Detecting Cross-language Memory Management Issues in Rust

Marco Antonio Corallo

University of Pisa

Course of Languages, Compilers and Interpreters

Overview

Overview

Introduction

Rust is a promising system-level programming language that can prevent memory corruption bugs using its strong type system and *ownership-based* memory management scheme. In practice, programmers usually write Rust code in conjunction with other languages such as C/C++ through *Foreign Function Interface (FFI)*.

Introduction

Rust is a promising system-level programming language that can prevent memory corruption bugs using its strong type system and *ownership-based* memory management scheme. In practice, programmers usually write Rust code in conjunction with other languages such as C/C++ through *Foreign Function Interface (FFI)*.

- Firefox

Introduction

Rust is a promising system-level programming language that can prevent memory corruption bugs using its strong type system and *ownership-based* memory management scheme. In practice, programmers usually write Rust code in conjunction with other languages such as C/C++ through *Foreign Function Interface (FFI)*.

- Firefox
- Google Fuchsia OS

Introduction

Rust is a promising system-level programming language that can prevent memory corruption bugs using its strong type system and *ownership-based* memory management scheme. In practice, programmers usually write Rust code in conjunction with other languages such as C/C++ through *Foreign Function Interface (FFI)*.

- Firefox
- Google Fuchsia OS
- Linux Kernel.

Introduction

Although it is widely believed that gradually re-implementing security-critical components in Rust is a way of enhancing software security, however, using FFI is inherently unsafe.

Introduction

Although it is widely believed that gradually re-implementing security-critical components in Rust is a way of enhancing software security, however, using FFI is inherently unsafe.

- Programmers may accidentally misuse the unsafe abilities that lead to vulnerabilities.

Introduction

Although it is widely believed that gradually re-implementing security-critical components in Rust is a way of enhancing software security, however, using FFI is inherently unsafe.

- Programmers may accidentally misuse the unsafe abilities that lead to vulnerabilities.
- Different assumptions made by different languages make it possible for attackers to maneuver between the FFI boundaries and exploit these vulnerabilities

Introduction

Although it is widely believed that gradually re-implementing security-critical components in Rust is a way of enhancing software security, however, using FFI is inherently unsafe.

- Programmers may accidentally misuse the unsafe abilities that lead to vulnerabilities.
- Different assumptions made by different languages make it possible for attackers to maneuver between the FFI boundaries and exploit these vulnerabilities
- Even for Rust packages written in pure safe Rust, they may still be affected because they may depend on other packages that include FFI.

Introduction

Therefore excluding FFI is unrealistic in the current Rust ecosystem;

Instead, people have made lots of efforts to secure the use of FFI.

Introduction

Therefore excluding FFI is unrealistic in the current Rust ecosystem;

Instead, people have made lots of efforts to secure the use of FFI.

- Some Rust packages to automatically generate FFI, preventing developers from misusing it.

Introduction

Therefore excluding FFI is unrealistic in the current Rust ecosystem;

Instead, people have made lots of efforts to secure the use of FFI.

- Some Rust packages to automatically generate FFI, preventing developers from misusing it.
- Rust community has drafted several guidelines for writing unsafe code, including FFI.

Introduction

Therefore excluding FFI is unrealistic in the current Rust ecosystem;

Instead, people have made lots of efforts to secure the use of FFI.

- Some Rust packages to automatically generate FFI, preventing developers from misusing it.
- Rust community has drafted several guidelines for writing unsafe code, including FFI.

However, they can only help developers to write correct interfaces with appropriate data types.
Memory corruption caused by heap memory allocation/deallocation across the FFI boundaries remains an open problem.

Introduction

Idea: Use static analysis techniques to to keep track of the states of heap memory, that is, while the heap memory is propagated among the control flow graph, we determine whether it is borrowed or moved.

Introduction

Idea: Use static analysis techniques to to keep track of the states of heap memory, that is, while the heap memory is propagated among the control flow graph, we determine whether it is borrowed or moved.

Finally, if any heap memory is passed across the FFI boundaries, we continue to analyze whether it is freed in the external code.

Introduction

Idea: Use static analysis techniques to to keep track of the states of heap memory, that is, while the heap memory is propagated among the control flow graph, we determine whether it is borrowed or moved.

Finally, if any heap memory is passed across the FFI boundaries, we continue to analyze whether it is freed in the external code.

Development of a tool called *FFIChecker*, which automatically collects all the generated *LLVM* intermediate representation (*IR*) for both Rust and C/C++ code, then performs static analysis on the LLVM IR and outputs diagnostic reports.

Overview

Rust

A strongly-typed compiled language, with a rigorous type system and an unique and innovative *ownership* system derived from *linear logic* and *linear types*.

Rust

A strongly-typed compiled language, with a rigorous type system and an unique and innovative *ownership* system derived from *linear logic* and *linear types*.

Each value has a unique *owner* (owner variable), which keeps track of the lifetime of the value. Once the owner variable goes out of its scope, the ownership system automatically releases the memory allocated for the value.

Rust

A strongly-typed compiled language, with a rigorous type system and an unique and innovative *ownership* system derived from *linear logic* and *linear types*.

Each value has a unique *owner* (owner variable), which keeps track of the lifetime of the value. Once the owner variable goes out of its scope, the ownership system automatically releases the memory allocated for the value.

To pass a value to other parts of code, one can either

Rust

A strongly-typed compiled language, with a rigorous type system and an unique and innovative *ownership* system derived from *linear logic* and *linear types*.

Each value has a unique *owner* (owner variable), which keeps track of the lifetime of the value. Once the owner variable goes out of its scope, the ownership system automatically releases the memory allocated for the value.

To pass a value to other parts of code, one can either

- *copy/clone* the owner variable

Rust

A strongly-typed compiled language, with a rigorous type system and an unique and innovative *ownership* system derived from *linear logic* and *linear types*.

Each value has a unique *owner* (owner variable), which keeps track of the lifetime of the value. Once the owner variable goes out of its scope, the ownership system automatically releases the memory allocated for the value.

To pass a value to other parts of code, one can either

- *copy/clone* the owner variable
- *move* the owner variable

Rust

A strongly-typed compiled language, with a rigorous type system and an unique and innovative *ownership* system derived from *linear logic* and *linear types*.

Each value has a unique *owner* (owner variable), which keeps track of the lifetime of the value. Once the owner variable goes out of its scope, the ownership system automatically releases the memory allocated for the value.

To pass a value to other parts of code, one can either

- *copy/clone* the owner variable
- *move* the owner variable
- *borrow* the owner variable

Rust

A strongly-typed compiled language, with a rigorous type system and an unique and innovative *ownership* system derived from *linear logic* and *linear types*.

Each value has a unique *owner* (owner variable), which keeps track of the lifetime of the value. Once the owner variable goes out of its scope, the ownership system automatically releases the memory allocated for the value.

To pass a value to other parts of code, one can either

- *copy/clone* the owner variable
- *move* the owner variable
- *borrow* the owner variable
    - mutable
    - immutable

FFI

As a system-level programming language, Rust can easily collaborate with other languages through the *Foreign Function Interface* (*FFI*).

FFI

As a system-level programming language, Rust can easily collaborate with other languages through the *Foreign Function Interface* (*FFI*).

Integrating Rust code with C/C++ code is prevalent and necessary

FFI

As a system-level programming language, Rust can easily collaborate with other languages through the *Foreign Function Interface* (*FFI*).

Integrating Rust code with C/C++ code is prevalent and necessary

- Many C/C++ projects integrate Rust into existing code-bases to enhance their security.

FFI

As a system-level programming language, Rust can easily collaborate with other languages through the *Foreign Function Interface* (*FFI*).

Integrating Rust code with C/C++ code is prevalent and necessary

- Many C/C++ projects integrate Rust into existing code-bases to enhance their security.
- It can avoid duplicated work and benefit from the rich ecosystem of libraries written in C/C++.

FFI

As a system-level programming language, Rust can easily collaborate with other languages through the *Foreign Function Interface* (*FFI*).

Integrating Rust code with C/C++ code is prevalent and necessary

- Many C/C++ projects integrate Rust into existing code-bases to enhance their security.
- It can avoid duplicated work and benefit from the rich ecosystem of libraries written in C/C++.
- C/C++ can be used for performance-critical scenarios

FFI

As a system-level programming language, Rust can easily collaborate with other languages through the *Foreign Function Interface* (*FFI*).

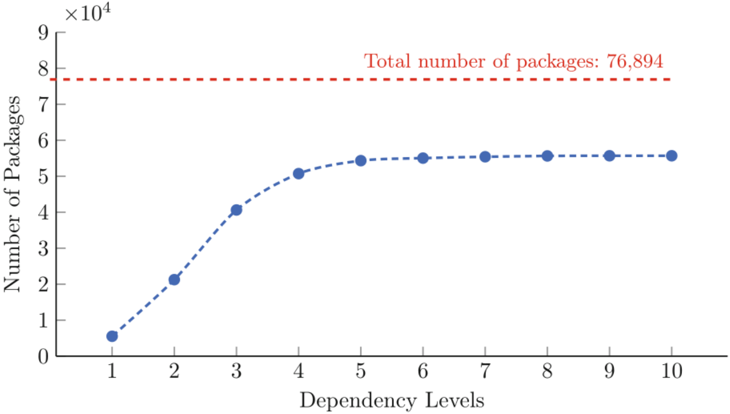
Integrating Rust code with C/C++ code is prevalent and necessary

- Many C/C++ projects integrate Rust into existing code-bases to enhance their security.
- It can avoid duplicated work and benefit from the rich ecosystem of libraries written in C/C++.
- C/C++ can be used for performance-critical scenarios

Since the Rust compiler cannot reason about the security of external code, calling FFI is inherently unsafe.
Programmers need to explicitly use the `unsafe` keyword to bypass the security check enforced by the compiler.

FFI

The incorrect use of the FFI has become a severe source of memory safety bugs. Even if programmers restrict themselves in pure safe Rust, their programs may still implicitly rely on FFI through dependencies.
In fact, more than 72% of packages on the official Rust package registry depend on **at least** one unsafe FFI-bindings package.



FFI

The manual memory management in C/C++ is naively unsafe, so we only consider the case where the heap memory is allocated in Rust and passed to C/C++.

There are two ways of passing a heap-allocated object across FFI:

- By borrowing the object as a reference

FFI

The manual memory management in C/C++ is naively unsafe, so we only consider the case where the heap memory is allocated in Rust and passed to C/C++.

There are two ways of passing a heap-allocated object across FFI:

- By borrowing the object as a reference
  the ownership remains on the Rust side, so the ownership system is responsible for releasing the memory after it goes out of its scope.

FFI

The manual memory management in C/C++ is naively unsafe, so we only consider the case where the heap memory is allocated in Rust and passed to C/C++.

There are two ways of passing a heap-allocated object across FFI:

- By borrowing the object as a reference
  the ownership remains on the Rust side, so the ownership system is responsible for releasing the memory after it goes out of its scope.
- By moving the ownership to the FFI

FFI

The manual memory management in C/C++ is naively unsafe, so we only consider the case where the heap memory is allocated in Rust and passed to C/C++.

There are two ways of passing a heap-allocated object across FFI:

- By borrowing the object as a reference
  the ownership remains on the Rust side, so the ownership system is responsible for releasing the memory after it goes out of its scope.
- By moving the ownership to the FFI
  one can first *forget* it from the ownership system, then pass it to the FFI via a raw pointer.
  The responsibility of memory management returns back to the programmers.

Overview

Security and Memory Management Issues via FFI

To explain why the memory management across the FFI boundaries may lead to security vulnerabilities and how the Rust ownership system gets involved, we give several bug examples detected by FFIChecker

Security and Memory Management Issues via FFI

To explain why the memory management across the FFI boundaries may lead to security vulnerabilities and how the Rust ownership system gets involved, we give several bug examples detected by FFIChecker

- Common Memory Corruption

Security and Memory Management Issues via FFI

To explain why the memory management across the FFI boundaries may lead to security vulnerabilities and how the Rust ownership system gets involved, we give several bug examples detected by FFIChecker

- Common Memory Corruption
- Exception Safety

Security and Memory Management Issues via FFI

To explain why the memory management across the FFI boundaries may lead to security vulnerabilities and how the Rust ownership system gets involved, we give several bug examples detected by FFIChecker

- Common Memory Corruption
- Exception Safety
- Undefined Behaviour caused by Mixing Memory Management Mechanisms

Memory Corruption

When heap memory is passed across the FFI boundaries, the ownership system cannot guarantee its safety. Therefore the responsibility of memory management returns back to the programmers, meaning that all kinds of common memory corruption bugs that happen in C, like *use-after-free*, *double free*, and *memory leak*, still exist.

Memory Corruption

```
let mut cost = Vec::with_capacity(X.rows());
for x in X.outer_iter() {
  let mut cost_i = Vec::with_capacity(Y.rows());  // Allocate a vector
  for y in Y.outer_iter() {
    cost_i.push(distance(&x, &y) as c_double);
  }
  // Forget the memory using `Box::into_raw`
  cost.push(Box::into_raw(cost_i.into_boxed_slice()) as *const c_double);
}

// Call FFI function
let d = unsafe { emd(X.rows(), weight_x.as_ptr(), Y.rows(), weight_y.as_ptr(), cost.as_ptr(), null()) };
```

Memory Corruption

```
let mut cost = Vec::with_capacity(X.rows());
for x in X.outer_iter() {
  let mut cost_i = Vec::with_capacity(Y.rows());  // Allocate a vector
  for y in Y.outer_iter() {
    cost_i.push(distance(&x, &y) as c_double);
  }
  // Forget the memory using `Box::into_raw`
  cost.push(Box::into_raw(cost_i.into_boxed_slice()) as *const c_double);
}

// Call FFI function
let d = unsafe { emd(X.rows(), weight_x.as_ptr(), Y.rows(), weight_y.as_ptr(), cost.as_ptr(), null())
↪  };
```

- Box is a *smart pointer* used to securely manage heap memory.

Memory Corruption

```
let mut cost = Vec::with_capacity(X.rows());
for x in X.outer_iter() {
  let mut cost_i = Vec::with_capacity(Y.rows());  // Allocate a vector
  for y in Y.outer_iter() {
    cost_i.push(distance(&x, &y) as c_double);
  }
  // Forget the memory using `Box::into_raw`
  cost.push(Box::into_raw(cost_i.into_boxed_slice()) as *const c_double);
}

// Call FFI function
let d = unsafe { emd(X.rows(), weight_x.as_ptr(), Y.rows(), weight_y.as_ptr(), cost.as_ptr(), null())
↪  };
```

- Box is a *smart pointer* used to securely manage heap memory.
- `Box::into_raw` expose the raw pointer of the heap memory managed by the Box in order to pass it to the FFI. The ownership system will *forget* the memory and will not reclaim it.
  The developer is responsible for releasing the memory.
  Otherwise, there will be a memory leak.

Exception Safety

Rust does not support the `try-catch` statement for catching exceptions. Instead, Rust provides a more reliable error handling mechanism: all *recoverable* errors must be handled or propagated back to the caller function, and all *unrecoverable* errors are handled by terminating the execution and unwinding the stack.
All the stack objects' destructors will be called during the stack unwinding to prevent resource leakage.

Exception Safety

When cooperating with external code, developers usually have to transiently create unsound states via unsafe code . Then after the external code finishes, developers manually clean up the states. If some error happens in between, the execution stops and the stack is unwound, so the clean-up procedure will not be executed. The remaining unsound state may cause security issues.

Exception Safety

```
1   pub fn bind(&mut self, params: impl IntoParams) -> Result<(), TaosError> {
2     let params = params.into_params();
3     unsafe {
4       let res = taos_stmt_bind_param(self.stmt, params.as_ptr() as _);
5       self.err_or(res)?;
6       let res = taos_stmt_add_batch(self.stmt);
7       self.err_or(res)?;
8     }
9     for mut param in params {
10      unsafe { param.free() };
11    }
12    Ok(())
13  }
```

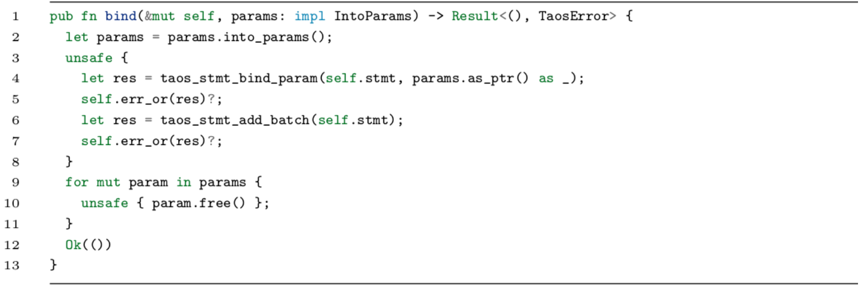
- line 2: variable `params` is initialized by allocating heap memory.

Exception Safety

```
1  pub fn bind(&mut self, params: impl IntoParams) -> Result<(), TaosError> {
2    let params = params.into_params();
3    unsafe {
4      let res = taos_stmt_bind_param(self.stmt, params.as_ptr() as _);
5      self.err_or(res)?;
6      let res = taos_stmt_add_batch(self.stmt);
7      self.err_or(res)?;
8    }
9    for mut param in params {
10     unsafe { param.free() };
11   }
12   Ok(())
13 }
```

- line 2: variable `params` is initialized by allocating heap memory.
- line 3-8: memory is passed to FFI in the `unsafe` block.

Exception Safety



```
1   pub fn bind(&mut self, params: impl IntoParams) -> Result<(), TaosError> {
2     let params = params.into_params();
3     unsafe {
4       let res = taos_stmt_bind_param(self.stmt, params.as_ptr() as _);
5       self.err_or(res)?;
6       let res = taos_stmt_add_batch(self.stmt);
7       self.err_or(res)?;
8     }
9     for mut param in params {
10      unsafe { param.free() };
11    }
12    Ok(())
13  }
```

- line 2: variable `params` is initialized by allocating heap memory.
- line 3-8: memory is passed to FFI in the `unsafe` block.
- line 5 and line 7: the `?` means that if the operation fails, the function returns early and propagates the error to the caller.

Exception Safety

```
 1  pub fn bind(&mut self, params: impl IntoParams) -> Result<(), TaosError> {
 2    let params = params.into_params();
 3    unsafe {
 4      let res = taos_stmt_bind_param(self.stmt, params.as_ptr() as _);
 5      self.err_or(res)?;
 6      let res = taos_stmt_add_batch(self.stmt);
 7      self.err_or(res)?;
 8    }
 9    for mut param in params {
10      unsafe { param.free() };
11    }
12    Ok(())
13  }
```

- line 2: variable `params` is initialized by allocating heap memory.
- line 3-8: memory is passed to FFI in the `unsafe` block.
- line 5 and line 7: the `?` means that if the operation fails, the function returns early and propagates the error to the caller.
- the memory may be leaked if the function returns early and the `free` at line 10 will not be called.

Mixing Memory Management Mechanisms

One possible error is mixing different memory allocation/deallocation procedures provided by different languages.
For example, it is illegal to allocate memory on the Rust-side using `Box` and release it on the C-side using `free`.

Mixing Memory Management Mechanisms

One possible error is mixing different memory allocation/deallocation procedures provided by different languages.
For example, it is illegal to allocate memory on the Rust-side using `Box` and release it on the C-side using `free`.
Mixing different memory management mechanisms is undefined behavior

- Rust and C may use different memory allocators

Mixing Memory Management Mechanisms

One possible error is mixing different memory allocation/deallocation procedures provided by different languages.
For example, it is illegal to allocate memory on the Rust-side using `Box` and release it on the C-side using `free`.
Mixing different memory management mechanisms is undefined behavior

- Rust and C may use different memory allocators
- Rust and C have totally different memory management mechanisms and they operate on different levels.

Mixing Memory Management Mechanisms

```
1   // Rust code:
2   pub unsafe extern "C" fn to_json(from: ext::Ext, text: *const c_char) -> *const c_char {
3     ... ...
4     // CString internally allocates heap memory
5     let output = CString::new(ext::json::serialize(&value.unwrap()).unwrap()).unwrap();
6     let ptr = output.as_ptr();
7     mem::forget(output);  // Memory is "forgotten" by the ownership system
8     ptr  // The raw pointer will be passed across the FFI boundary
9   }
10
11  // C code:
12  int main() {
13    ... ...
14    const char* output = to_json(Yaml, input);
15    ... ...
16    free((char*)output);  // Memory allocated in Rust is freed by free()
17    return 0;
18  }
```

- line 5: a string is constructed through `CString::new`, which uses Rust's memory allocator for heap.

Mixing Memory Management Mechanisms

```
1   // Rust code:
2   pub unsafe extern "C" fn to_json(from: ext::Ext, text: *const c_char) -> *const c_char {
3     ... ...
4     // CString internally allocates heap memory
5     let output = CString::new(ext::json::serialize(&value.unwrap()).unwrap()).unwrap();
6     let ptr = output.as_ptr();
7     mem::forget(output);  // Memory is "forgotten" by the ownership system
8     ptr  // The raw pointer will be passed across the FFI boundary
9   }
10
11  // C code:
12  int main() {
13    ... ...
14    const char* output = to_json(Yaml, input);
15    ... ...
16    free((char*)output);  // Memory allocated in Rust is freed by free()
17    return 0;
18  }
```

- line 5: a string is constructed through `CString::new`, which uses Rust's memory allocator for heap.
- line 7-8: the string is explicitly leaked by `mem::forget` and a raw pointer is returned.

Mixing Memory Management Mechanisms

```
1   // Rust code:
2   pub unsafe extern "C" fn to_json(from: ext::Ext, text: *const c_char) -> *const c_char {
3     ... ...
4     // CString internally allocates heap memory
5     let output = CString::new(ext::json::serialize(&value.unwrap()).unwrap()).unwrap();
6     let ptr = output.as_ptr();
7     mem::forget(output);  // Memory is "forgotten" by the ownership system
8     ptr  // The raw pointer will be passed across the FFI boundary
9   }
10
11  // C code:
12  int main() {
13    ... ...
14    const char* output = to_json(Yaml, input);
15    ... ...
16    free((char*)output);  // Memory allocated in Rust is freed by free()
17    return 0;
18  }
```

- line 5: a string is constructed through `CString::new`, which uses Rust's memory allocator for heap.
- line 7-8: the string is explicitly leaked by `mem::forget` and a raw pointer is returned.
- line 16: the heap memory is freed by C's `free`.

Overview

1. Introduction
2. Background
3. Security and Memory Management Issues via FFI
4. **Abstract Interpretation**
5. Algorithms
6. Evaluation and Conclusion
7. Conclusion

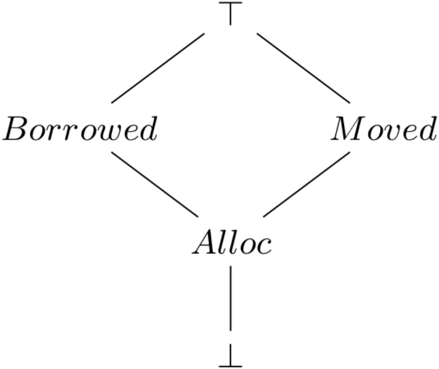
Abstract Values

Let **Var** as the set of all the variables in the CFG
and Block as the set of all basic blocks in the CFG.

Abstract Values

Let **Var** as the set of all the variables in the CFG
and Block as the set of all basic blocks in the CFG.

Let **MState** as the lattice



Abstract Domain

To keep track of the abstract values for each basic block, we maintain a lookup table σ_b : **Var** $\rightarrow$ **MState** for each basic block *b*.
We define **AState** as a map lattice constisting of all the mappings from **Var** to **MState**.

Abstract Domain

To keep track of the abstract values for each basic block, we maintain a lookup table $\sigma_b : \textbf{Var} \rightarrow \textbf{MState}$ for each basic block b.
We define **AState** as a map lattice consisting of all the mappings from **Var** to **MState**.

AState is still a lattice, with
$\sqsubseteq$: For $\sigma_1, \sigma_2 \in \textbf{AState}, \sigma_1 \sqsubseteq \sigma_2 \iff \forall a \in \textbf{Var}, \sigma_1(a) \sqsubseteq \sigma_2(a)$.

Abstract Domain

To keep track of the abstract values for each basic block, we maintain a lookup table $\sigma_b : \mathbf{Var} \to \mathbf{MState}$ for each basic block b.
We define **AState** as a map lattice consisting of all the mappings from **Var** to **MState**.

AState is still a lattice, with
$\sqsubseteq$: For $\sigma_1, \sigma_2 \in \mathbf{AState}, \sigma_1 \sqsubseteq \sigma_2 \iff \forall a \in \mathbf{Var}, \sigma_1(a) \sqsubseteq \sigma_2(a)$.
$\sqcup : \forall \sigma_1, \sigma_2 \in \mathbf{AState}, \sigma_1 \sqcup \sigma_2 = (a, \sigma_1(a) \sqcup \sigma_2(a)) : \forall a \in \mathbf{Var}$

Abstract Domain

To keep track of the abstract values for each basic block, we maintain a lookup table $\sigma_b : \textbf{Var} \rightarrow \textbf{MState}$ for each basic block b.
We define **AState** as a map lattice consisting of all the mappings from **Var** to **MState**.

AState is still a lattice, with
$\sqsubseteq$: For $\sigma_1, \sigma_2 \in \textbf{AState}, \sigma_1 \sqsubseteq \sigma_2 \iff \forall a \in \textbf{Var}, \sigma_1(a) \sqsubseteq \sigma_2(a)$.
$\sqcup : \forall \sigma_1, \sigma_2 \in \textbf{AState}, \sigma_1 \sqcup \sigma_2 = (a, \sigma_1(a) \sqcup \sigma_2(a)) : \forall a \in \textbf{Var}$

Finally, the *Abstract Domain* is defined as a mapping from **Block** to **AState**.

Transfer Functions

Since the analysis runs on LLVM IR, there is a transfer function for each LLVM instruction according to its semantics.
In particular, we focus on

- `load`

```
fn analyze_load(&mut self, load: &Load) {
    // dest <- address
    let address = &load.address;
    let dest = &load.dest;
    match address {
        Operand::LocalOperand { name, .. } => {
            self.state.propagate_taint(name, dest);
        }
        _ => (),
    }
}
```

Transfer Functions

Since the analysis runs on LLVM IR, there is a transfer function for each LLVM instruction according to its semantics.
In particular, we focus on

- `load`
- `store`

```
fn analyze_store(&mut self, store: &Store) {
    // address <- value
    let address = &store.address;
    let value = &store.value;
    match (address, value) {
        (
            Operand::LocalOperand {
                name: address_name, ..
            },
            Operand::LocalOperand {
                name: value_name, ..
            },
        ) => {
            self.state.propagate_taint(value_name, address_name);
        }
        _ => (),
    }
}
```

Transfer Functions

Since the analysis runs on LLVM IR, there is a transfer function for each LLVM instruction according to its semantics.
In particular, we focus on

- `load`
- `store`
- `GetElementPtr`

```
fn analyze_getelementptr(&mut self, getelementptr: &GetElementPtr) {
    // dest <- address
    let address = &getelementptr.address;
    let dest = &getelementptr.dest;
    if let Operand::LocalOperand {
        name: value_name, ..
    } = address
    {
        self.state.propagate_taint(value_name, dest);
    }
}
```

Transfer Functions

Since the analysis runs on LLVM IR, there is a transfer function for each LLVM instruction according to its semantics.
In particular, we focus on

- `load`
- `store`
- `GetElementPtr`
- `Call`
- `Invoke`

Function Calls

When analyzing instructions that call other functions, such as `Call` and `Invoke`, the analysis performs *context-sensitive* interprocedural analysis: different functions need different treatments. In particular:

Function Calls

When analyzing instructions that call other functions, such as `Call` and `Invoke`, the analysis performs *context-sensitive* interprocedural analysis: different functions need different treatments. In particular:

- Functions that allocate heap memory:

Function Calls

When analyzing instructions that call other functions, such as `Call` and `Invoke`, the analysis performs *context-sensitive* interprocedural analysis: different functions need different treatments. In particular:

- Functions that allocate heap memory:
  taint source of the algorithm: the resulting variable stores heap memory, so its abstract state will be *Alloc*.
- Functions that *borrow* a reference or *move* the ownership:

Function Calls

When analyzing instructions that call other functions, such as `Call` and `Invoke`, the analysis performs *context-sensitive* interprocedural analysis: different functions need different treatments. In particular:

- Functions that allocate heap memory:
  taint source of the algorithm: the resulting variable stores heap memory, so its abstract state will be *Alloc*.
- Functions that *borrow* a reference or *move* the ownership:
  these functions change the abstract state of heap memory into either *Borrowed* or *Moved*.
- Foreign functions called through FFI:

Function Calls

When analyzing instructions that call other functions, such as `Call` and `Invoke`, the analysis performs *context-sensitive* interprocedural analysis: different functions need different treatments. In particular:

- Functions that allocate heap memory:
  taint source of the algorithm: the resulting variable stores heap memory, so its abstract state will be *Alloc*.
- Functions that *borrow* a reference or *move* the ownership:
  these functions change the abstract state of heap memory into either *Borrowed* or *Moved*.
- Foreign functions called through FFI:
  potentially vulnerable functions, analyze these functions and see whether there are any bugs.

Overview

1. Introduction
2. Background
3. Security and Memory Management Issues via FFI
4. Abstract Interpretation
5. Algorithms
6. Evaluation and Conclusion
7. Conclusion

Fixed-Point Algorithm

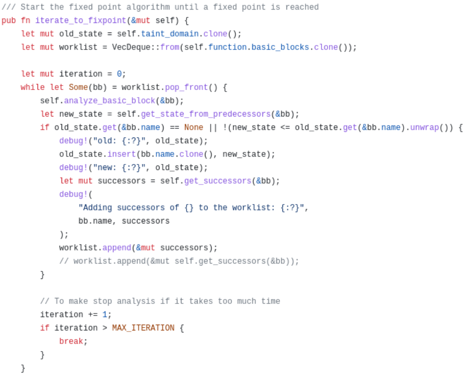
FFIChecker traverses a given CFG and iteratively runs transfer functions to update the abstract state until it reaches a fixed point. The fixed-point algorithm chosen is the classical *worklist* algorithm.

Fixed-Point Algorithm

FFIChecker traverses a given CFG and iteratively runs transfer functions to update the abstract state until it reaches a fixed point. The fixed-point algorithm chosen is the classical *worklist* algorithm.

Algorithm 1: Fixed-point algorithm for FFIChecker

```
Input: Control Flow Graph: $CFG$
Output: Abstract State: $State$
Initialization: $State[n] \leftarrow \bot$ for all $n$
Function FixedPoint($CFG$, $State$):
    $W \leftarrow CFG.basicblocks$
    while $W \neq \emptyset$ do
        $b \leftarrow W.$remove()
        foreach $instr \in b.instructions$ do
            Transfer($State[b], instr$)
        Transfer($State[b], b.terminator$)
        $new\_state \leftarrow \bigsqcup_{n \in \texttt{Predecessors}(b)} State[n]$
        if $new\_state \not\sqsubseteq State[b]$ then
            $State[b] \leftarrow new\_state$
            foreach $v \in$ Successors($b$) do
                $W.$insert($v$)
    return $State$
```



```rust
/// Start the fixed point algorithm until a fixed point is reached
pub fn iterate_to_fixpoint(&mut self) {
    let mut old_state = self.taint_domain.clone();
    let mut worklist = VecDeque::from(self.function.basic_blocks.clone());

    let mut iteration = 0;
    while let Some(bb) = worklist.pop_front() {
        self.analyze_basic_block(&bb);
        let new_state = self.get_state_from_predecessors(&bb);
        if old_state.get(&bb.name) == None || !(new_state <= old_state.get(&bb.name).unwrap()) {
            debug!("old: {:?}", old_state);
            old_state.insert(bb.name.clone(), new_state);
            debug!("new: {:?}", old_state);
            let mut successors = self.get_successors(&bb);
            debug!(
                "Adding successors of {} to the worklist: {:?}",
                bb.name, successors
            );
            worklist.append(&mut successors);
            // worklist.append(&mut self.get_successors(&bb));
        }

        // To make stop analysis if it takes too much time
        iteration += 1;
        if iteration > MAX_ITERATION {
            break;
        }
    }
```

Context-Sensitive Interprocedural Analysis

To avoid duplicated analysis for the same function, FFIChecker implements a *summary-based* method: it caches previously computed results (*summaries*) in a lookup table cache : *((f, in_state), out_state)* that maps a calling context *(f, in_state)* to an output *out_state*.

Context-Sensitive Interprocedural Analysis

To avoid duplicated analysis for the same function, FFIChecker implements a *summary-based* method: it caches previously computed results (*summaries*) in a lookup table cache : *((f, in_state), out_state)* that maps a calling context *(f, in_state)* to an output *out_state*.

Before analyzing a function, it first check whether there is an existing summary that has been computed. If it is the case, the fixed-point algorithm is skipped and the result is directly returned. If not, the fixed-point algorithm is performed and the analysis result is cached in the lookup table.

Overview

1. Introduction
2. Background
3. Security and Memory Management Issues via FFI
4. Abstract Interpretation
5. Algorithms
6. Evaluation and Conclusion
7. Conclusion

Evaluation

FFIChecker also tag a *confidence level* on each generated warning, depending on how much information it can leverage during the analysis.

Evaluation

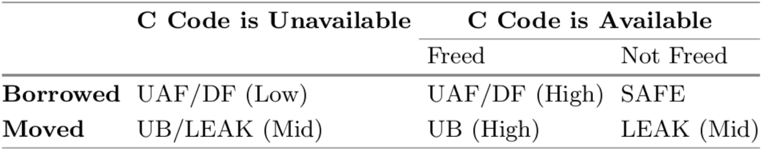
FFIChecker also tag a *confidence level* on each generated warning, depending on how much information it can leverage during the analysis.

For example, the LLVM IR of a foreign function is not always available because it may come from a dynamically linked C library. In this case, FFIChecker cannot further analyze the foreign function, so it generates warnings with **lower** confidence.
This design helps to suppress false alarms.

Evaluation

FFIChecker also tag a *confidence level* on each generated warning, depending on how much information it can leverage during the analysis.

For example, the LLVM IR of a foreign function is not always available because it may come from a dynamically linked C library. In this case, FFIChecker cannot further analyze the foreign function, so it generates warnings with **lower** confidence.
This design helps to suppress false alarms.



	C Code is Unavailable	**C Code is Available**	
		Freed	Not Freed
Borrowed	UAF/DF (Low)	UAF/DF (High)	SAFE
Moved	UB/LEAK (Mid)	UB (High)	LEAK (Mid)

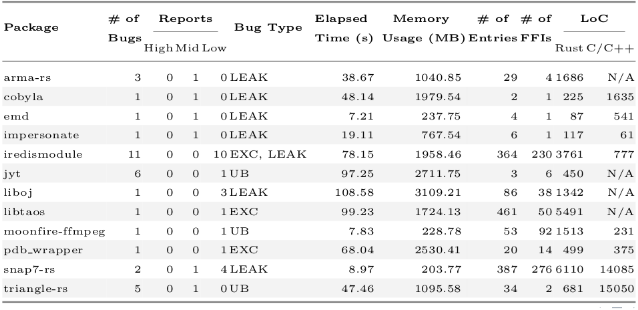
Evaluation

For testing FFIChecker has been collected 987 packages that are of category *external-ffi-bindings* or depend on other packages that assist the use of FFI, for a total of 3,232,574 lines of Rust and 46,321,573 lines of C/C++.

Evaluation

For testing FFIChecker has been collected 987 packages that are of category *external-ffi-bindings* or depend on other packages that assist the use of FFI, for a total of 3,232,574 lines of Rust and 46,321,573 lines of C/C++.

On this dataset, FFIChecker generates 222 warnings. Manually inspect them, 34 bugs (19 memory leaks, 3 exception-related bugs and 12 undefined behaviours) has been confirmed.



Package	# of Bugs	Reports High	Reports Mid	Reports Low	Bug Type	Elapsed Time (s)	Memory Usage (MB)	# of Entries	# of FFIs	LoC Rust	LoC C/C++
arma-rs	3	0	1	0	LEAK	38.67	1040.85	29	4	1686	N/A
cobyla	1	0	1	0	LEAK	48.14	1979.54	2	1	225	1635
emd	1	0	1	0	LEAK	7.21	237.75	4	1	87	541
impersonate	1	0	1	0	LEAK	19.11	767.54	6	1	117	61
iredismodule	11	0	0	10	EXC, LEAK	78.15	1958.46	364	230	3761	777
jyt	6	0	0	1	UB	97.25	2711.75	3	6	450	N/A
liboj	1	0	0	3	LEAK	108.58	3109.21	86	38	1342	N/A
libtaos	1	0	0	1	EXC	99.23	1724.13	461	50	5491	N/A
moonfire-ffmpeg	1	0	0	1	UB	7.83	228.78	53	92	1513	231
pdb_wrapper	1	0	0	1	EXC	68.04	2530.41	20	14	499	375
snap7-rs	2	0	1	4	LEAK	8.97	203.77	387	276	6110	14085
triangle-rs	5	0	1	0	UB	47.46	1095.58	34	2	681	15050

Overview

Conclusion

Most bugs we found are memory leaks. We interpret this as a limitation of Rust's security guarantees: memory leak is considered *safe* in Rust. The reason behind this design choice is that leaking resources is possible in pure safe Rust Therefore, the authors of the Rust standard library decide not to mark functions that leak memory as unsafe.

Conclusion

Most bugs we found are memory leaks. We interpret this as a limitation of Rust's security guarantees: memory leak is considered *safe* in Rust. The reason behind this design choice is that leaking resources is possible in pure safe Rust Therefore, the authors of the Rust standard library decide not to mark functions that leak memory as unsafe.

As a result, the Rust compiler will not give any warnings when inexperienced programmers misuse these functions and cause memory leaks, leading to denial of service attacks or information leakage.

Based on the work of Z. Li, J. Wang, M. Sun and J. C. S. Lui

Thank You.