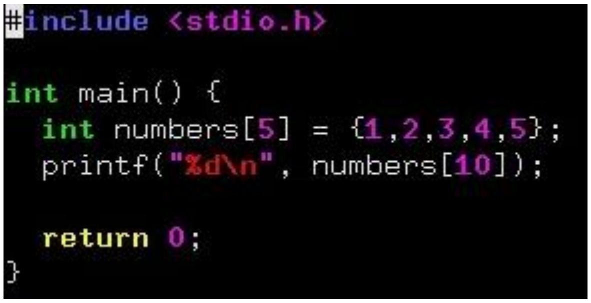
Brandon Montalvo

Lab 3: What does rust actually do

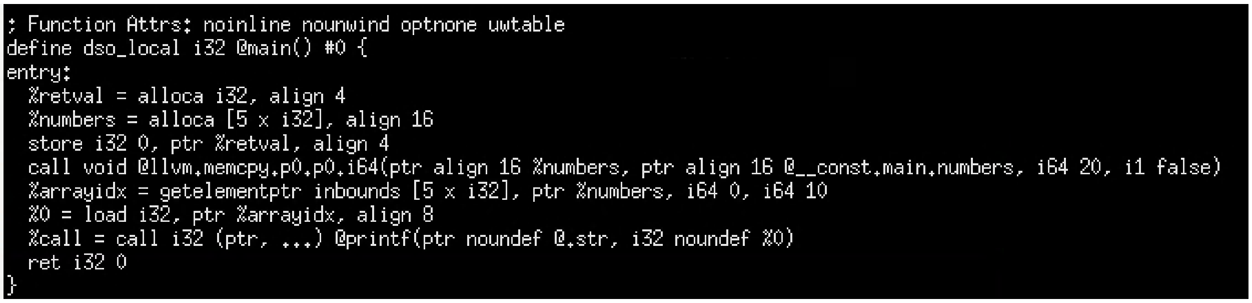
Out of bounds read and write:

hardcoded.c:



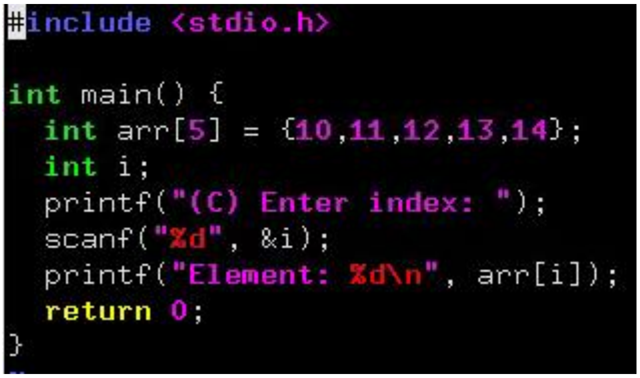
```c
#include <stdio.h>

int main() {
  int numbers[5] = {1,2,3,4,5};
  printf("%d\n", numbers[10]);

  return 0;
}
```

c_hardcoded.ll:



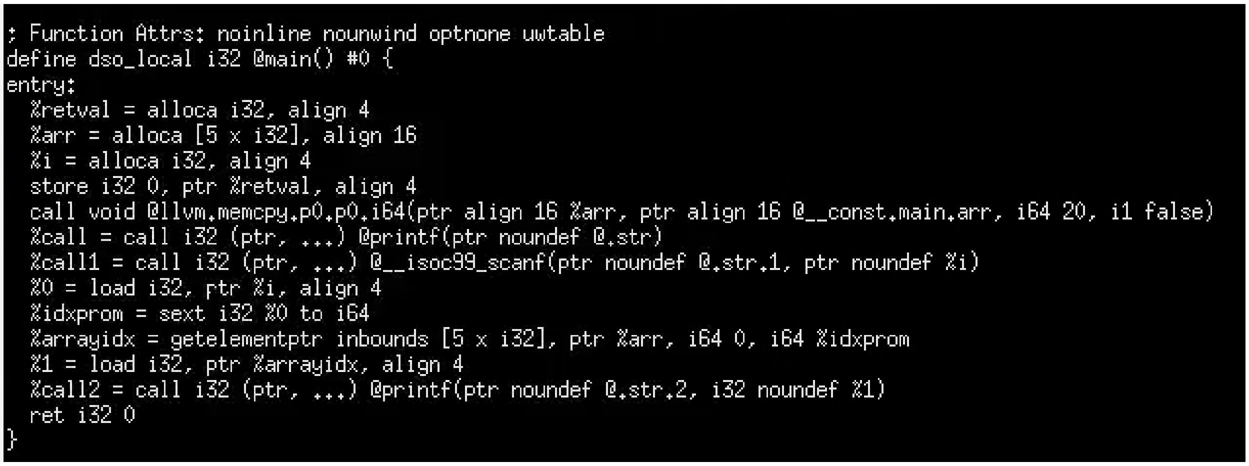
```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %numbers = alloca [5 x i32], align 16
  store i32 0, ptr %retval, align 4
  call void @llvm.memcpy.p0.p0.i64(ptr align 16 %numbers, ptr align 16 @__const.main.numbers, i64 20, i1 false)
  %arrayidx = getelementptr inbounds [5 x i32], ptr %numbers, i64 0, i64 10
  %0 = load i32, ptr %arrayidx, align 8
  %call = call i32 (ptr, ...) @printf(ptr noundef @.str, i32 noundef %0)
  ret i32 0
}
```

dynamic.c:



```c
#include <stdio.h>

int main() {
  int arr[5] = {10,11,12,13,14};
  int i;
  printf("(C) Enter index: ");
  scanf("%d", &i);
  printf("Element: %d\n", arr[i]);
  return 0;
}
```

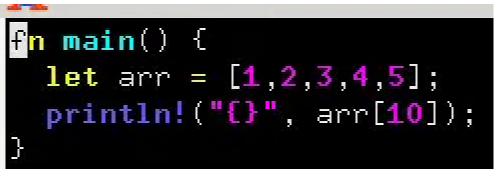
c_dynamic.ll



```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %arr = alloca [5 x i32], align 16
  %i = alloca i32, align 4
  store i32 0, ptr %retval, align 4
  call void @llvm.memcpy.p0.p0.i64(ptr align 16 %arr, ptr align 16 @__const.main.arr, i64 20, i1 false)
  %call = call i32 (ptr, ...) @printf(ptr noundef @.str)
  %call1 = call i32 (ptr, ...) @__isoc99_scanf(ptr noundef @.str.1, ptr noundef %i)
  %0 = load i32, ptr %i, align 4
  %idxprom = sext i32 %0 to i64
  %arrayidx = getelementptr inbounds [5 x i32], ptr %arr, i64 0, i64 %idxprom
  %1 = load i32, ptr %arrayidx, align 4
  %call2 = call i32 (ptr, ...) @printf(ptr noundef @.str.2, i32 noundef %1)
  ret i32 0
}
```

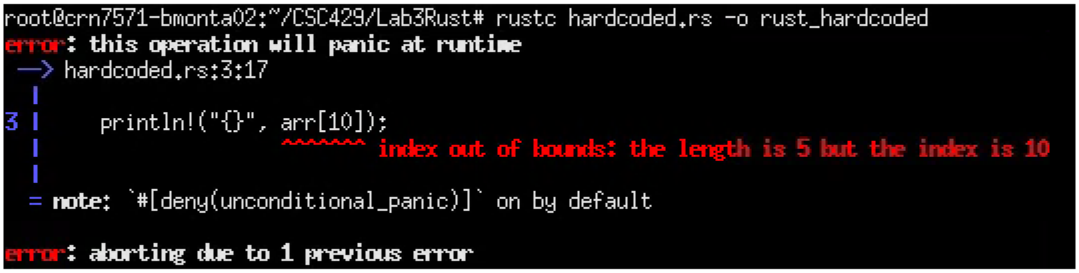
In the llvm for c in both hardcoded and dynamic, there is no bounds check. In the hardcoded c file, it just does a direct getelementptr to index 10. When we compile, there is a warning about the index being past the end of the array. In the dynamic C file, there is no warning during compile time and in the llvm, it loads the user input and again, accesses the array with no bounds check.

hardcoded.rs:

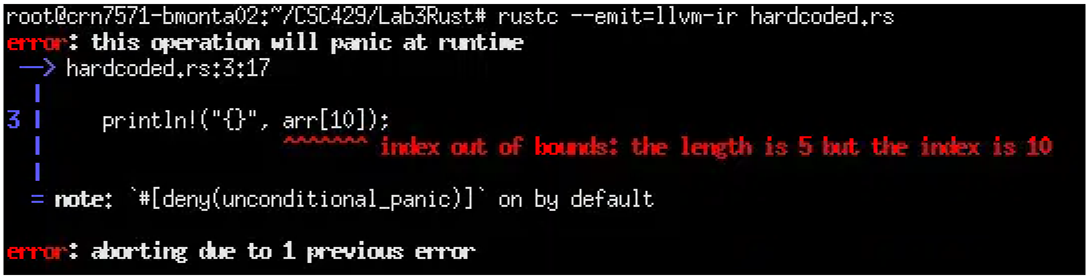


```
fn main() {
   let arr = [1,2,3,4,5];
   println!("{}", arr[10]);
}
```

Trying to compile hardcoded rust:



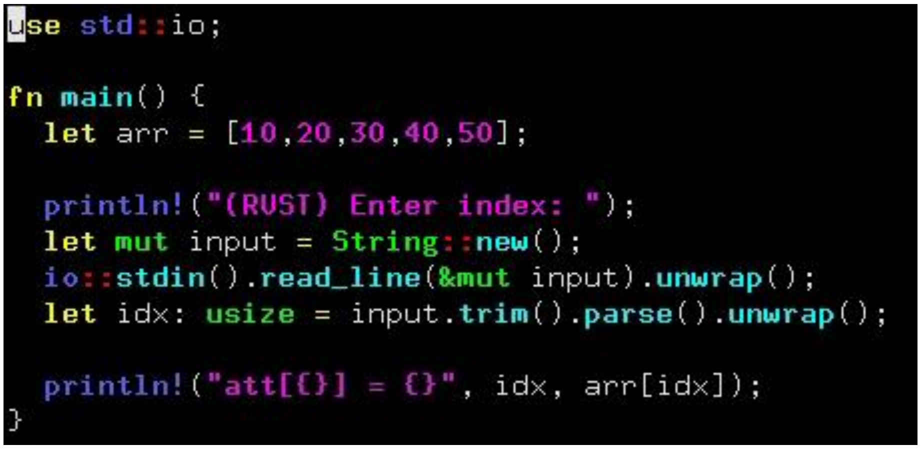
```
root@crn7571-bmonta02:~/CSC429/Lab3Rust# rustc hardcoded.rs -o rust_hardcoded
error: this operation will panic at runtime
 --> hardcoded.rs:3:17
  |
3 |     println!("{}", arr[10]);
  |                    ^^^^^^^ index out of bounds: the length is 5 but the index is 10
  |
  = note: `#[deny(unconditional_panic)]` on by default

error: aborting due to 1 previous error
```



```
root@crn7571-bmonta02:~/CSC429/Lab3Rust# rustc --emit=llvm-ir hardcoded.rs
error: this operation will panic at runtime
 --> hardcoded.rs:3:17
  |
3 |     println!("{}", arr[10]);
  |                    ^^^^^^^ index out of bounds: the length is 5 but the index is 10
  |
  = note: `#[deny(unconditional_panic)]` on by default

error: aborting due to 1 previous error
```

When I try to compile hardcoded rust file, it is an immediate compile error with index out of bounds. No LLVM-IR could be generated either. So, this would lead to zero runtime overhead since it is not allowed to compile in the first place

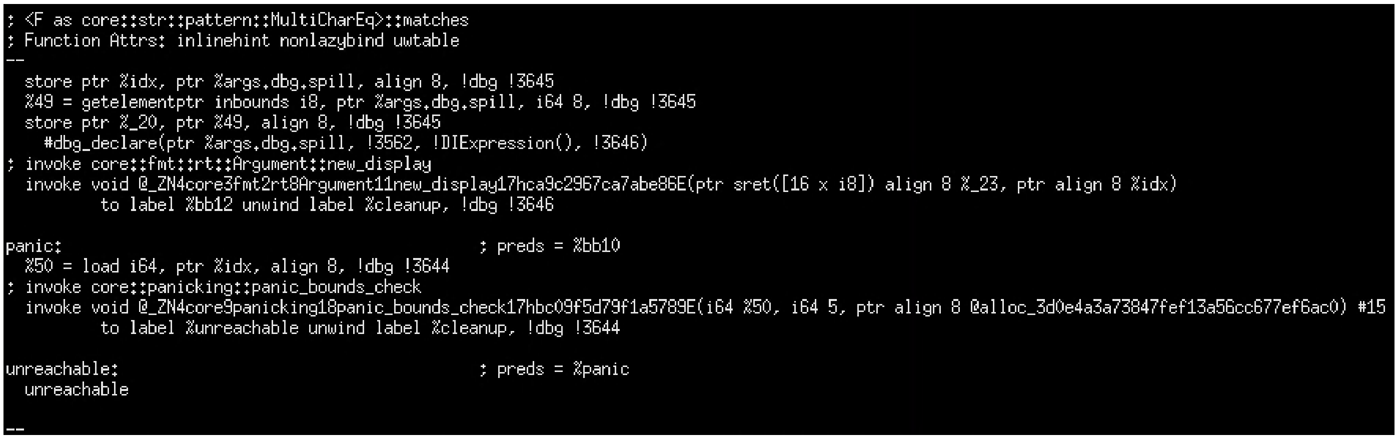
Dynamic.rs:



```rust
use std::io;

fn main() {
  let arr = [10,20,30,40,50];

  println!("(RUST) Enter index: ");
  let mut input = String::new();
  io::stdin().read_line(&mut input).unwrap();
  let idx: usize = input.trim().parse().unwrap();

  println!("att[{}] = {}", idx, arr[idx]);
}
```

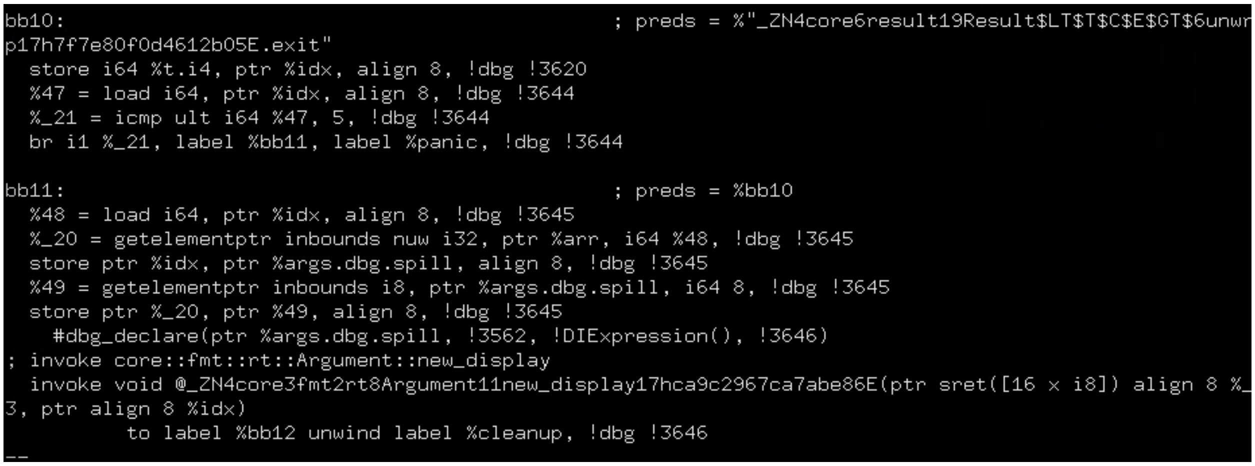
Rust_dynamic.ll:

The file is compiled successfully. In the LLVM-IR, bounds checks are inserted at runtime.



```
; <F as core::str::pattern::MultiCharEq>::matches
; Function Attrs: inlinehint nonlazybind uwtable
--
  store ptr %idx, ptr %args.dbg.spill, align 8, !dbg !3645
  %49 = getelementptr inbounds i8, ptr %args.dbg.spill, i64 8, !dbg !3645
  store ptr %_20, ptr %49, align 8, !dbg !3645
    #dbg_declare(ptr %args.dbg.spill, !3562, !DIExpression(), !3646)
; invoke core::fmt::rt::Argument::new_display
  invoke void @_ZN4core3fmt2rt8Argument11new_display17hca9c2967ca7abe86E(ptr sret([16 x i8]) align 8 %_23, ptr align 8 %idx)
          to label %bb12 unwind label %cleanup, !dbg !3646

panic:                                            ; preds = %bb10
  %50 = load i64, ptr %idx, align 8, !dbg !3644
; invoke core::panicking::panic_bounds_check
  invoke void @_ZN4core9panicking18panic_bounds_check17hbc09f5d79f1a5789E(i64 %50, i64 5, ptr align 8 @alloc_3d0e4a3a73847fef13a56cc677ef6ac0) #15
          to label %unreachable unwind label %cleanup, !dbg !3644

unreachable:                                      ; preds = %panic
  unreachable

--
```

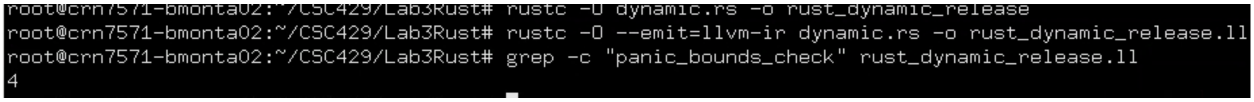
In the screenshot above, it shows "; preds = %bb10" which means that the panic label comes from bb10



```
bb10:                                             ; preds = %"_ZN4core6result19Result$LT$T$C$E$GT$6unwr
p17h7f7e80f0d4612b05E.exit"
  store i64 %t.i4, ptr %idx, align 8, !dbg !3620
  %47 = load i64, ptr %idx, align 8, !dbg !3644
  %_21 = icmp ult i64 %47, 5, !dbg !3644
  br i1 %_21, label %bb11, label %panic, !dbg !3644

bb11:                                             ; preds = %bb10
  %48 = load i64, ptr %idx, align 8, !dbg !3645
  %_20 = getelementptr inbounds nuw i32, ptr %arr, i64 %48, !dbg !3645
  store ptr %idx, ptr %args.dbg.spill, align 8, !dbg !3645
  %49 = getelementptr inbounds i8, ptr %args.dbg.spill, i64 8, !dbg !3645
  store ptr %_20, ptr %49, align 8, !dbg !3645
    #dbg_declare(ptr %args.dbg.spill, !3562, !DIExpression(), !3646)
; invoke core::fmt::rt::Argument::new_display
  invoke void @_ZN4core3fmt2rt8Argument11new_display17hca9c2967ca7abe86E(ptr sret([16 x i8]) align 8 %_
3, ptr align 8 %idx)
          to label %bb12 unwind label %cleanup, !dbg !3646
--
```

Looking at bb10, there is a comparison where it checks if idx < 5 then there is a branch to either bb11 if true or else go to panic. This is the runtime overhead that rust adds for safety.

If we look at the release mode:

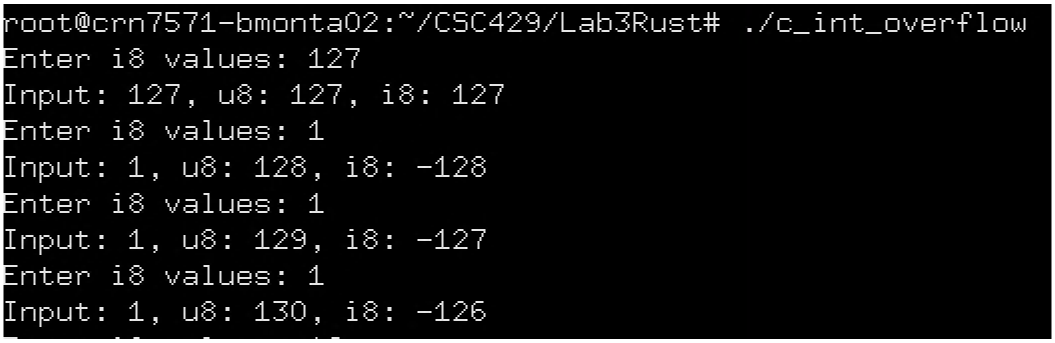


```
root@crn7571-bmonta02:~/CSC429/Lab3Rust# rustc -O dynamic.rs -o rust_dynamic_release
root@crn7571-bmonta02:~/CSC429/Lab3Rust# rustc -O --emit=llvm-ir dynamic.rs -o rust_dynamic_release.ll
root@crn7571-bmonta02:~/CSC429/Lab3Rust# grep -c "panic_bounds_check" rust_dynamic_release.ll
4
```

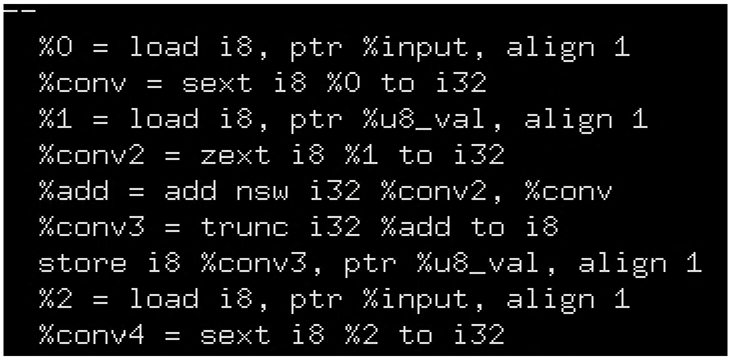
This screenshot shows that the release mode still has bound checks. If we count how many times “panic_bounds_check” appears, it is 4 but in the debug mode, the count is 8 so the release may optimize some things differently

Integer Overflows:

C int overflow:



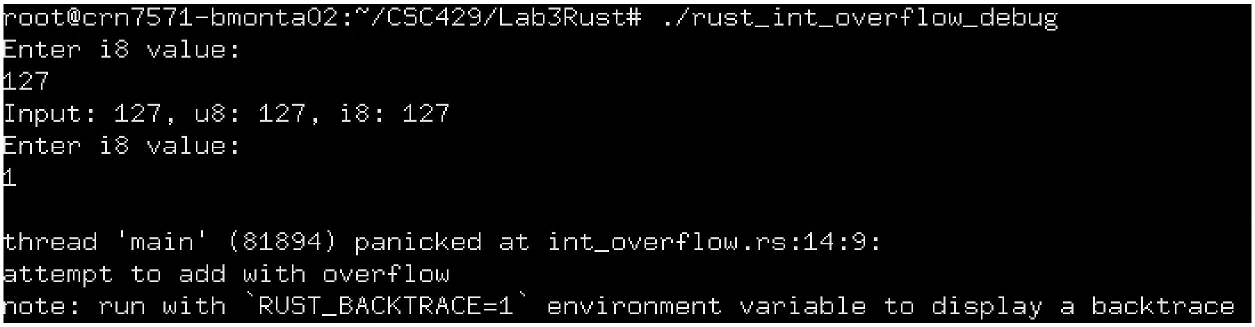
```
root@crn7571-bmonta02:~/CSC429/Lab3Rust# ./c_int_overflow
Enter i8 values: 127
Input: 127, u8: 127, i8: 127
Enter i8 values: 1
Input: 1, u8: 128, i8: -128
Enter i8 values: 1
Input: 1, u8: 129, i8: -127
Enter i8 values: 1
Input: 1, u8: 130, i8: -126
```



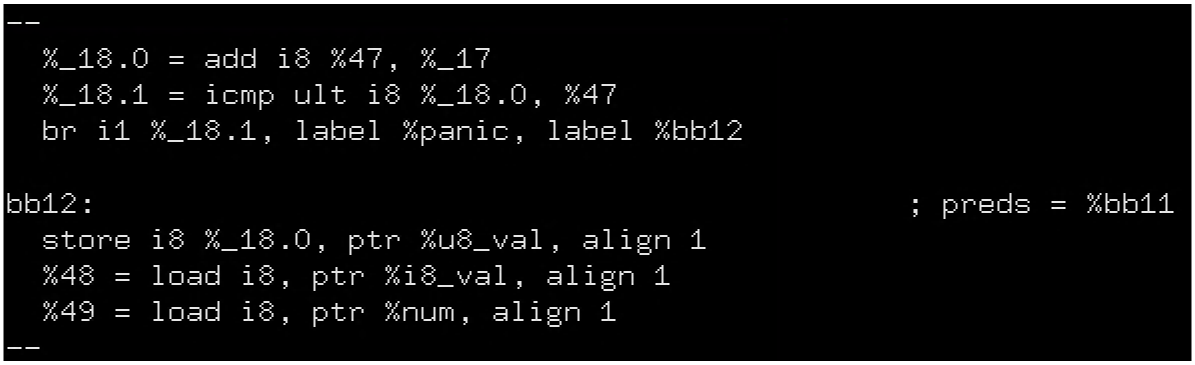
```
--
  %0 = load i8, ptr %input, align 1
  %conv = sext i8 %0 to i32
  %1 = load i8, ptr %u8_val, align 1
  %conv2 = zext i8 %1 to i32
  %add = add nsw i32 %conv2, %conv
  %conv3 = trunc i32 %add to i8
  store i8 %conv3, ptr %u8_val, align 1
  %2 = load i8, ptr %input, align 1
  %conv4 = sext i8 %2 to i32
```

C just has an add instruction with no overflow detection. The result is truncated and stored directly, so it has silent wrapping on overflow.

Rust int overflow (debug):



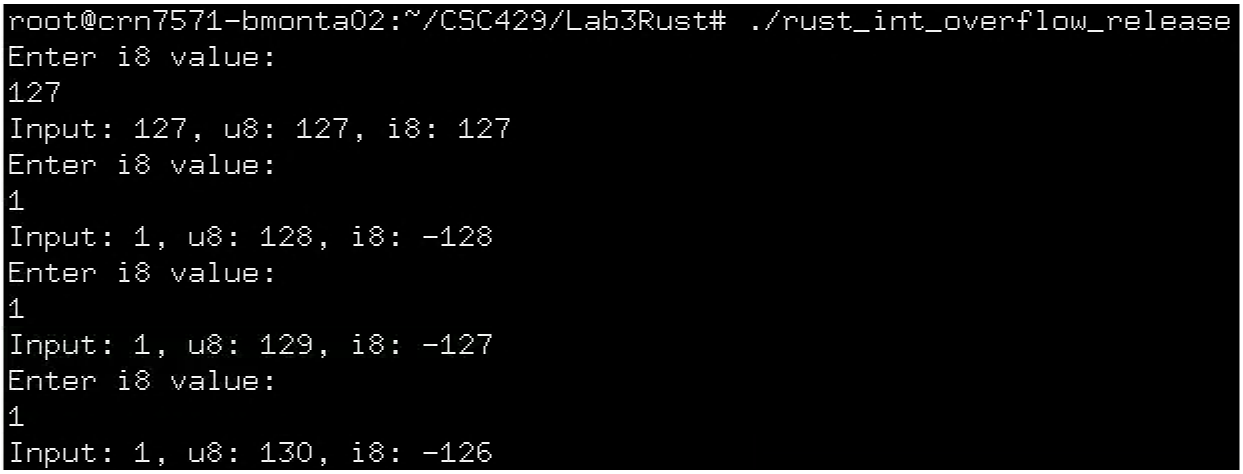
```
root@crn7571-bmonta02:~/CSC429/Lab3Rust# ./rust_int_overflow_debug
Enter i8 value:
127
Input: 127, u8: 127, i8: 127
Enter i8 value:
1

thread 'main' (81894) panicked at int_overflow.rs:14:9:
attempt to add with overflow
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```



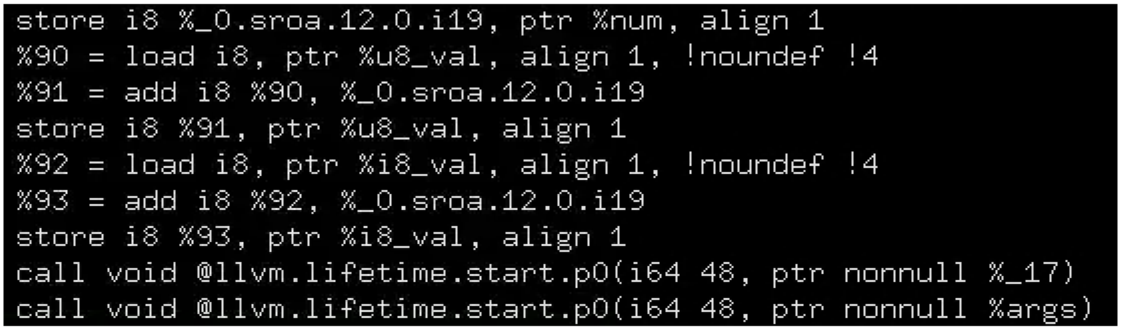
```
--
  %_18.0 = add i8 %47, %_17
  %_18.1 = icmp ult i8 %_18.0, %47
  br i1 %_18.1, label %panic, label %bb12

bb12:                                             ; preds = %bb11
  store i8 %_18.0, ptr %u8_val, align 1
  %48 = load i8, ptr %i8_val, align 1
  %49 = load i8, ptr %num, align 1
--
```

In Rust debug, it inserts an overflow check using icmp and a conditional branch to panic if overflow is detected. So, for the overhead there is an extra check and a conditional branch per addition.

Rust int overflow (release):



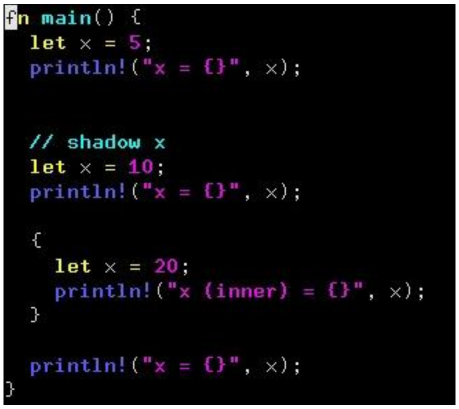
```
root@crn7571-bmonta02:~/CSC429/Lab3Rust# ./rust_int_overflow_release
Enter i8 value:
127
Input: 127, u8: 127, i8: 127
Enter i8 value:
1
Input: 1, u8: 128, i8: -128
Enter i8 value:
1
Input: 1, u8: 129, i8: -127
Enter i8 value:
1
Input: 1, u8: 130, i8: -126
```



```
store i8 %_0.sroa.12.0.i19, ptr %num, align 1
%90 = load i8, ptr %u8_val, align 1, !noundef !4
%91 = add i8 %90, %_0.sroa.12.0.i19
store i8 %91, ptr %u8_val, align 1
%92 = load i8, ptr %i8_val, align 1, !noundef !4
%93 = add i8 %92, %_0.sroa.12.0.i19
store i8 %93, ptr %i8_val, align 1
call void @llvm.lifetime.start.p0(i64 48, ptr nonnull %_17)
call void @llvm.lifetime.start.p0(i64 48, ptr nonnull %args)
```

In the llvm, there is no zero overflow checking it just has an add instruction. This means that it is faster but unsafe. It allows wrapping behavior without panicking.

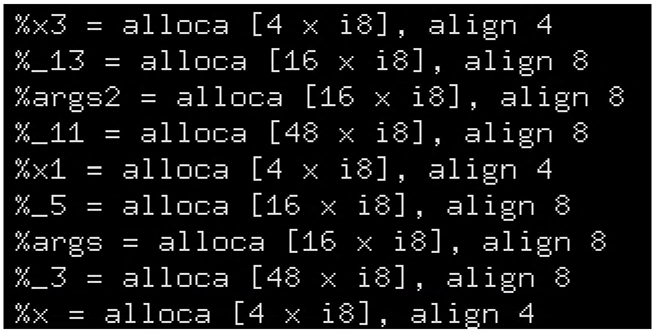
Shadowing:



```
fn main() {
  let x = 5;
  println!("x = {}", x);


  // shadow x
  let x = 10;
  println!("x = {}", x);

  {
    let x = 20;
    println!("x (inner) = {}", x);
  }

  println!("x = {}", x);
}
```

In my code, I define x, then shadow it, and then shadow it again within a scope while printing at each step.

Using grep to find all the alloca's in the LLVM file, I found this part:



```
%x3 = alloca [4 x i8], align 4
%_13 = alloca [16 x i8], align 8
%args2 = alloca [16 x i8], align 8
%_11 = alloca [48 x i8], align 8
%x1 = alloca [4 x i8], align 4
%_5 = alloca [16 x i8], align 8
%args = alloca [16 x i8], align 8
%_3 = alloca [48 x i8], align 8
%x = alloca [4 x i8], align 4
```

Rust is creating three separate stack variables (%x, %x1, and %x3) even though I used the same name "x" in my code. So, when I exit the inner scope, %x3 goes out of scope and %x1 is still there with value 10. I think what is happening is that shadowing creates distinct stack allocations for each shadowed variable. So, each time we use 'let x', it creates a unique LLVM variable name. When the scope ends, the shadowed variables are dropped, but the outer shadowed variables are still there.

Summary:

Which programs compiled?

- All C programs compiled (hardcoded and dynamic) successfully.
- Rust hardcoded failed to compile with a bounds check error.
- Rust dynamic compiled successfully as well.

What errors did the rust compiler produce when it didn't build?

- Rust error said that the operation will panic at runtime because of the index out of bounds.

Which programs allowed hardcoded OOB read/write?

- Only C allowed hardcoded out-of-bounds access.

Which programs allowed user-inputted OOB read/write?

- C allowed it with no runtime check while Rust debug and release panicked on out-of-bounds input.

Runtime Behavior in Integer Overflows:

- C: 127 + 1 = -128

- Rust Debug: Panics
- Rust Release: 127 + 1 = -128