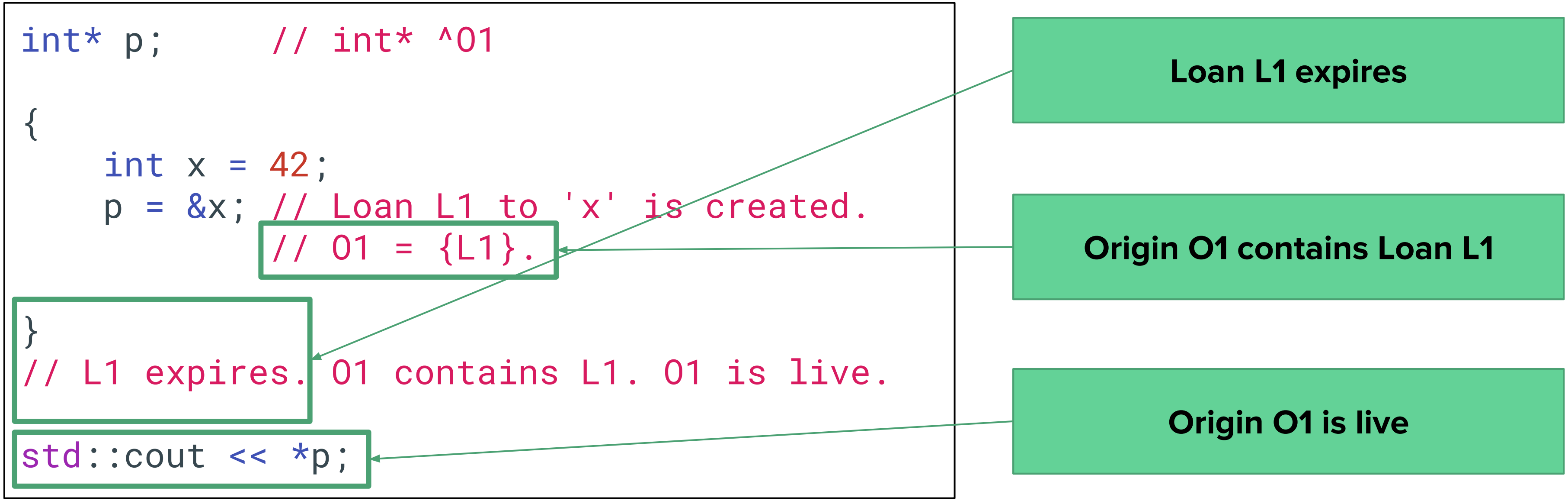
Putting it all together

A lifetime violation is identified at program point **P** if:

- A Loan **L** expires at **P**
- An Origin **O** contains the loan **L** at **P**
- The Origin **O** is live at **P**

“A **live** origin should not contain an **expired** loan”

Lifetime policy: Example



```
int* p;      // int* ^O1

{
    int x = 42;
    p = &x; // Loan L1 to 'x' is created.
            // O1 = {L1}.

}
// L1 expires. O1 contains L1. O1 is live.
std::cout << *p;
```

Loan L1 expires

Origin O1 contains Loan L1

Origin O1 is live

Demo

```
void foo() {
    std::string_view view;
    {
        std::string small = "small scoped string";
        view = small;
        //     ^^^^^ error: object does not live long enough.

    }   // note: destroyed here.
    std::cout << view;
    //           ^^^^ note: later used here.
}
```

3-points diagnostic:

- Borrow site
- Invalidation site
- Use site

Try it out:

- `-Xclang -fexperimental-lifetime-safety -Wexperimental-lifetime-safety`
- https://godbolt.org/z/dEvjP8q86

Dealing with Abstractions

Function calls

If there were no function calls, we would be done here!

What can a function call do ?

Aliasing

Invalidations

Lifetime Contracts

“**Compositional analysis**” instead of inter-procedural analysis.

Extend the language

… with **annotations** and API contracts.

Lifetime Contracts: **Aliasing**

```
std::string_view Identity(const std::string& in) {
    return in;
}
```

```
std::string_view StripSuffix(const std::string& in,
                             const std::string& suffix);
```

Lifetime Contracts: **Aliasing**

```
std::string_view Identity(const std::string& in [[clang::lifetimebound]]) {
    return in;
}
```

```
std::string_view StripSuffix(const std::string& in [[clang::lifetimebound]],
                             const std::string& suffix);
```

Limited solution

- `[[clang::lifetimebound]]` and family...

Lifetime Contracts: **Invalidations**

Invalidations ➔

`push_back()`, `clear()`, `insert()`

```
std::vector<int> v = {1, 2, 3, 4};
auto it = v.find(1);
v.push_back(5); // invalidates 'it'.
```

No solution atm

But can be introduced in the future:

- E.g.
  `[[clang::invalidates(...)]]`
  (or something similar)

Under construction.

Lookout for updates in
2026.

Lifetime Safety

Non goals

What it is not ?

- Rigorous temporal memory safety guarantees for C++
- Borrow checker

Find us more at:

RFC#86291: https://discourse.llvm.org/t/rfc-intra-procedural-lifetime-analysis-in-clang/86291

Biweekly sync

- Lifetime Safety Breakout Group @Wednesdays, 2:30 PM CET
- Added to calendar@llvm.org

Github

- Label: `clang:temporal-safety`
- Umbrella Issue: https://github.com/llvm/llvm-project/issues/152520
- **Project 39**: https://github.com/orgs/llvm/projects/39/

Discord: https://discord.com/channels/636084430946959380/1431071362365128908

Thank you!

Questions ?

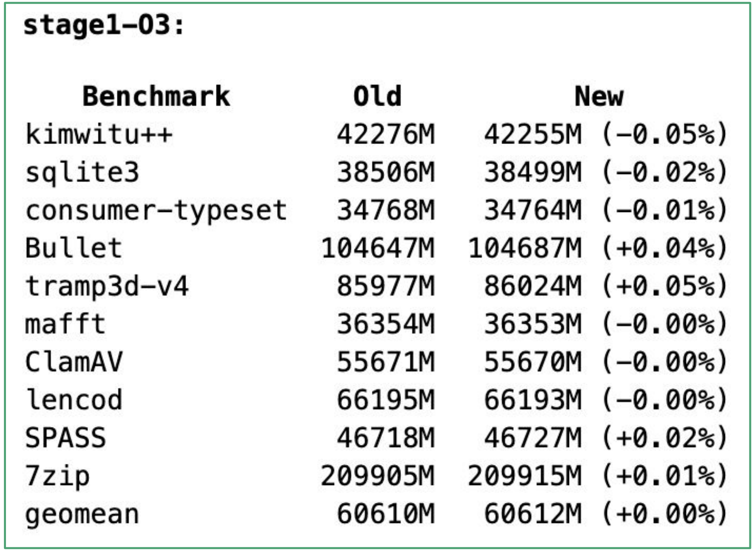
Credits to all the contributors:

- Yitzhak Mandelbaum
- Gábor Horváth
- Haojian Wu
- Kinuko Yasuda
- Dmytro Hrybenko
- Martin Brænne

… and many more!

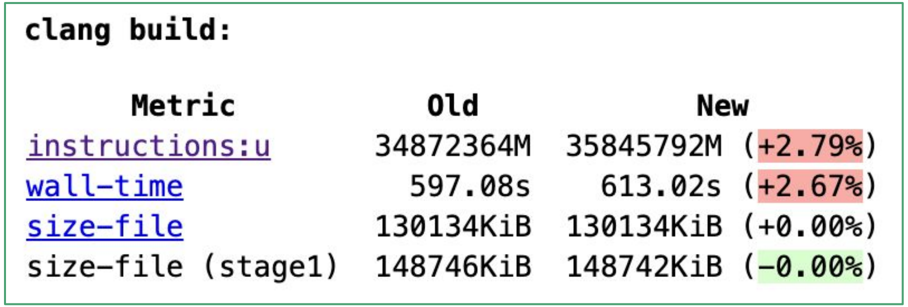
Backup slides

Regressions: Compile-times and Performance



```
stage1-O3:

   Benchmark            Old         New
kimwitu++             42276M    42255M (-0.05%)
sqlite3               38506M    38499M (-0.02%)
consumer-typeset      34768M    34764M (-0.01%)
Bullet               104647M   104687M (+0.04%)
tramp3d-v4            85977M    86024M (+0.05%)
mafft                 36354M    36353M (-0.00%)
ClamAV                55671M    55670M (-0.00%)
lencod                66195M    66193M (-0.00%)
SPASS                 46718M    46727M (+0.02%)
7zip                 209905M   209915M (+0.01%)
geomean               60610M    60612M (+0.00%)
```



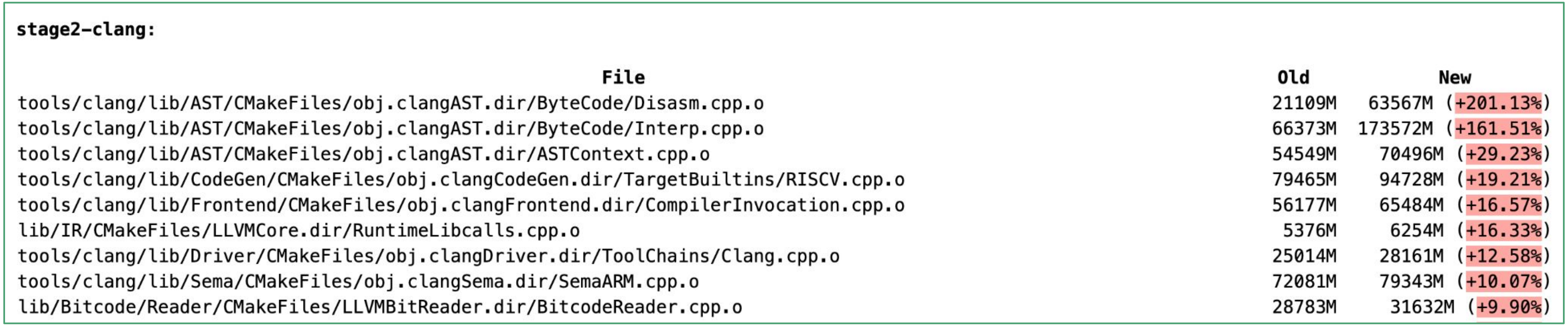
```
clang build:

      Metric                  Old              New
instructions:u           34872364M   35845792M (+2.79%)
wall-time                  597.08s     613.02s (+2.67%)
size-file                130134KiB   130134KiB (+0.00%)
size-file (stage1)       148746KiB   148742KiB (-0.00%)
```

No regressions on plenty of codebases

2-3% regression on LLVM/Clang

Compile-times and Performance



```
stage2-clang:

                                      File                                                Old          New
tools/clang/lib/AST/CMakeFiles/obj.clangAST.dir/ByteCode/Disasm.cpp.o                   21109M    63567M (+201.13%)
tools/clang/lib/AST/CMakeFiles/obj.clangAST.dir/ByteCode/Interp.cpp.o                   66373M   173572M (+161.51%)
tools/clang/lib/AST/CMakeFiles/obj.clangAST.dir/ASTContext.cpp.o                        54549M    70496M (+29.23%)
tools/clang/lib/CodeGen/CMakeFiles/obj.clangCodeGen.dir/TargetBuiltins/RISCV.cpp.o      79465M    94728M (+19.21%)
tools/clang/lib/Frontend/CMakeFiles/obj.clangFrontend.dir/CompilerInvocation.cpp.o      56177M    65484M (+16.57%)
lib/IR/CMakeFiles/LLVMCore.dir/RuntimeLibcalls.cpp.o                                     5376M     6254M (+16.33%)
tools/clang/lib/Driver/CMakeFiles/obj.clangDriver.dir/ToolChains/Clang.cpp.o            25014M    28161M (+12.58%)
tools/clang/lib/Sema/CMakeFiles/obj.clangSema.dir/SemaARM.cpp.o                         72081M    79343M (+10.07%)
lib/Bitcode/Reader/CMakeFiles/LLVMBitReader.dir/BitcodeReader.cpp.o                     28783M     31632M (+9.90%)
```

Worst-case 200% hit to compile-time (~generated code!)

(Possible) Future directions

- Rust-style annotation syntax in Clang
- Annotation suggestion, verification
- Large-scale adoption of Lifetime contract
- Iterator/Pointer Invalidations, e.g. `[[clang::invalidates(...)]]`
- [No]escape analysis
- Summary-based full-program analysis
- Incremental borrow checker rules