RustMC: Automated Verification of Real-World Concurrent Rust

Oliver Pearce, Julien Lange, Dan O'Keeffe

Rust's safety rules

In safe Rust a value can have **either**:

- One mutable reference
- Any number of immutable references

- Data race
  - Multiple threads concurrently access the same memory location, where at least one access is mutable.

Motivation

The `unsafe` keyword allows developers to opt out of Rust's compiler-enforced safety guarantees.

Unsafe usage

- The most common purpose (42%-45%) for using `unsafe` was reusing existing code. [Qin et al. 2020, Evans et al. 2020]
- 44.6% of `unsafe` function definitions are static bindings to foreign items. [Astrauskas et al. 2020]

Cross language data races

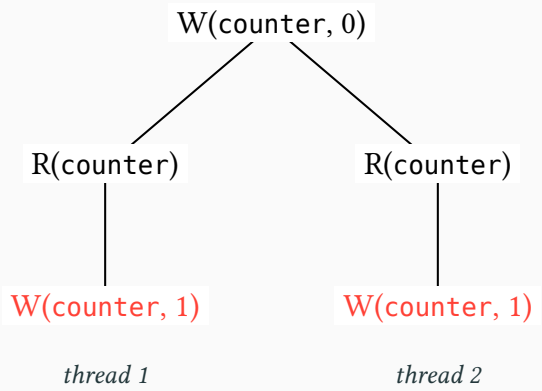
Rust Code

```
extern "C" {fn inc_C_counter();}

#[test]
fn test_C_counter() {
    thread::spawn(|| {
        unsafe {inc_C_counter();}
    });
    thread::spawn(|| {
        unsafe {inc_C_counter();}
    });
}
```

C Dependency

```
int counter = 0;

void inc_C_counter() {
    counter++;
}
```

- Two threads make unsynchronised mutable accesses to `counter` via FFI calls to `inc_C_counter()`.



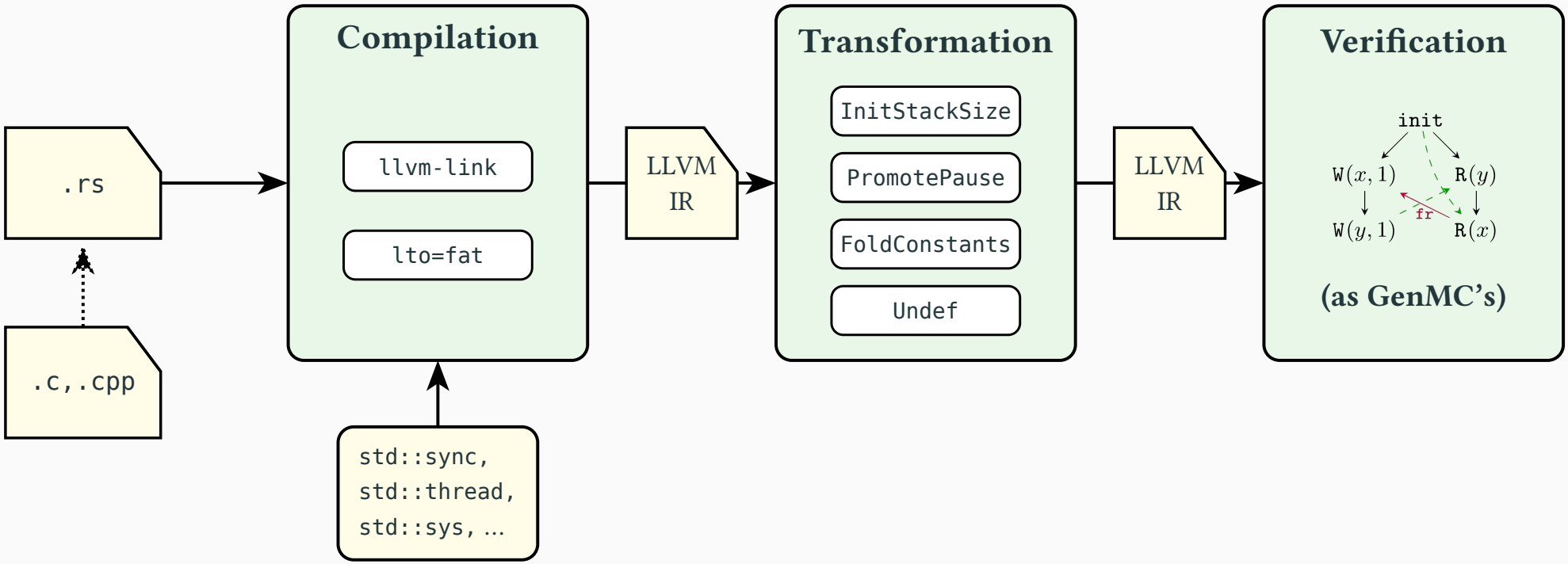
Concurrency bug detection for Rust

- Shuttle
  - Randomised scheduler, focuses on scalability over soundness.
  - Explores interleavings up to a given depth.
- Miri
  - Sanitiser based.
  - Dynamic, non-exhaustive data race detection.
- Loom
  - Exhaustively explores all interleavings
  - Requires manual porting of tests to the Loom API.
  - Does not support mixed Rust/C code.

RustMC

- Stateless model checker for concurrent Rust code
- Push-button verification of unmodified real-world concurrent Rust
- Allows for mixed Rust/C code
- Extends state-of-the-art stateless model checker for C code (GenMC)

RustMC architecture



Key challenges

- **Uninitialised memory**
  - Rust code can emit reads from uninitialised memory.
  - Verifier's model of memory accesses requires each read to have a preceding write to take its value from.
- **Memory intrinsics**
  - Memory intrinsics perform untyped accesses on blocks of memory.
  - Must be promoted to a series of individual typed accesses.

Uninitialised memory

LLVM `undef`

- Used to model uninitialised memory.
- Indicates to the compiler that a program is well defined no matter what value is used, providing freedom to optimise.
- Each use of an instruction that depends on `undef` can observe a different value.

Uninitialised memory

LLVM `undef`

- Used to model uninitialised memory.
- Indicates to the compiler that a program is well defined no matter what value is used, providing freedom to optimise.
- Each use of an instruction that depends on `undef` can observe a different value.

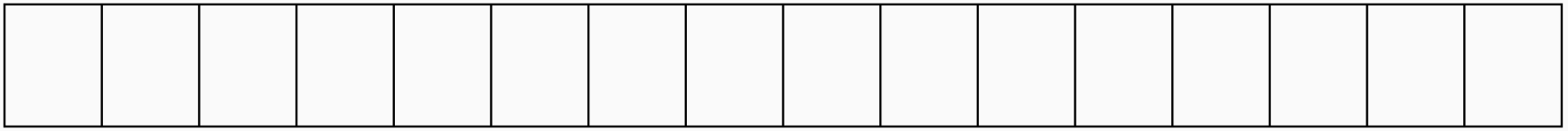
- Providing `undef` as an operand to an instruction that triggers UB for some values of that operand triggers immediate UB.
  - e.g. `udiv %x, undef`

Uninitialised memory

```
pub fn foo() -> Result<(), i64> { Ok(()) }
```

```
define { i64, i64 } @foo() {
```

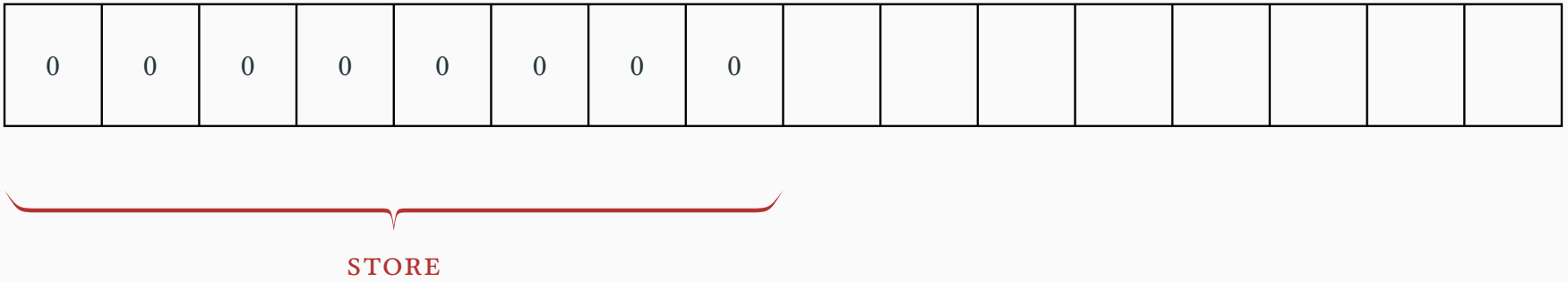
Uninitialised memory



```
pub fn foo() -> Result<(), i64> { Ok(()) }
```

```
define { i64, i64 } @foo() {
%_0 = alloca [16 x i8]
```

Uninitialised memory

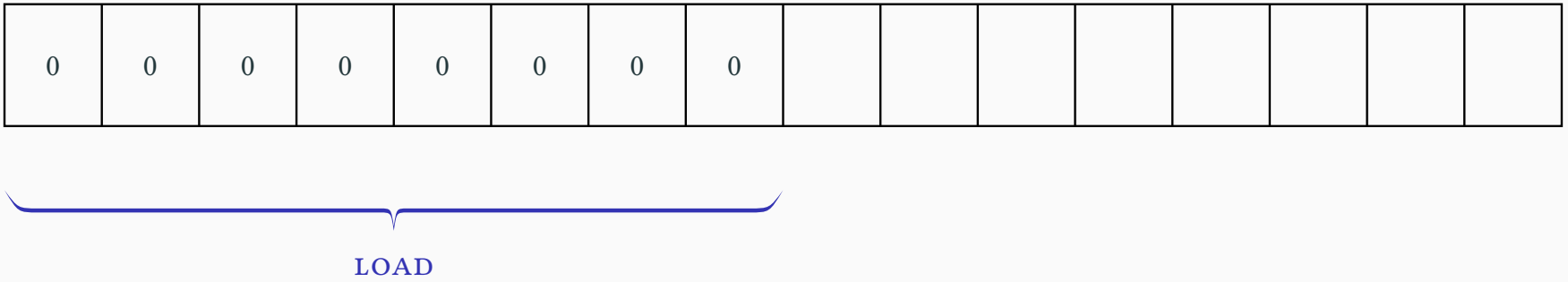



```
pub fn foo() -> Result<(), i64> { Ok(()) }
```

```
define { i64, i64 } @foo() {
%_0 = alloca [16 x i8]
store i64 0, ptr %_0
```

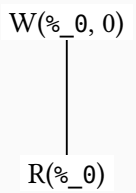
W(%_0, 0)

Uninitialised memory

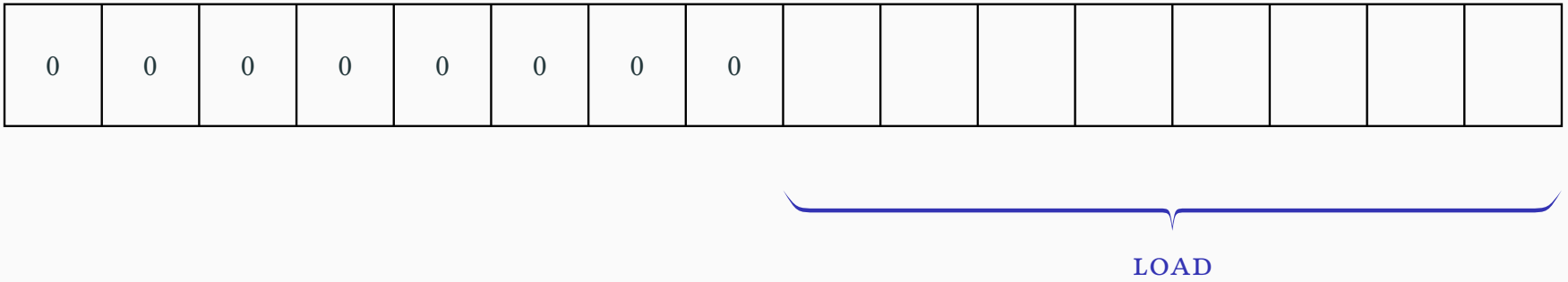



```rust
pub fn foo() -> Result<(), i64> { Ok(()) }
```

```
define { i64, i64 } @foo() {
%_0 = alloca [16 x i8]
store i64 0, ptr %_0
%0 = load i64, ptr %_0
```

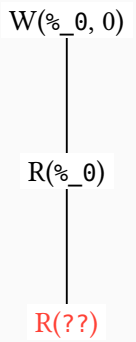


Uninitialised memory

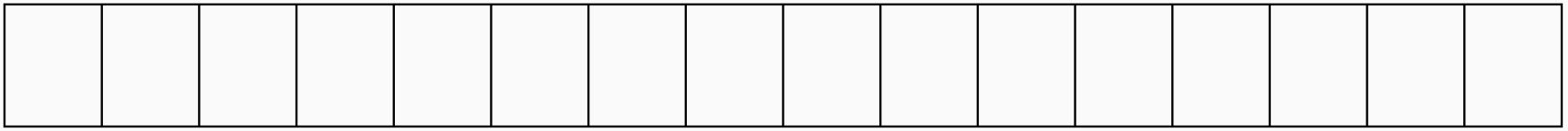



```
pub fn foo() -> Result<(), i64> { Ok(()) }
```

```
define { i64, i64 } @foo() {
%_0 = alloca [16 x i8]
store i64 0, ptr %_0
%0 = load i64, ptr %_0
%1 = getelementptr inbounds i8, ptr %_0, i64 8
%2 = load i64, ptr %1
```



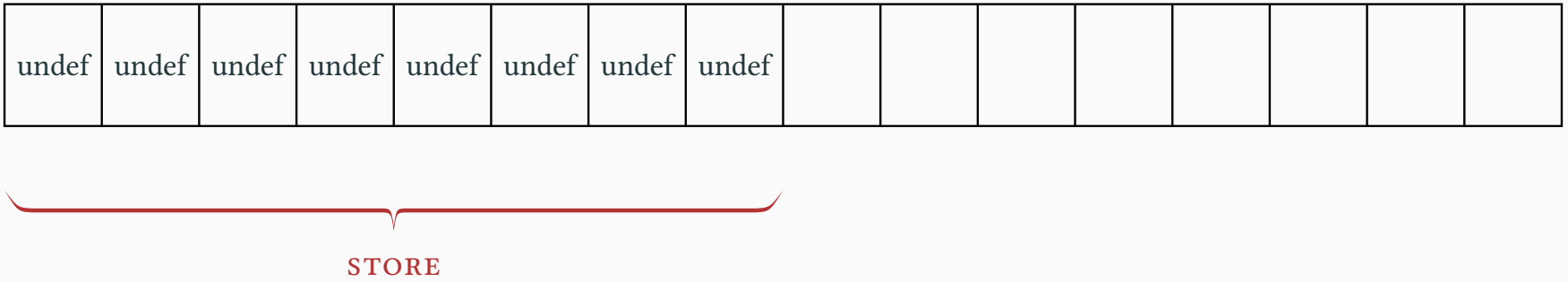
Uninitialised memory



```rust
pub fn foo() -> Result<(), i64> { Ok(()) }
```

```
define { i64, i64 } @foo() {
%_0 = alloca [16 x i8]
```

Uninitialised memory

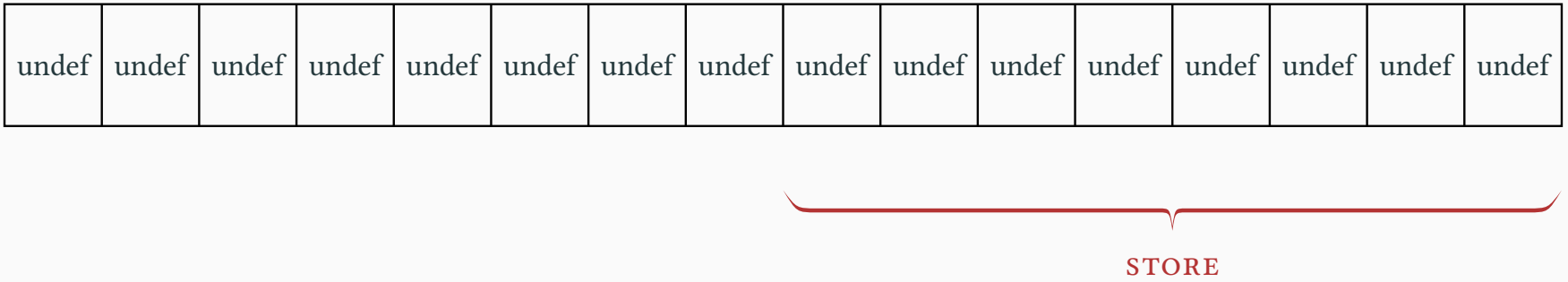



```
pub fn foo() -> Result<(), i64> { Ok(()) }
```

```
define { i64, i64 } @foo() {
%_0 = alloca [16 x i8]
store i64 undef, ptr %_0
```

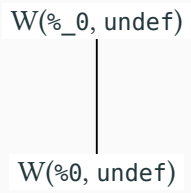
W(%_0, undef)

Uninitialised memory

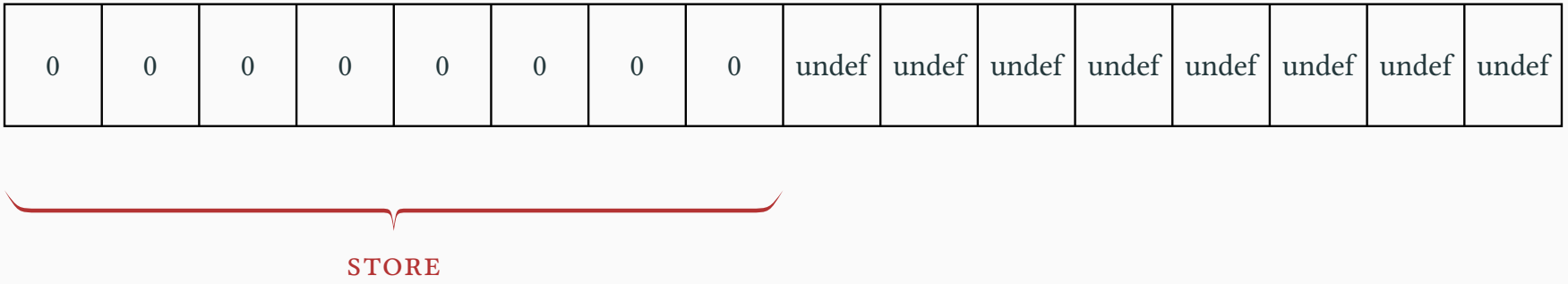



```
pub fn foo() -> Result<(), i64> { Ok(()) }
```

```
define { i64, i64 } @foo() {
%_0 = alloca [16 x i8]
store i64 undef, ptr %_0
%0 = getelementptr inbounds i8, ptr %_0, i64 8
store i64 undef, ptr %0
```

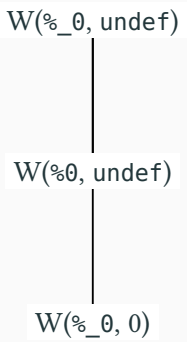


Uninitialised memory




```rust
pub fn foo() -> Result<(), i64> { Ok(()) }
```

```
define { i64, i64 } @foo() {
%_0 = alloca [16 x i8]
store i64 undef, ptr %_0
%0 = getelementptr inbounds i8, ptr %_0, i64 8
store i64 undef, ptr %0
store i64 0, ptr %_0
```



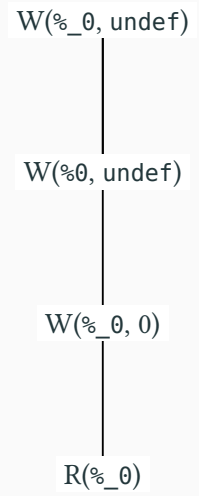
Uninitialised memory

0	0	0	0	0	0	0	0	undef	undef	undef	undef	undef	undef	undef	undef

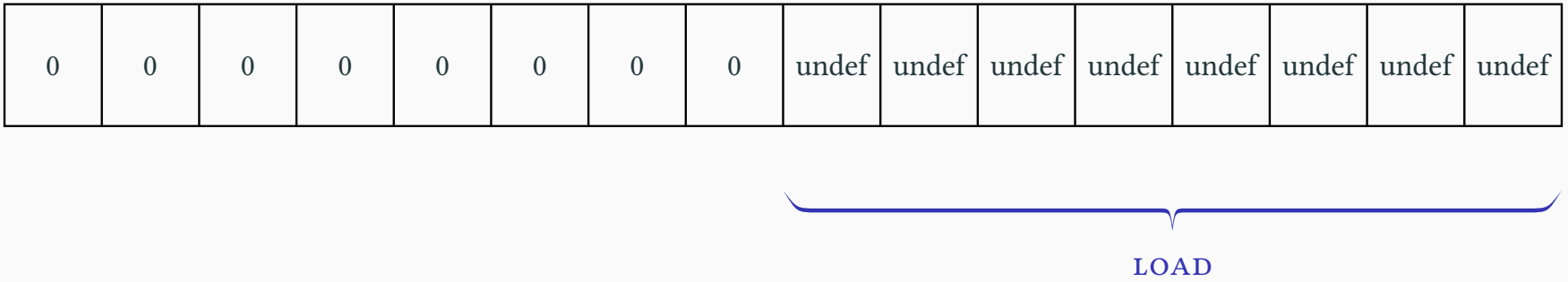
LOAD

```
pub fn foo() -> Result<(), i64> { Ok(()) }
```

```
define { i64, i64 } @foo() {
%_0 = alloca [16 x i8]
store i64 undef, ptr %_0
%0 = getelementptr inbounds i8, ptr %_0, i64 8
store i64 undef, ptr %0
store i64 0, ptr %_0
%1 = load i64, ptr %_0
```

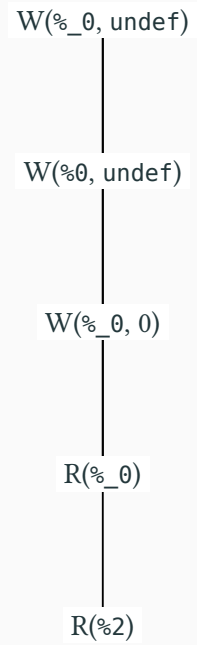


Uninitialised memory




```rust
pub fn foo() -> Result<(), i64> { Ok(()) }
```

```
define { i64, i64 } @foo() {
%_0 = alloca [16 x i8]
store i64 undef, ptr %_0
%0 = getelementptr inbounds i8, ptr %_0, i64 8
store i64 undef, ptr %0
store i64 0, ptr %_0
%1 = load i64, ptr %_0
%2 = getelementptr inbounds i8, ptr %_0, i64 8
%3 = load i64, ptr %2
```



Memory intrinsics

```
@llvm.memcpy(ptr %dest, ptr %src, i64 24)
```

- Performs an untyped copy of 24 bytes from `%src` to `%dest`.
  - `%src` and `%dest` must be either equal or non-overlapping.
  - Transformation phase lowers any `memcpy` with a resolvable length into individual accesses.
  - Intrinsics with unresolvable lengths cause runtime errors if encountered during verification.

Memory intrinsics

```
@llvm.memcpy(ptr %dest, ptr %src, i64 24)
```

- Performs an untyped copy of 24 bytes from `%src` to `%dest`.
  - `%src` and `%dest` must be either equal or non-overlapping.
  - Transformation phase lowers any `memcpy` with a resolvable length into individual accesses.
  - Intrinsics with unresolvable lengths cause runtime errors if encountered during verification.
- Cannot be implemented as a loop of load/store instructions.
  - Loss of provenance information.

Memory intrinsics

```
@llvm.memcpy(ptr %dest, ptr %src, i64 24)
```

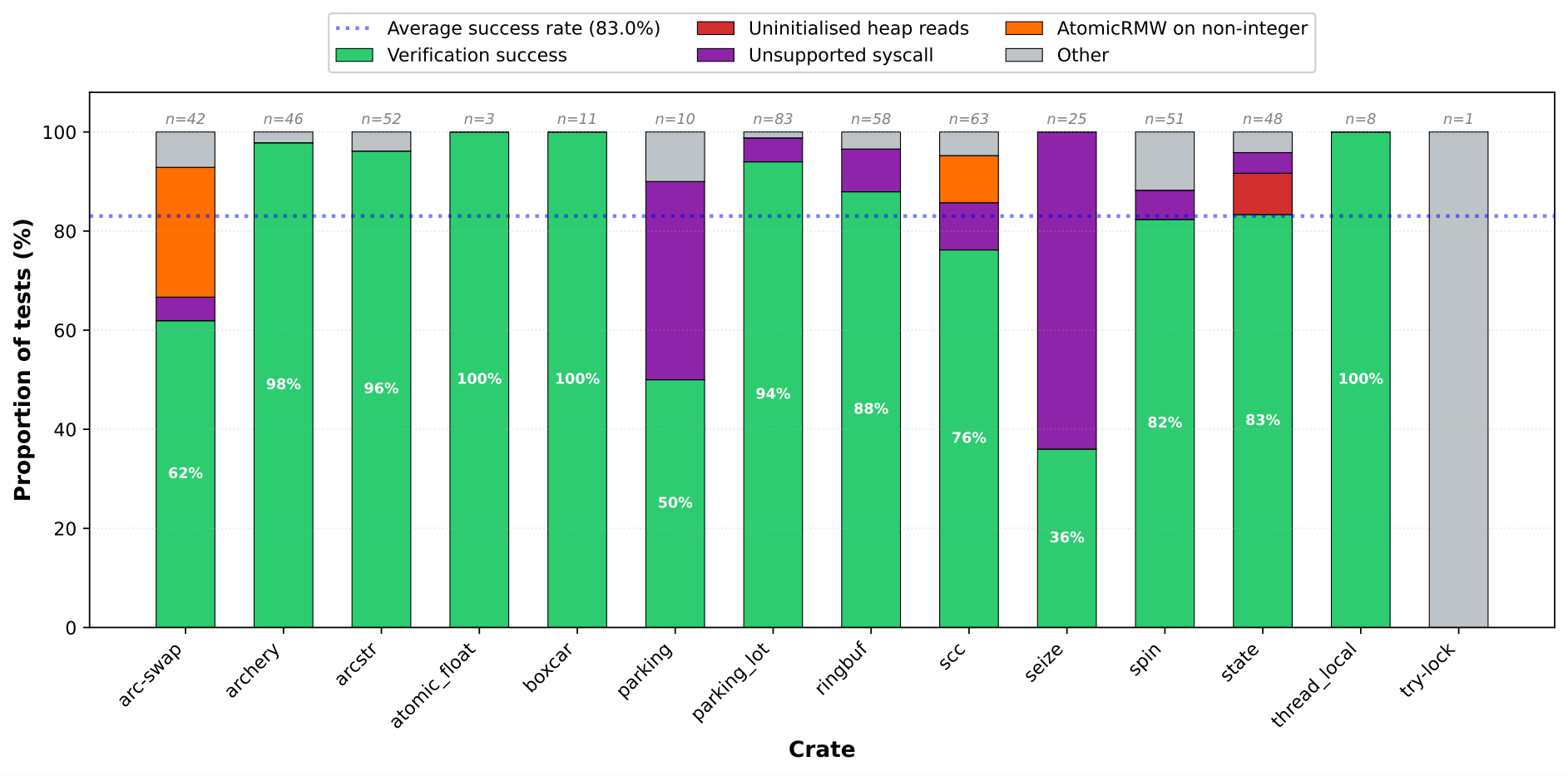
- Performs an untyped copy of 24 bytes from `%src` to `%dest`.
  - `%src` and `%dest` must be either equal or non-overlapping.
  - Transformation phase lowers any `memcpy` with a resolvable length into individual accesses.
  - Intrinsics with unresolvable lengths cause runtime errors if encountered during verification.
- Cannot be implemented as a loop of load/store instructions.
  - Loss of provenance information.
- Interpreter implementation:
  - Invoked at runtime after intrinsic lowering.
  - `memcpy` semantics are preserved for prior passes.

Evaluation

Evaluation goals

- Evaluate RustMC's support for real world Rust language features by verifying test functions from 14 popular concurrency crates
- Assess RustMC's bug detection accuracy on third party benchmarks

Real-world crates

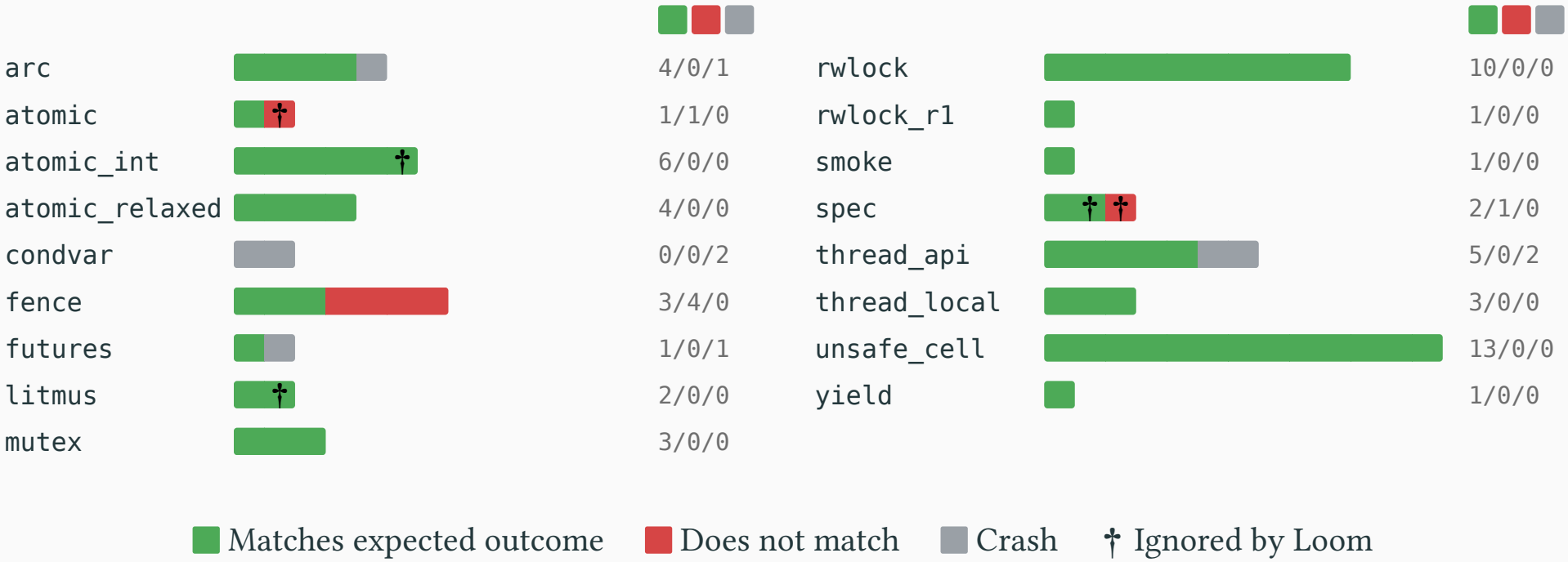


- 83% Support rate across dataset including production grade data structures.
- Verification performed on unmodified test cases.
- Varying success rate reflects diversity of concurrent programming patterns in the Rust ecosystem.

Comparison with Loom

- Assessed bug detection accuracy and performance on 72 concurrent Rust tests taken from the Loom model checker.
  - Operates on the Rust API level while our tool works at the lower LLVM-IR level abstraction.
  - Requires manual porting of tests to the Loom API.

Loom test suite



72 tests — **60** match expected outcome (83%), **6** do not match, **6** crash. RustMC results on tests adapted from Loom.

Conclusion

Key takeaways

- First exhaustive exploration of mixed language concurrent programs state space
- Rust's compilation strategy introduces unique verification challenges at the LLVM-IR level
- Verification plugs into existing Rust test infrastructure.

Future work

- Support for uninitialised heap reads and syscalls
- Directing verification towards high-risk concurrent code paths

Links

Tool Repo:

- https://github.com/Ollie-Pearce/rustmc

Slides:

- https://olliepearce.xyz/presentations/rustmc.pdf

Uninitialised value handling

If a program contains some UB from the use of an uninitialised value, we explore all execution traces in which `undef` is observed as 0.

Alternatives:

- Explore all possible `undef` values.
- Check for uses of `undef` values and terminate early.