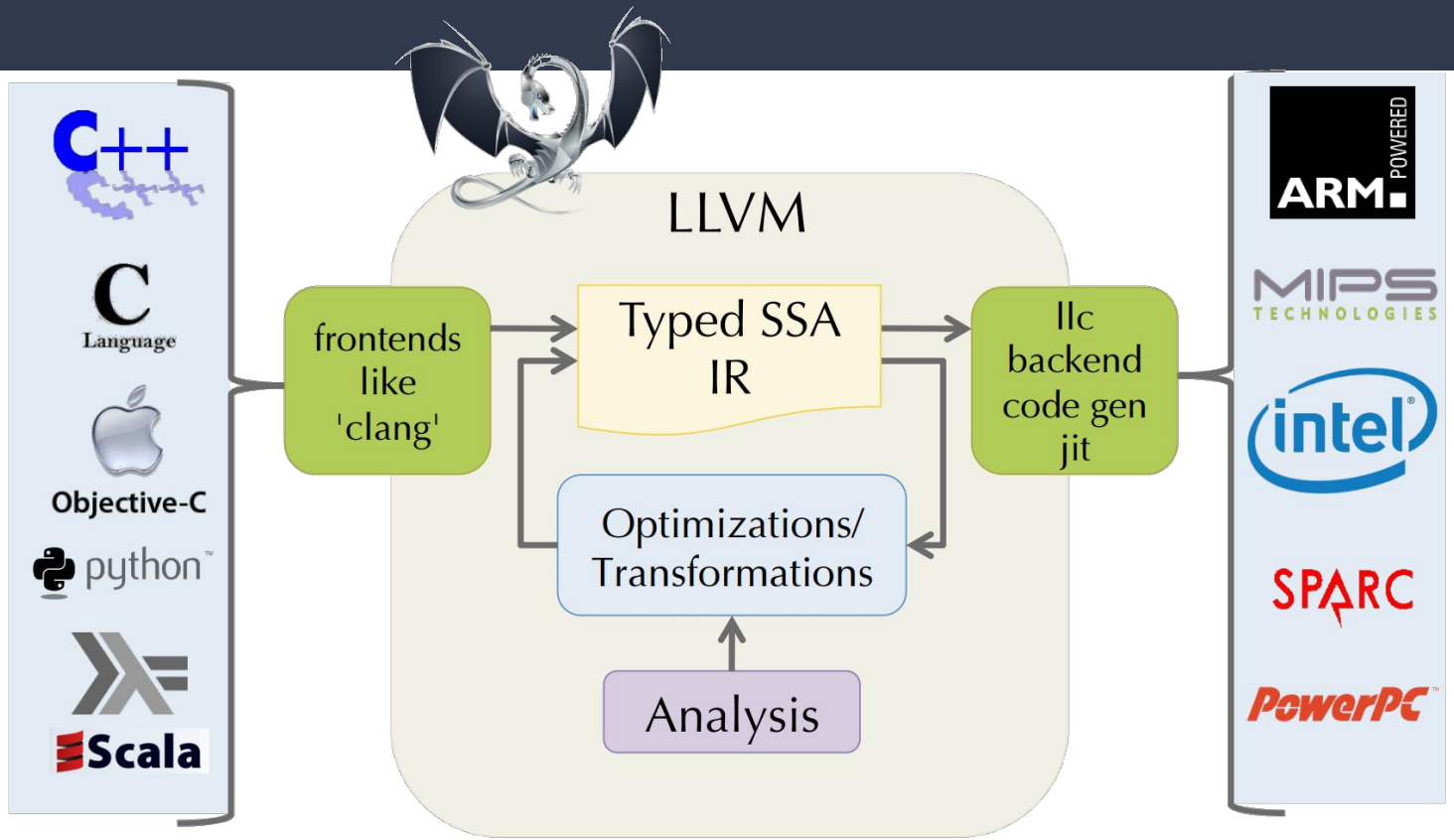
Fuzzing LLVM for Miscompiles

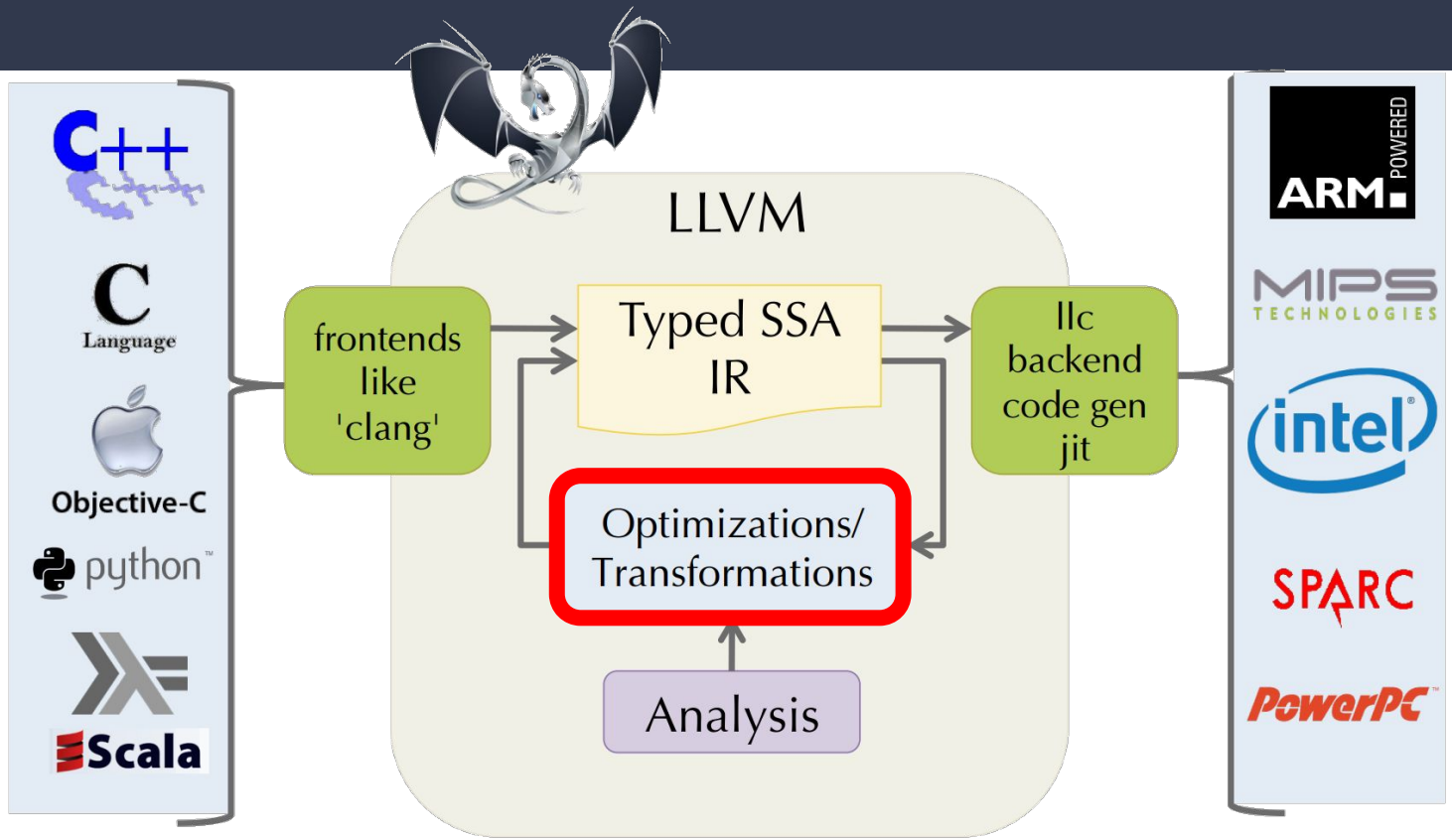
Emmett Neyman (@eneyman)
SWE Intern Project
C++ Dynamic Tools

Slides at go/fuzz-llvm

LLVM in One Slide



LLVM in One Slide



The Problem

When our compiler optimizes code, how do we know that generated machine code or executable is correct?

In short, we are taught to trust that the compiler is correct.

Usually, if there's a bug, it's the programmer's fault, not the compiler's.

But this isn't always the case...

There are two types of compiler bugs...

Compiler Crashes

When you try to compile your code, the compiler crashes and no output is produced.

Fairly common.

Easy to find with fuzzing.

Compiler Miscompiles

When you try to compile your code, incorrect output is generated and no warnings/errors are given.

Not as common and very dangerous. These are the bugs we want to find.

So how do we use fuzzing to find miscompiles?

1. Compile two versions of the code: optimized and not
2. Feed each version the same input and run them
3. Check if the output is the same
4. Repeat

If the output differs, we've found a miscompile!

Fuzzing the LLVM Loop Vectorizer

Why the Loop Vectorizer?

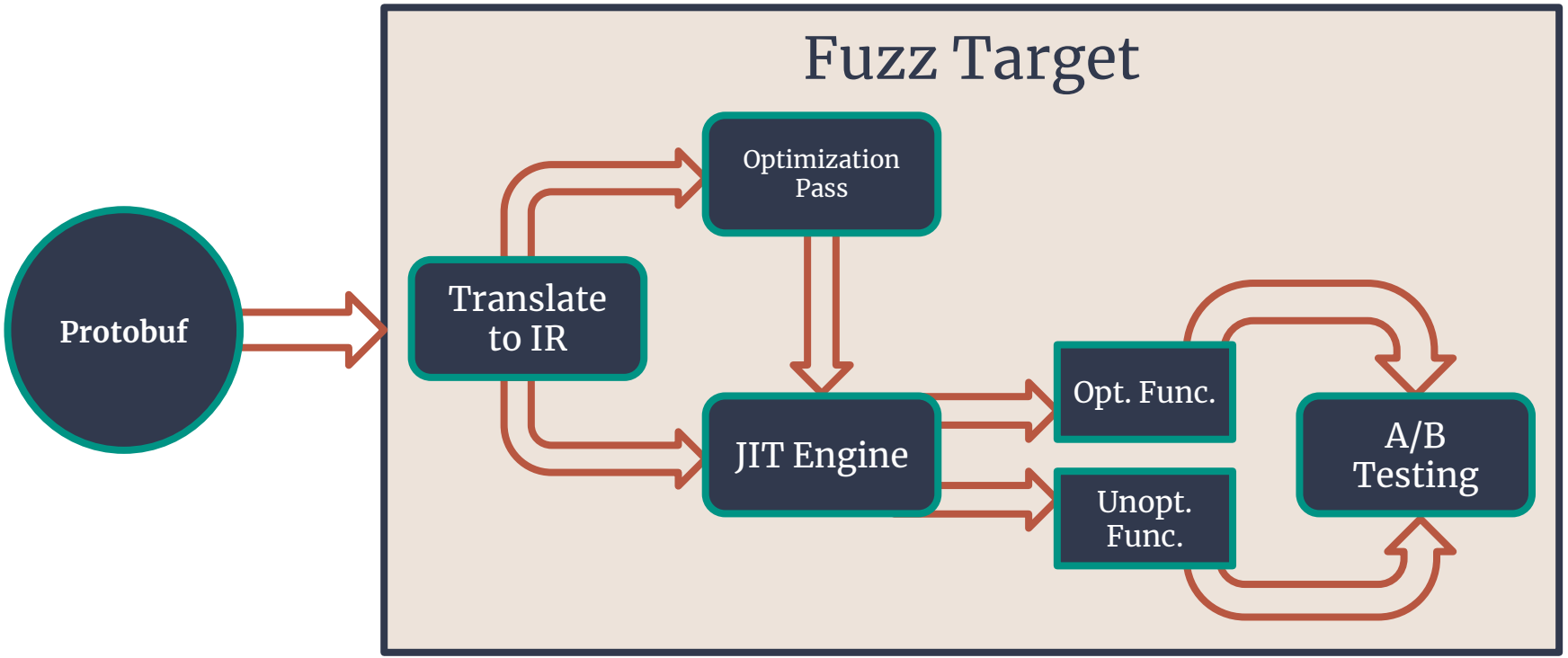
- Easy to trigger; you just need a loop
- Fairly complicated optimization
- Easy to test

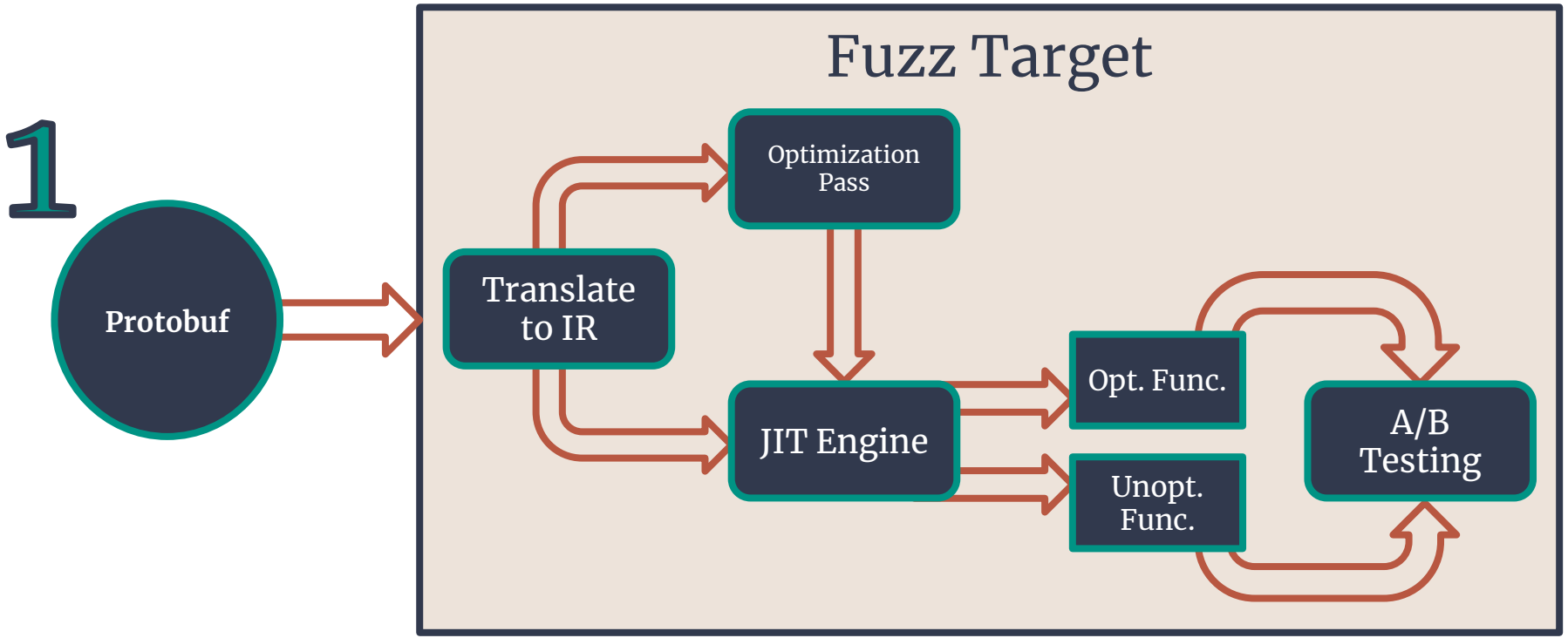
Existing Technology

`protobuf-mutator`
and
`clang-proto-fuzzer`

- Define a subset of C++ as a protobuf
- Bypass lexing/parsing stage
- Only finds compiler crashes, not miscompiles

Overview of the LLVM Proto Fuzzer




1
Protobuf
Fuzz Target
Translate to IR
Optimization Pass
JIT Engine
Opt. Func.
Unopt. Func.
A/B Testing

First Step:
A New Protobuf

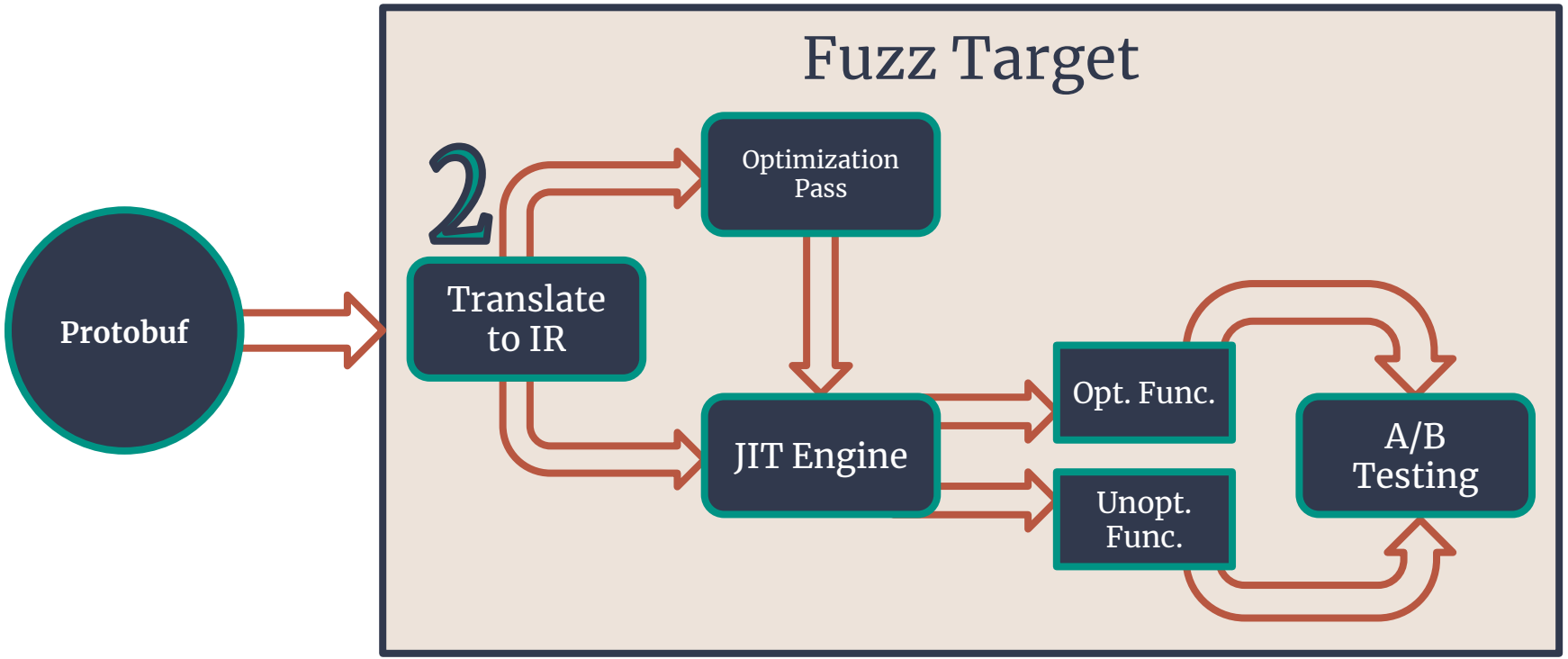
`cxx-loop-proto`

```
message Const {
  required int32 val = 1;
}

message VarRef {
  // Add an enum for each
array in function signature
  enum Arr {
    ARR_A = 0;
    ARR_B = 1;
    ARR_C = 2;
  };
  required Arr arr = 1;
}
```

```
message BinaryOp {
  enum Op {
    PLUS = 0;
    MINUS = 1;
    MUL = 2;
    XOR = 3;
    AND = 4;
    OR = 5;
    EQ = 6;
    NE = 7;
    LE = 8;
    GE = 9;
    LT = 10;
    GT = 11;
  };
  required Op op = 1;
  required Rvalue left = 2;
  required Rvalue right = 3;
}
```

```
message Rvalue {
  oneof rvalue_oneof {
    Const cons = 1;
    BinaryOp binop = 2;
    VarRef varref = 3;
  }
}

message AssnStmt {
  required VarRef varref = 1;
  required Rvalue rvalue = 2;
}

message Statement {
  required AssnStmt assignment = 1;
}

message StatementSeq {
  repeated Statement statements = 1;
}

message LoopFunction {
  required StatementSeq statements = 1;
}
```

- Create a protobuf to represent a subset of C++ (and LLVM IR) that is loop “friendly”
- Want generated code to be more likely to stress the loop vectorizer


Fuzz Target
2
Protobuf
Translate to IR
Optimization Pass
JIT Engine
Opt. Func.
Unopt. Func.
A/B Testing

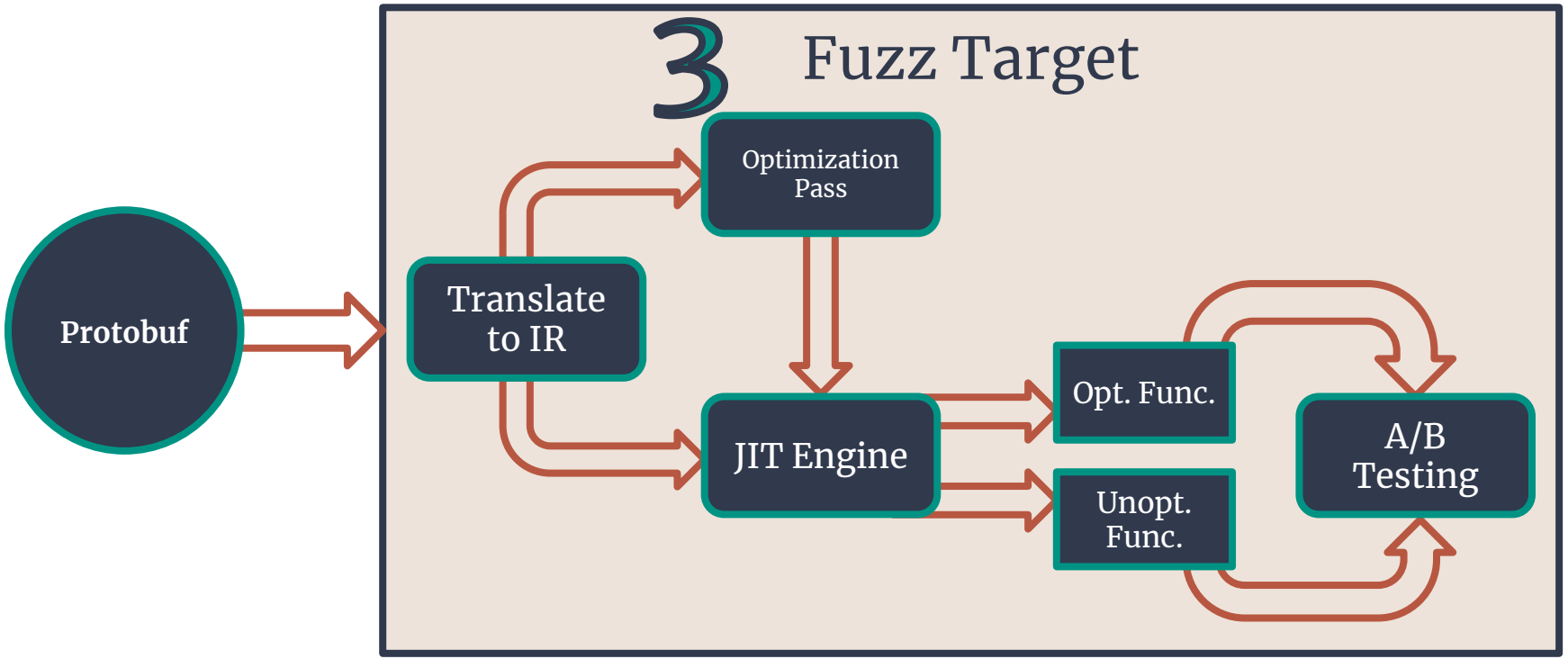
Second Step:
Produce Valid LLVM IR

`loop-proto-to-llvm`

```
target triple = "x86_64-unknown-linux-gnu"
define void @foo(i32* %a, i32* %b, i32* %c, i64 %s) {
%1 = icmp sgt i64 %s, 0
br i1 %1, label %start, label %end
start:
br label %loop
end:
ret void
loop:
%ct   = phi i64 [ %ctnew, %loop ], [ 0, %start ]
%var0 = getelementptr inbounds i32, i32* %b, i64 %ct
store i32 1, i32* %var0
%var1 = getelementptr inbounds i32, i32* %a, i64 %ct
store i32 1, i32* %var1
%var2 = getelementptr inbounds i32, i32* %b, i64 %ct
%var3 = load i32, i32* %var2
%var4 = getelementptr inbounds i32, i32* %b, i64 %ct
store i32 %var3, i32* %var4
%ctnew = add i64 %ct, 1
%j = icmp eq i64 %ctnew, %s
br i1 %j, label %end, label %loop, !llvm.loop !0
}
!0 = distinct !{!0, !1, !2}
!1 = !{!"llvm.loop.vectorize.enable", i1 true}
!2 = !{!"llvm.loop.vectorize.width", i32 64}
```

- Along the same lines of proto-to-cxx, generate valid LLVM IR function from a protobuf

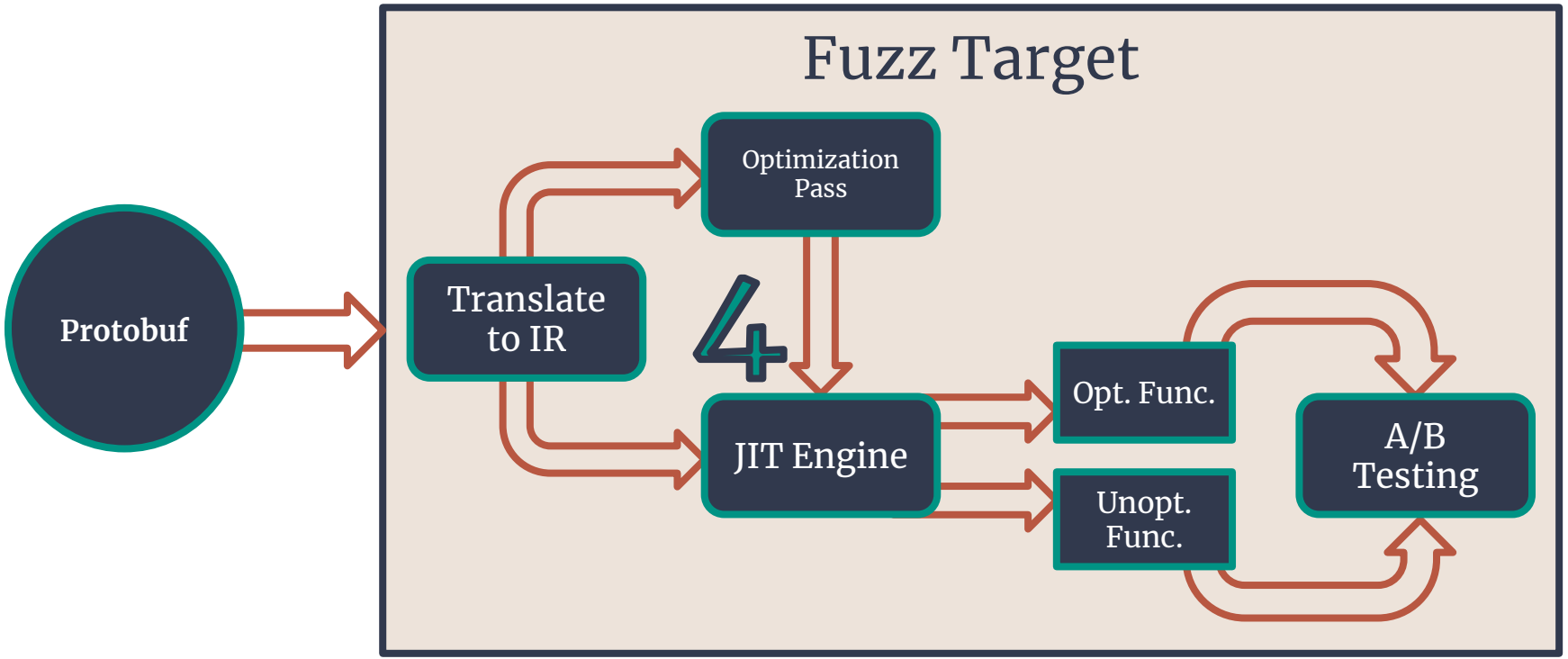
- Convert protobuf to LLVM IR before passing it to fuzz target


3
Fuzz Target
Protobuf
Translate to IR
Optimization Pass
JIT Engine
Opt. Func.
Unopt. Func.
A/B Testing

Third Step:
Run the Optimization Pass

- Before compiling the IR code, run a loop vectorize pass over one of the versions

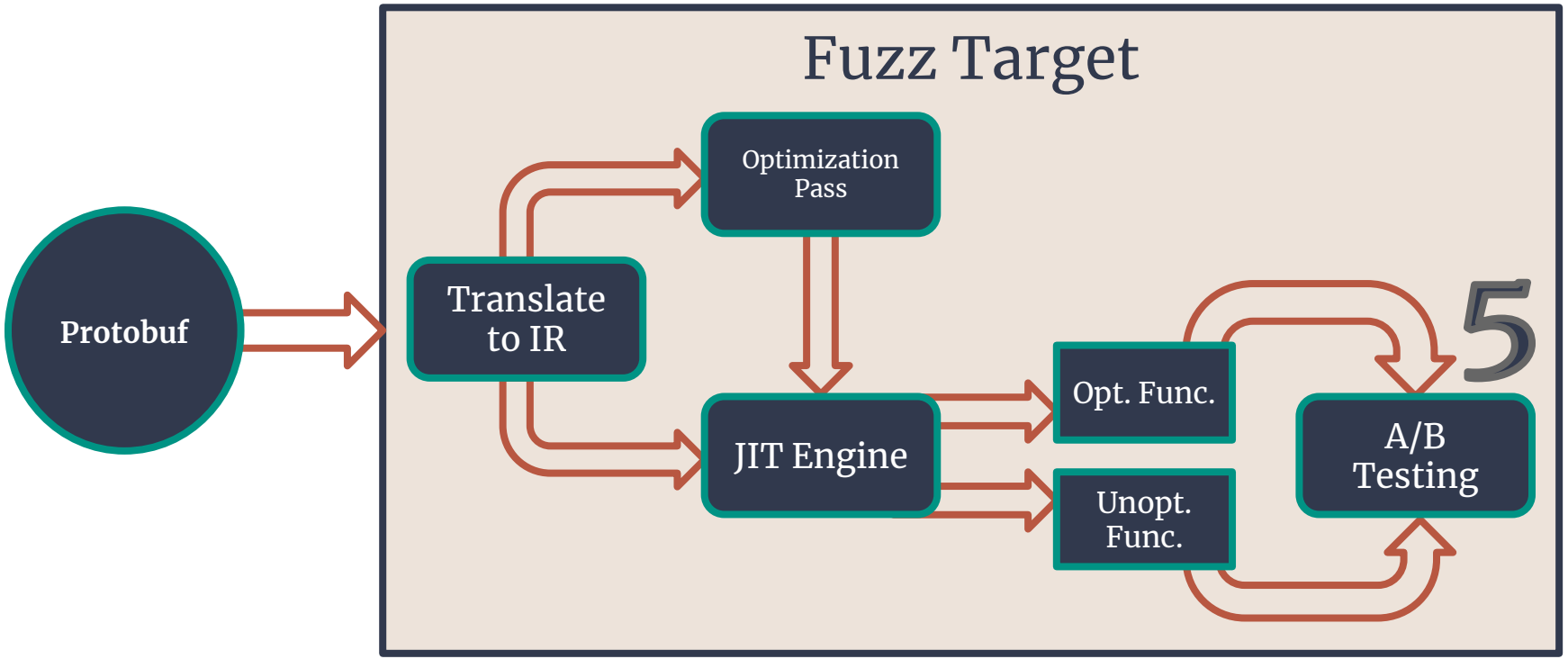
- We have two versions of the IR code: optimized and unoptimized


Fuzz Target
Protobuf
Translate to IR
Optimization Pass
4
JIT Engine
Opt. Func.
Unopt. Func.
A/B Testing

Fourth Step:
Compile the IR Code

- Use LLVM’s JIT engine to compile both versions of the IR

- Obtain a function pointer to each of the compiled functions


Fuzz Target
Protobuf
Translate to IR
Optimization Pass
JIT Engine
Opt. Func.
Unopt. Func.
A/B Testing
5

Fifth Step:
Run the Functions

- Build a suite of inputs that both functions will be executed on

- Run each function on the same input and check that they have the same output

Making Sure it Works:
Introducing a Bug

We want to make sure the fuzzer is achieving good coverage over the loop vectorizer code.

So, we introduce a bug into the vectorizer and use our fuzzer to find it.

What's Next?

- Find some bugs!!!!
- Extend the protobuf
    - More loops
    - Nested loops
    - Conditionals
- Add more optimization passes

Thanks to...

- Kostya
- Matt
- And the rest of the Dynamic Tools team