

Refactoring C for Automated Translation into Safe and Idiomatic Rust

by

Maya Rebholz

S.B. Computer Science and Engineering, Massachusetts Institute of Technology, 2025

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2026

© 2026 Maya Rebholz. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Maya Rebholz
Department of Electrical Engineering and Computer Science
May 15, 2026

Certified by: Jeff Perkins
Research Scientist, Thesis Supervisor

Certified by: Martin Rinard
Professor of Computer Science and Engineering, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Refactoring C for Automated Translation into Safe and Idiomatic Rust

by

Maya Rebholz

Submitted to the Department of Electrical Engineering and Computer Science
on May 15, 2026 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE

ABSTRACT

Automating the translation of unsafe legacy C code into safe Rust requires a deep understanding of the program’s core logic and aggressive restructuring of C patterns into idiomatic Rust. The main existing translation tool, C2Rust, however, does a simple line-by-line translation and even efforts to improve this fall short of generating clean Rust. Tenjin is a new project from Aarno labs and MIT that aims to redesign the conversion into a multistage process that can fully leverage Rust’s safety. This thesis focuses on the first stage of this process, *preparatory refactoring*, which transforms complex and potentially unsafe C into C that is more compatible with Rust. Two specific refactoring passes were built that targeted the following unsafe C memory patterns: union punning (a hack to reinterpret data as a different type) and pointers used in arrays. The union punning pass detects valid punning candidates and rewrites them as explicit, isolated conversions that can be later mapped to safe Rust bit operations. The pointer arithmetic pass transforms raw pointers into array index expressions within and across functions such that they are ready to be translated into their Rust slice or reference equivalents. These conversions simplify the later analysis stages in Tenjin and enable the generation of idiomatic Rust. Tested on a given research test corpus as well as on public repositories, results for both passes show coverage over the vast majority of cases and behavioral correctness for generated code. The passes create robust, modular code that strengthens the pipeline’s ability to reason about and restructure low level C patterns.

Thesis supervisor: Jeff Perkins

Title: Research Scientist

Thesis supervisor: Martin Rinard

Title: Professor of Computer Science and Engineering

Acknowledgments

I would like to thank my supervisors Professor Martin Rinard and Jeff Perkins. Professor Martin Rinard offered me the chance to do research with him in CSAIL and introduced me to Jeff and Aarno Labs. Jeff Perkins helped me greatly with his support and advice to me throughout my research. Through his feedback and guidance, my research became more robust, well-tested, and clear.

Additionally, I would like to thank the researchers at Aarno Labs, specifically Ben Karel, who helped me integrate my passes into the Tenjin pipeline and supplied me with the TRACTOR test cases to ensure the correctness of my work.

Contents

<i>List of Tables</i>	9
<i>List of Listings</i>	11
1 Introduction	13
1.1 Motivation of Translating C to Rust	13
1.2 Translation Strategy	14
1.3 Preparatory Refactoring	14
1.4 Union Punning Pass	15
1.5 Pointer Pass	16
2 Related Work	19
2.1 C2Rust	19
2.2 Pointer Translation	19
3 Union Punning Pass	21
3.1 Methods	21
3.1.1 Preprocessing and Parsing the File	21
3.1.2 Union Detection and Access Collection	22
3.1.3 Access Pattern Validation	22
3.1.4 Source Code Rewriting	25
3.1.4.1 analyzeCutPoint	25
3.1.4.2 generateTemporaryVariables	25
3.1.4.3 generateConversionFunctionCode and generateConversionCall	26
3.1.4.4 handleInitOnlyCase	26
3.1.4.5 collectReplacementEdits	27
3.1.4.6 applyEdits	27
3.1.5 Transformation Examples	27
3.1.6 Rejected Code Examples	31
3.2 Results	33
3.3 Future Work	34
4 Pointer Pass	35
4.1 Methods	35
4.1.1 Pointer Collection and Access Classification	35
4.1.1.1 Base Array Identification	35
4.1.1.2 Access Classification	36

4.1.2	Validation	38
4.1.3	Within-Function Transformation	38
4.1.3.1	Declaration	38
4.1.3.2	Dereference	38
4.1.3.3	Structured access	39
4.1.3.4	Mutations	39
4.1.3.5	Assignments	39
4.1.3.6	Comparisons	39
4.1.3.7	Boolean contexts	40
4.1.3.8	Passing to non-transformed functions	40
4.1.4	Special RustSlice Transformation	40
4.1.4.1	Detection	40
4.1.4.2	Typedef emission	40
4.1.4.3	Signature rewrite	40
4.1.4.4	Body transformation	41
4.1.4.5	Call site rewriting	41
4.1.4.5.1	From non-transformed code	41
4.1.4.5.2	From another transformed function (pass-through)	41
4.1.4.5.3	From another transformed function (subslices)	41
4.1.4.6	Pointer pairs and inclusive end	41
4.1.4.7	Singleton callees	41
4.1.4.8	Propagation To Other Functions in Caller/Callee Chain	42
4.1.5	Full Transformation Example On Quicksort	42
4.2	Results	43
4.3	Future Work	44
5	Conclusion	47
	<i>References</i>	49

List of Tables

3.1	Number of Identified, Replaced, and Rejected Unions By Repository	33
4.1	Number of Identified, Replaced, and Rejected Pointers By Repository	44

List of Listings

1.1	Union Punning Example	15
1.2	Tenjin Union Snippet Before Pass	15
1.3	Tenjin Union Snippet After Pass	15
1.4	Invalid Union Punning Candidate	16
1.5	Basic Pointer Example Before Pass	16
1.6	Basic Pointer Example After Pass	17
3.1	Valid Access Patterns	23
3.2	Invalid Access Patterns	24
3.3	Typedef Generation	25
3.4	Standard Conversion Function	26
3.5	Initialization Special Case	27
3.6	Control Flow Example	28
3.7	Array Example	28
3.8	Macro Example	29
3.9	Loop Example	30
3.10	Rejected Code Examples	31
4.1	Quicksort Before Pass	42
4.2	Quicksort After Pass	43
4.3	Rejected Sentinel Pointer	44

Chapter 1

Introduction

1.1 Motivation of Translating C to Rust

There are many things software engineers want from their code and almost always safety and correctness are at the top. Programming bugs can unfortunately slip into code bases, though, and result in incorrect data, attacks and crashes. To prevent bugs from sneaking into deployed code, a good strategy is to simply minimize the bugs that are written in the first place. How this prevention is achieved when writing code depends heavily on the specific programming language used. Languages that are higher level tend to have fewer bugs because more details (and opportunities for mistakes) are abstracted away from the developer. This abstraction, however, is also a limitation as the developer has less flexibility to tailor the code to their specific goal. For projects that demand highly performant code, programmers often sacrifice safety and choose lower level languages.

The programming language C is an often picked lower level language because of its historical prevalence and proven performance. With the growth of the newer language Rust, however, programmers no longer need to trade safety for speed. Rust's structure inherently provides a safer interface, while maintaining fast execution time. Specifically, Rust has an ownership structure that ensures strong memory safety, while still giving programmers low level control and performance. Further, its preference towards explicit array management through slices and references compared to C's implicit pointer usage adds clarity and protection from overflow issues. This redesigned approach to memory management is extremely important as almost 70% of vulnerabilities arise from memory safety problems [2].

Even though many now recognize Rust's benefits, it is not easy for programmers to suddenly switch to a new language when all their existing code is in another language. Most projects accumulate large interdependent code bases over years and manually rewriting legacy C into Rust would be painful. This is the motivation of automated translation of C code to Rust code, the center of this research project.

1.2 Translation Strategy

This research was conducted as a part of the larger joint Aarno Labs¹ and MIT Tenjin research project for the Darpa TRACTOR program². The goal of Tenjin is to design a novel C to Rust translation pipeline. Tenjin’s translation occurs in 5 main stages:

1. *Preparatory Refactoring* translates input C source code into functionally equivalent, but more Rust compatible C code
2. *Prealysis* expands the understanding of the C code’s behavior
3. *Translation* converts the code into safe Rust using the knowledge accumulated in previous steps
4. *Improvement* reforms the outputted Rust code to be more idiomatic
5. *Validation* tests the final Rust code for behavioral equivalence with the input C code.

The multistage pipeline is needed because the two different languages have inherently different structures, so generating idiomatic Rust code cannot rely on outputting statement for statement equivalent code. Instead, a deep understanding of the C code’s goal and structure is often needed in order to identify the core logic from an implicit and complex implementation. Then, using knowledge of how Rust would express such logic, the code is restructured and translated. A huge reason why C programs are difficult to translate is because C allows direct memory access at the most fundamental level. Therefore, it can be hard to know what parts of a program have access to what parts of memory, making it challenging to prove safety claims about the values in memory. Rust on the other hand has stricter ownership guidelines on what pieces of code can modify memory. So doing this translation from monolithic, unbounded C code requires carefully analysis before it can be reorganized into safe Rust.

1.3 Preparatory Refactoring

The research of this thesis focused on the first step of Tenjin’s pipeline, preparatory refactoring. The goal of this stage is to statically analyze the C code and map specific unsafe C memory patterns into safer, higher level code that is more ready to be directly converted into Rust. It is easiest to implement these static analyses when they can be broken down into separate passes where each one targets a specific C pattern. For example, the practice of using output parameters (passing to-be-written pointer parameters instead of explicitly returning values) can be directly targeted in a particular pass by looking at function signatures and discerning whether any of them are pointers that are only written to. Preparatory refactoring enables this modularity of code analysis as well as interoperability because as long as each sweep takes in C source code and outputs C source code, it doesn’t matter what the sweep is doing.

¹<https://www.aarno-labs.com/>

²<https://www.darpa.mil/research/programs/translating-all-c-to-rust>

Freedom to insert, rearrange, and remove independent passes adds flexibility to work outside the large core C to Rust translator as well.

The goal of this research was to build two preparatory refactoring passes focusing on union punning, when data types are implicitly converted using union fields, and pointers used as arrays.

1.4 Union Punning Pass

Union punning in C is when data is cast from one type to another using a union. The union contains two fields, the source data type and the destination data type, and punning code writes data to the source field and reinterprets the bits via a later read from the destination field. This reinterprets the bits into the destination type because both fields of a C union are stored in the same memory.

```
union {
    float f;
    uint32_t n;
} union_punner;
union_punner.f = 1.234; // write union with float value
uint32_t num = union_punner.n; // same memory reinterpreted as uint32_t
```

Listing 1.1: Union Punning Example

Union punning in C is unsafe because there is nothing inherently linking this write and read together. If the source field is accidentally not written to, the read will then read uninitialized data leading to undefined behavior. The goal of the union punning pass was to link these operations together such that the conversion happened in a single, isolated function call with a recognizable name. Then this function call can be later identified during the actual Rust translation to signal the use of Rust’s idiomatic type conversion using `to_bits()` and `from_bits(..)`. Below is the before and after of the union punning pass from a Tenjin test case. The refactored code links the union punning into a single `memcpy(..)` function call ready to be identified and translated to idiomatic Rust later on.

```
uint16_t float2half(float flt) {
    union {
        float flt;
        uint32_t num;
    } in;
    uint32_t n, j;
    in.flt = flt;
    n = in.num;
    j = (n >> 23) & 0x1ff;
    return (uint16_t)((uint32_t)m__base[j]
        + ((n & 0x007fffff) >> m__shift[j]));
}
```

Listing 1.2: Tenjin Union Snippet Before Pass

```
static void __tenjin_109_5_float_to_uint32_t(float x, uint32_t *out) {
    memcpy(out, &x, 4);
}
```

```

uint16_t float2half(float flt) {
    float __tenjin_tmp_in_in;
    uint32_t __tenjin_tmp_out_in;

    uint32_t n, j;
    __tenjin_tmp_in_in = flt;
    __tenjin_109_5_float_to_uint32_t(
        __tenjin_tmp_in_in, &__tenjin_tmp_out_in);
    n = __tenjin_tmp_out_in;
    j = (n >> 23) & 0x1ff;
    return (uint16_t)((uint32_t)m__base[j]
        + ((n & 0x007fffff) >> m__shift[j]));
}

```

Listing 1.3: Tenjin Union Snippet After Pass

Before performing the transformation, though, there is a set of criteria that source code must meet in order to become a valid union punning candidate. A simple restriction is that the union has two fields that are the same bit-width. Examples of more complex requirements are that if the source field is an array, every index must be written to (Listing 1.4) and that all source writes must predominate all destination reads. There are also restrictions that are placed to rule out rare and hard to translate edge cases. For example, using the same union variable for multiple type puns or complex data flow patterns are not supported, though they can be addressed in future work. This allows the pass to be effective on a large number of cases, without needing to spend time adapting the refactor for every possible situation.

```

double test(uint16_t a) {
    union {
        double d;
        uint16_t n[4];
    } u;
    u.n[0] = a;
    u.n[1] = a*4;
    u.n[3] = a-9;
    return u.d; // u.n[2] never written to, reading uninitialized data!
}

```

Listing 1.4: Invalid Union Punning Candidate

1.5 Pointer Pass

The basic idea for refactoring pointers to be more Rust compatible is to translate C pointers used to iterate or manipulate arrays into index variables that directly reference the array they are linked to. Specifically, each pointer `p` is replaced by an index variable `p_index` and accesses to `p` like `*p` becomes `arr[p_index]` and `p+=n` becomes `p_index+=n`. Listings 1.5 and 1.6 show a basic such conversion.

```

static uint32_t get_bits(bs_t *bs, int n) {
    uint32_t next, cache = 0, s = bs->pos & 7;
    int shl = n + s;

```

```

const uint8_t *p = bs->buf + (bs->pos >> 3);
if ((bs->pos += n) > bs->limit)
    return 0;
next = *p++ & (255 >> s);
while ((shl -= 8) > 0) {
    cache |= next << shl;
    next = *p++;
}
return cache | (next >> -shl);
}

```

Listing 1.5: Basic Pointer Example Before Pass

```

static uint32_t get_bits(bs_t *bs, int n) {
    uint32_t next, cache = 0, s = bs->pos & 7;
    int shl = n + s;
    int p_index = (bs->pos >> 3);
    if ((bs->pos += n) > bs->limit)
        return 0;
    next = (bs->buf)[p_index++] & (255 >> s);
    while ((shl -= 8) > 0) {
        cache |= next << shl;
        next = (bs->buf)[p_index++];
    }
    return cache | (next >> -shl);
}

```

Listing 1.6: Basic Pointer Example After Pass

This can be upgraded, however, to go beyond intrafunction refactoring through stronger static analysis. By looking at these pointers across the call graph instead of just within one function, the tool can recognize patterns that map to richer Rust types, like slices, and use those types instead. Once one function is transformed this way, the change can propagate to its callers and callees so that related functions use consistent types as well. This produces Rust output that is closer to what a Rust programmer would actually write, with type information flowing across function boundaries rather than stopping at them. Through multifaceted analysis, these transformations bring C pointer code closer to safe, idiomatic Rust.

Chapter 2

Related Work

2.1 C2Rust

One of the most well-known automatic C to Rust translation tools is called C2Rust¹. This tool applies a very direct translation that is focused on the simple line by line syntactic differences between C and Rust. More complicated C code that doesn't have a direct equivalent in Rust, though, gets translated into `unsafe`² Rust. Using the `unsafe` keyword in Rust basically tells the compiler to ignore all the strict safety requirements the language is known for and allow the use of unsafe C practices like dereferencing raw pointers. In this way C2Rust can correctly generate valid Rust code, but it often lacks the safety that was one of the main motivators of its creation in the first place.

One of the ways that C2Rust is being improved is to use multiple Rust to Rust passes post translation that target certain unidiomatic patterns like scalar pointers, locks, union with tags, and output parameters [3]. Tenjin aims to address these C patterns as well, but at the beginning of the pipeline in preparatory refactoring. This allows every later tool or pass in the pipeline to take advantage of the simplified code and understand the logic more deeply. This strengthens their ability to generate Rust code as knowledge of the C code's behavior is built earlier on and can be leveraged more effectively to generate cleaner Rust.

2.2 Pointer Translation

Recently, another tool CROWN (an acronym for C to Rust OWNership) has been developed by researchers at Bristol University to also improve Rust code post C2Rust translation [4]. Its focus is upgrading the typically unsafe Rust code generated from C pointers into code with the idiomatic ownership style encapsulation that Rust provides. To do its translation, CROWN identifies all raw pointers, tries to retype them into Rust `Box` pointers if they match the ownership pattern, and lastly rewrites all these pointer accesses to use better Rust syntax like `as_deref_mut()` and `unwrap()`. This is done by developing an ownership constraint system based on all pointer accesses in a program and solving it with a SAT solver.

¹<https://github.com/immunant/c2rust>

²<https://doc.rust-lang.org/book/ch20-01-unsafe-rust.html>

An additional paper targets pointers too, but this time in terms of lifetime analysis [1]. It performs a static analysis that helps determine which pointers can be safely converted to references, resulting in an increase of safe pointer conversions from 12% to 21%, a 75% increase.

Though these tools are powerful, they are limited by their rewriting scope. In CROWN and in the lifetime analysis paper, only statement level changes are made as opposed to larger loop or function level restructuring. These small changes aren't enough when a C program's memory usage is fundamentally incompatible with Rust. In an example where arrays or array values are passed as pointer arguments into functions, neither of these methods is aggressive enough to replace this with Rust objects or slices. Further, moving the refactoring to C enables more powerful translation to Rust. The pointer pass aimed to fill this gap by addressing pointer logic across larger code structures as well as gaining the benefits of doing so before Rust translation.

Chapter 3

Union Punning Pass

3.1 Methods

The union punning pass is a tool that runs on a single file, searches for potential union punning behavior, evaluates code to ensure that a valid transformation can be performed safely and simply, and then generates new source code replacing raw memory operations. The tool operates through clang because clang’s CFG, domination, and AST visitor support allows for thorough static analysis, but is still close enough to the original source code to make rewriting the source code relatively simple. An alternative would have been LLVM which has more powerful analysis tools, but is too far away from the source code to allow for rewriting. In this section, the implementation of the pass and how it analyzes, transforms, or rejects code is described in a step by step manner.

3.1.1 Preprocessing and Parsing the File

The pass starts by taking in a file, preprocessing it, and then running the static analysis and code generation. Preprocessing first is important because many real world union punning examples come from within macros. If the pass operated on unpreprocessed code, clang would still expand the macros while it generates its internal AST from the code, but when trying to rewrite new source code, this expanded code wouldn’t actually exist in the source file for the clang Rewriter. Therefore there would be a mismatch and the tool would have to rewrite the raw text based macro. That is challenging because macros have stringification, concatenation, or even nested macros within. Further, the macro can be used in many files so changes to one macro can affect other call sites in unpredictable ways. For safety, simplicity, and to keep each transformation isolated and independent, the code is preprocessed beforehand. Though this changes the pipeline to be from C-source-to-C-source to C-source-to-preprocessed-C-source, there is a tool in Tenjin that can take modified preprocessed code and convert it back to valid C so this change in format isn’t an issue.

For further simplicity and to avoid cross-function analysis, the pass searches and transforms union punning at a per-function basis. When clang parses a file, an `ASTConsumer` looks out for function declarations and each function is separately transformed.

3.1.2 Union Detection and Access Collection

Union detection is implemented via visitor methods in the `UnionAccessCollector` class, a clang AST visitor. These methods populate the `accesses` map which maps `TypePunCandidates` to a list of `FieldAccesses`. A `TypePunCandidate` specifies a particular union, its declaration, its two fields, and the field bit-width. A `FieldAccess` is an access to a field of a union like `u.x` or `u.x.a[i]`. It records which field is accessed (including an array index or struct field if the union field itself is an array or struct), whether it is a read or write, the source location (for later rewriting), and the AST `MemberExpr` itself. It is important to gather index and field information for array and struct union fields to ensure that no uninitialized data is read. The pass checks that every bit of the source field (all indices of arrays and all fields of structs) is written before data type transformation. This is discussed more in the next section on validating the union accesses once they are all collected.

The two visitor methods implemented are `VisitMemberExpr` and `VisitVarDecl` which gather `FieldAccesses` and assign them to the corresponding `TypePunCandidate`. `VisitMemberExpr` first finds the owner variable, the union itself, through a pointer or a direct access. Then it validates the union with `isValidUnion(..)`. This method ensures that the union follows the punning structure by making sure there are exactly 2 fields of the same bit-width and that the union variable is local. After the union is validated, the access is checked to see if it is a read or write and whether the access writes to a particular index of a union field array or field of a union field struct. Finally the access is recorded in the `accesses` map.

`VisitMemberExpr` covers almost all union variable reads and writes, but misses the case of initialization writes. The second visitor method, `VisitVarDecl`, addresses this by similarly validating the union and gathering all the necessary `FieldAccess` information. By working on `VarDecls` instead of `MemberExprs`, though, it can capture initialization writes through the union's declaration statement.

3.1.3 Access Pattern Validation

In order to determine if it is safe to transform a union access pattern into a `memcpy(..)` call, the union usage pattern needs to reflect the exact behavior of union punning. An example restriction is that a source field should not be written after the destination field has been read because that would require multiple transformations from one type to another and then back again. This section outlines thoroughly the requirements that the access pattern must follow and also identifies where in the CFG the `memcpy(..)` call should take place.

The following validation checks assumes one union field has been identified as the source and the other as the destination. In practice, the pass checks validity for both directions of one field to another and either picks the direction that works or rejects the union if neither works.

The primary requirement is that all source field accesses dominate all destination field accesses. Right at the switch from source to destination is called the `cutPoint` where the `memcpy(..)` must occur. The analysis here identifies the basic block where the `cutPoint` lives, later on in `analyzeCutPoint`, the exact location is determined.

For the basic block level analysis, the tool uses the CFG to collect a set of source blocks and destination blocks that contain accesses to the source and destination fields, respectively.

Then domination analysis is used to find the basic block that post-dominates all source accesses or predominates all destination accesses. The usual case is that the `cutPoint` exists at the beginning or end of the identified block. Sometimes, though, there is a block that has both source and destination accesses but the access pattern is still valid. This mixed access block must contain the `cutPoint`, therefore, and to ensure valid instruction level domination of this block, a simple sequential check is made on source and destination field accesses within this block.

Note that given the nature of these access pattern requirements, it is tricky for code with accesses in a loop to still pass the `cutPoint` check. It is still possible though and a valid example is shown in Listing 3.1.

There are some final miscellaneous safety checks to make sure there are no uninitialized data reads or untracked modifications happening to the data. For example, if the union is passed as a reference or a pointer to another function where its modifications can't be tracked, the union is rejected. Further, if a union source field is an array or a struct, the accesses are checked such that at the `cutPoint`, no matter what path through the CFG the code takes, all indices and all struct members are initialized. The reason why uninitialized data is important to leave unchanged is because even if the original code has some cases of uninitialized data reads and the code works as intended, by modifying the layout of the code through the transformation, the underlying uninitialized data may also change. Therefore, even if the transformation does not change the uninitialized data read pattern, it is still best to reject the union.

Below in Listings 3.1 and 3.2 are a few examples of different valid and invalid access patterns.

```
// Cut point with if and elses
uint32_t test(float flt) {
    union {
        float flt;
        uint32_t num;
    } in;
    if (flt>2) {
        in.flt = flt+2;
    } else {
        in.flt = flt*2;
    }
    // cut point here!
    if (flt>10) {
        return in.num -9;
    }
    return in.num*9;
}
```

```
// Cut point in loop
uint32_t test(float flt) {
    union {
        float flt;
        uint32_t num;
    } in;
    uint32_t sum = 0;
    for (int i=0; i<10; i++) {
```

```

        in.flt = flt + i;
        // cut point here!
        sum += sum * in.num;
    }
    return sum;
}

// It is okay to modify in.flt and read in.num multiple times as long as
// all src accesses still dominate all dst accesses
uint32_t test(float flt) {
    union {
        float flt;
        uint32_t num;
    } in;
    uint32_t n, j;
    in.flt = flt;
    in.flt = in.flt + 3;
    uint32_t a = in.num*3;
    uint32_t b = in.num+2;
    return a-b;
}

// It is okay to have the src field be written via initialization
typedef union {
    int i[2];
    double d;
} number;

double test(float flt) {
    const number d3 = {{0x55555555, (int)flt} };
    return d3.d;
}

```

Listing 3.1: Valid Access Patterns

```

// u.x.c is never written - uninitialized data hazard
struct s_t {
    char a;
    char b;
    char c;
};

int test(char c) {
    if (sizeof(struct s_t) > 1) {
        union {
            struct s_t x;
            char b[sizeof(struct s_t)];
        } u;
        u.x.a = 'a';
        u.x.b = 'b';
        return u.b[0]+u.b[1]+u.b[2];
    }
    return -1;
}

```

```

// No cut point exists
uint32_t test(float flt) {
    union {
        float flt;
        uint32_t num;
    } in;
    uint32_t sum = 0;
    in.num = 3;
    for (int i=0; i<10; i++) {
        in.flt = flt;
        sum += sum * in.num;
    }
    return sum;
}

```

Listing 3.2: Invalid Access Patterns

3.1.4 Source Code Rewriting

Now that the transformation has been validated, the actual generation of the new source code can occur. This happens in the method `emitTransformation(..)` that follows six steps to rewrite the code.

3.1.4.1 analyzeCutPoint

The first step is to find the exact location in the source code to insert the call to the `memcpy(..)` method. In `analyzeCutPoint(..)`, the `cutPoint` block is iterated through to see where, if any, are the source and destination access. If the `cutPoint` has a destination access, then insert the conversion function call before the first destination access. If the `cutPoint` has only source writes, then insert after the last source write. If there are neither source nor destination accesses, then the insertion just goes at the top of the block.

3.1.4.2 generateTemporaryVariables

The second step is to create the temporary variable declarations in `generateTemporaryVariables(..)`. For each union, two temporary variables, `__tenjin_tmp_in_<unionVarName>` and `__tenjin_tmp_out_<unionVarName>`, are created to hold the source and destination field values, respectively. These variables will replace all accesses to the union fields. When the types of these variables are primitives, their declarations can just be inserted where the union declaration takes place. In the case where the definition contains a struct, though, then `typedefs` are created as well. The generated name of the previously unnamed type simply uses a sequence of numbers based on the corresponding AST `RecordDecl` pointer to ensure uniqueness.

```

// Before Pass
union {
    uint64_t __11;
    struct {

```

```

        uint32_t __l, __h;
    } __i;
} __x;

// After Pass
typedef struct {
    uint32_t __l;
    uint32_t __h;
} tenjin_struct_94734661729992;
...
uint64_t __tenjin_tmp_in___x;
tenjin_struct_94734661729992 __tenjin_tmp_out___x;

```

Listing 3.3: Typedef Generation

Lastly, there is an attempt to delete the union declaration such that there is no trace left in the generated code of the now unused union. In most cases it is deleted, but in cases where the union type is named with an identifier or multiple unions are declared at the same time it is kept to prevent breaking other code. If deletion is possible, code is not deleted right away, but pushed to a list of pending edits that are all executed at the very end (more on this in `applyEdits(...)`). Similarly the generated declarations, as well as all future code changes, are pushed to this pending edits list as well.

3.1.4.3 `generateConversionFunctionCode` and `generateConversionCall`

The `memcpy(...)` code is wrapped in a designated conversion function that is used to isolate the memory operation and serve as an identifier for later conversion into Rust (can be replaced with Rust's `to_bits()` and `from_bits(...)`). The function is declared as a static void method that takes in the source value, a pointer to the destination temporary variable, and the number of bytes to copy. The name of the method is as follows `__tenjin_<line number of union def>_<col number of union def>_<source type>_to_<destination type>` for uniqueness.

There is some variation to how the temporary variables are passed as arguments, for example destination array variables are passed by value and degraded to pointers, while other as passed by pointer. The call to the generated conversion function is placed at the location found by `analyzeCutPoint(...)`.

```

static void __tenjin_1890_5_float_to_uint32_t(float x, uint32_t *out) {
    memcpy(out, &x, 4);
}
...
__tenjin_1890_5_float_to_uint32_t(
    __tenjin_tmp_in_in, &__tenjin_tmp_out_in);

```

Listing 3.4: Standard Conversion Function

3.1.4.4 `handleInitOnlyCase`

There is a special case that needs to be handled when the union is written via initialization. Here the tool extracts the initializer expression text from the union declaration and patches it into the temporary source variable declaration generated previously.

```

// Before Pass
union {
    float flt;
    uint32_t num;
} in = {flt};

// After Pass
float __tenjin_tmp_in_in = flt;
uint32_t __tenjin_tmp_out_in;

```

Listing 3.5: Initialization Special Case

3.1.4.5 collectReplacementEdits

In this step, all the `union.member` expressions are located and set to be replaced with their corresponding temporary variable references. These edits are collected and added to the pending edits list.

3.1.4.6 applyEdits

Only at the very end are any modifications made to the source code. This is because sequential rewrites that modify the same source location can interfere with each other and result in garbage code through accidental deletions or interleavings. Therefore, all pending edits are tracked in a list along with the corresponding source locations and edits are applied from the bottom of the file up. This way shifts made from edits at the beginning of the file don't affect later edits.

3.1.5 Transformation Examples

This section gives three full end to end examples of the pass's transformations. The original unpreprocessed C source and the final relevant parts of the generated preprocessed code are shown as the before and after. For all cases, the generated code follows a consistent structure:

1. A `memcpy(...)` forward declaration at the top of the file (inserted once).
2. An optional `typedef` for any unnamed struct or union field types.
3. A `static void` conversion function added before the transformed function, with a single `memcpy(...)` call. Its name encodes the variables types and source location for uniqueness and later identification in C2Rust pipeline.
4. Two temporary variable declarations (`__tenjin_tmp_in_<name>` and `__tenjin_tmp_out_<name>`) replacing the union declaration.
5. A call to the conversion function at the identified cut point.

Note that in the macro example, the whitespace may be slightly off, but this can easily be fixed with a pass of a favorite formatter.

```

// Before Pass
uint32_t test(float flt) {
    union {
        float flt;
        uint32_t num;
    } in;
    if (flt>2) {
        in.flt = flt+2;
    } else {
        in.flt = flt*2;
    }
    if (flt>10) {
        return in.num-9;
    } else {
        return in.num*10;
    }
}

// After Pass
void *memcpy(void *dest, const void *src, unsigned long n);
...
static void __tenjin_1890_5_float_to_uint32_t(float x, uint32_t *out) {
    memcpy(out, &x, 4);
}

uint32_t test(float flt) {
    float __tenjin_tmp_in_in;
    uint32_t __tenjin_tmp_out_in;

    if (flt>2) {
        __tenjin_tmp_in_in = flt+2;
    } else {
        __tenjin_tmp_in_in = flt*2;
    }
    __tenjin_1890_5_float_to_uint32_t(
        __tenjin_tmp_in_in, &__tenjin_tmp_out_in);
    if (flt>10) {
        return __tenjin_tmp_out_in-9;
    } else {
        return __tenjin_tmp_out_in*10;
    }
}

```

Listing 3.6: Control Flow Example

```

// Before Pass
typedef long T;

T test(char c) {
    T x = 300000;
    if (sizeof(T) > 1) {
        union {
            T x;
            char b[sizeof(T)];

```

```

    } u;
    u.b[0] = c;
    u.b[1] = c-3;
    u.b[2] = c+2;
    u.b[3] = c;
    u.b[4] = c+2;
    u.b[5] = c*2;
    u.b[6] = c+2;
    u.b[7] = c+1;
    x = u.x;
    return x+3;
}
return -1;
}

// After Pass
void *memcpy(void *dest, const void *src, unsigned long n);
...
typedef long T;

static void __tenjin_1893_9_char_to_T(char *x, T *out) {
    memcpy(out, x, 8);
}

T test(char c) {
    T x = 300000;
    if (sizeof(T) > 1) {
        char __tenjin_tmp_in_u[8];
        T __tenjin_tmp_out_u;

        __tenjin_tmp_in_u[0] = c;
        __tenjin_tmp_in_u[1] = c-3;
        __tenjin_tmp_in_u[2] = c+2;
        __tenjin_tmp_in_u[3] = c;
        __tenjin_tmp_in_u[4] = c+2;
        __tenjin_tmp_in_u[5] = c*2;
        __tenjin_tmp_in_u[6] = c+2;
        __tenjin_tmp_in_u[7] = c+1;
        __tenjin_1893_9_char_to_T(__tenjin_tmp_in_u, &__tenjin_tmp_out_u);
        x = __tenjin_tmp_out_u;
        return x+3;
    }
    return -1;
}

```

Listing 3.7: Array Example

```

// Before Pass
#define FLOAT_TO_BITS(x) ({ union { float f; uint32_t i; } u = { x }; u.i;
    })

uint32_t test(float flt) {
    return FLOAT_TO_BITS(flt);
}

```

```

// After Pass
void *memcpy(void *dest, const void *src, unsigned long n);
...
static void __tenjin_1891_15_float_to_uint32_t(float x, uint32_t *out) {
    memcpy(out, &x, 4);
}

uint32_t test(float flt) {
    return ({ float __tenjin_tmp_in_u = flt;
    uint32_t __tenjin_tmp_out_u;
    union { float f; uint32_t i; } u = { flt };
    __tenjin_1891_15_float_to_uint32_t(__tenjin_tmp_in_u, &
    __tenjin_tmp_out_u); __tenjin_tmp_out_u; });
}

```

Listing 3.8: Macro Example

```

// Before Pass
uint32_t test(float flt) {
    union {
        float flt;
        uint32_t num;
    } in;
    uint32_t sum = 0;
    uint32_t acc = 0;
    in.flt = 0.0f;
    int i = 0;
    do {
        in.flt = flt * (i + 1);
        sum += in.num;
        acc ^= in.num;
        i++;
    } while (i < 5);
    return sum + acc;
}

// After Pass
void *memcpy(void *dest, const void *src, unsigned long n);
...
static void __tenjin_1890_5_float_to_uint32_t(float x, uint32_t *out) {
    memcpy(out, &x, 4);
}

uint32_t test(float flt) {
    float __tenjin_tmp_in_in;
    uint32_t __tenjin_tmp_out_in;

    uint32_t sum = 0;
    uint32_t acc = 0;
    __tenjin_tmp_in_in = 0.0f;
    int i = 0;
    do {

```

```

    __tenjin_tmp_in_in = flt * (i + 1);
    __tenjin_1890_5_float_to_uint32_t(
        __tenjin_tmp_in_in, &__tenjin_tmp_out_in);
    sum += __tenjin_tmp_out_in;
    acc ^= __tenjin_tmp_out_in;
    i++;
} while (i < 5);
return sum + acc;
}

```

Listing 3.9: Loop Example

3.1.6 Rejected Code Examples

Below is a list of source code examples that would be rejected for transformation by the pass.

```

// Would require multiple cut points to transform
uint32_t test(float flt) {
    union {
        float flt;
        uint32_t num;
    } in;
    uint32_t sum = 0;
    in.flt = flt;
    for (int i = 0; i < 10; i++) {
        sum += in.num;
        in.flt = flt + i;
    }
    return sum;
}

// Would require multiple cut points to transform
float test(float flt) {
    union {
        float flt;
        uint32_t num;
    } in;
    uint32_t n, j;
    in.flt = flt;
    in.num = 2;
    float a = in.flt;
    return a + 10;
}

// Would require multiple cut points to transform
uint32_t test(float flt) {
    union {
        float flt;
        uint32_t num;
    } in;
    uint32_t n = 0;

    in.flt = flt;
    n += in.num;
}

```

```

    in.flt = flt*flt;
    n += in.num;

    return n;
}

// u.x.c never written
struct s_t {
    char a;
    char b;
    char c;
};

int test(char c) {
    if (sizeof(struct s_t) > 1) {
        union {
            struct s_t x;
            char b[sizeof(struct s_t)];
        } u;
        u.x.a = 'a';
        u.x.b = 'b';
        return u.b[0]+u.b[1]+u.b[2];
    }
    return -1;
}

// External function mutates union
void mutator(void* ptr) {
    uint32_t* num_ptr = (uint32_t*)ptr;
    *num_ptr = 0xDEADBEEF;
}

uint32_t test(float flt) {
    union {
        float flt;
        uint32_t num;
    } in;
    uint32_t res = 4;

    in.flt = flt;
    mutator(&in);
    res = in.num;

    return res;
}

```

Listing 3.10: Rejected Code Examples

3.2 Results

There are 3 test cases involving unions from the TRACTOR test suite. The pass was run on each of them and the transformed code showed correct behavior on a set of test inputs as well. This small set, though, is not very robust. To increase the number of test cases available, three open source repositories — `eastdc`, `glibc`, and `musl` — were also used to gather examples of union punning. The pass was run on each file in the repository and statistics on the number of accepted and rejected unions were collected. The generated files for each repository were also compiled to ensure that the transformation generated valid code. The total rate of identified union punning behavior being transformed is a reasonably acceptable 86% across all repositories.

	# identified	# replaced	# rejected
eastdc	44	44	0
musl	819	717	102
glibc	1036	872	164
Total	1899	1633	266

Table 3.1: Number of Identified, Replaced, and Rejected Unions By Repository

Of the unions that were identified but not transformed, 82% were rejected because they had an unsupported, complicated control flow pattern. 15% came from cases where there was uninitialized data in the source field prior to transformation which the tool rejects for safety reasons as explained in the Access Pattern Validation section. The remaining 3% came from situations where the union was passed as a pointer or a reference to a setup method that mutated the source field. These were rejected due to their cross-function behavior.

To test correctness it would have been ideal to dynamically test correct behavior of the transformed files beyond just compilation, but given that the unions are deep within large repositories, that is challenging. Instead, the transformed code pieces were spot checked for correctness statically (by hand). The unions that were rejected were categorized on why they were rejected and checked to ensure it was because the union was a case the tool explicitly didn't support.

Lastly, a set of 50 regression tests were built that isolated the logic found in real world examples and actually ran with test inputs. Though these of course do not replace the value of testing the exact transformed repository code, the extracted logic of the regression tests mirrored an array of patterns found in the repositories. In fact, many of the regression tests were created when generated code failed to compile to ensure the new fix is correct and that later changes do not break the test case.

The pass has proven successful and robust against the given TRACTOR test suite and a large collection of real world test cases.

3.3 Future Work

Future work would focus on supporting more advanced control flow patterns, since this is the largest source of rejections. The current tool requires that all source field accesses dominate all destination field accesses and that a single cut point can be identified at a basic block boundary or via a simple sequential check within a mixed-access block. This rules out valid punning patterns where the access flow is more complex, such as loops where source and destination accesses are interleaved across iterations in ways that still preserve punning semantics. Extending the analysis to handle these cases would likely require a more general treatment of the cut point, possibly allowing multiple cut points along different paths through the CFG, each with its own conversion call, rather than collapsing the transformation to a single insertion site.

Another effective extension would be cross-function analysis. Right now the tool operates strictly per function and rejects any union that can be mutated via a pointer or reference passed to another function, since modifications made elsewhere aren't currently tracked. This is a safe default but excludes a lot of real-world punning code, especially in larger codebases where helper functions read or write union fields on behalf of a caller. Supporting this would require tracking how the union flows across function boundaries and verifying that the access pattern requirements still hold across the combined call graph.

Chapter 4

Pointer Pass

The pointer pass shares much of its structure with the union punning pass. For example, it is built on clang as a source-to-source rewriter, uses AST visitors to collect references of candidate variables, and applies all source edits at the end from the bottom of the file upward to avoid interference between rewrites. Rather than repeat these shared details, this chapter focuses on what differs: how candidate pointers are identified and classified, how their usage is validated, and how each kind of access is rewritten. Further, the transformations themselves are shown as before-and-after tables instead of detailed walkthroughs of the implementation, since the pointer pass covers a much wider range of cases and the rules are easier to read side by side.

4.1 Methods

The pointer pass converts pointer operations into explicit array indexing when array-like mutation and usage is detected. The transformation typically involves removing the original pointer and replacing it with a corresponding index variable, but can also make a more aggressive rewriting to generate code that mirrors Rust slices. This is done by combining the pointer and length variable into a struct similarly to how a Rust slice has the length of the slice concretely tied to the data.

4.1.1 Pointer Collection and Access Classification

Within each function the tool identifies pointer candidates for the array indexing transformation and collects information to classify their behavior. The pass transforms all of a pointer's references into operations containing the base array the pointer is linked too and/or an index variable that represents the pointer's current position in the array. Therefore, in order to do this transformation, the pass needs to first identify this base array and then classify every one of the pointer's reference to know how to correctly transform it.

4.1.1.1 Base Array Identification

Base array identification usually happens through looking at a pointer's initialization or its first assignment. The following initialization types are understood by the tool:

1. *Direct array reference*: `int *p = buf`
2. *Array plus offset*: `int *p = buf + 5`
3. *Address-of subscript*: `int *p = &buf[i]`
4. *Null initialization*: `int *p = NULL`

If the pointer remains uninitialized or if it is a function parameter then the base array may be set later on when the tool evaluates cross-function pointers.

4.1.1.2 Access Classification

Once the base array has been set, it is important that the pointer is used only in an array-like manner. This means that it is only used to be read, written, or incremented/decremented, and if it is assigned a new value it must be an element of the same base array. Every type of allowed pointer operation is therefore classified and attributes of the operation like how much it is incremented by are tracked as well. The following access types are accepted:

1. Initialization:

- (a) `InitNull` — initialized to `NULL` or `0`
- (b) `InitArray` — initialized to a direct array/pointer reference (`int *p = buf`)
- (c) `InitArrayOffset` — initialized to an array plus offset (`int *p = buf + 5` or `int *p = &buf[i]`)

2. Assignment:

- (a) `AssignNull` — `p = NULL`
- (b) `AssignArray` — `p = buf` (must match existing base array)
- (c) `AssignAddrOf` — `p = &buf[i]`
- (d) `AssignArrayOffset` — `p = buf + offset`
- (e) `AssignFromAllowedFunc` — `p = strchr(p, c)` (assigned from an list of allowed library functions that returns a pointer)

3. Pointer Arithmetic:

- (a) `Increment` — standalone `p++` or `++p`
- (b) `Decrement` — standalone `p--` or `--p`
- (c) `PlusAssign` — `p += n`
- (d) `MinusAssign` — `p -= n`

4. Dereference Read:

- (a) `Deref` — `*p`

- (b) `DerefPostInc` — `*p++`
- (c) `DerefPreInc` — `*++p`
- (d) `DerefPostDec` — `*p--`
- (e) `DerefPreDec` — `*--p`
- (f) `DerefOffset` — `*(p + expr)` or `*(p - expr)` (offset must be a constant)

5. Dereference Write:

- (a) `DerefWrite` — `*p = val`
- (b) `DerefOffsetWrite` — `*(p + expr) = val` (offset must be a constant)

6. Structured Access of Index or Member:

- (a) `ArrowAccess` — `p->field` (read)
- (b) `ArrowWrite` — `p->field = val` or `p->field++` (write)
- (c) `Subscript` — `p[i]` (read)
- (d) `SubscriptWrite` — `p[i] = val` (write)

7. Comparison:

- (a) `ComparisonNull` — `p == NULL` or `p != NULL`
- (b) `ComparisonExpr` — `p < buf + n`, `p >= buf`, `p < end` (comparison that can be resolved to an array index comparison)
- (c) `Comparison` — any pointer comparison that cannot be resolved to an index form (causes rejection)

8. Implicit boolean:

- (a) `BoolTrue` — `if (p)`, `p && ...`, `p || ...`, `while (p)`, `p ? a : b`
- (b) `BoolFalse` — `!p`

9. Function Interactions:

- (a) `PassedToAllowedFunc` — pointer passed as argument to an allowed list of functions like `strchr` or `sscanf`
- (b) `PassedToFunc` — pointer passed as argument to any other function
- (c) `ReturnPtr` — `return p`

10. Miscellaneous Rejected Usages:

- (a) `AddressOf` — `&p`
- (b) `Unknown` — any use of the pointer that doesn't match a recognized pattern

4.1.2 Validation

Now that a pointer's behavior has been fully classified it needs to pass certain safety checks to ensure the pointer matches the transformation's desired behavior and that the transformation can validly transform the code.

Beyond the fact that there must be a single consistent base array, which was verified through checking that new assignments match the original base, it is also important to ensure that the pointer's type matches the base array's type. If, for example, there is a `float *p = (float *)int_array` then transforming the pointer dereference writes into `int_array` index writes will result in different bits being written as types can represent the same value in different ways. Therefore, cases where the pointer and base array's type mismatch are rejected by the pass on safety concerns.

There can be further correctness issues if the candidate pointer has its address taken. Not only will the transformation possibly introduce bugs by replacing a pointer whose address matters, but any mutations through the new pointer pointer variable must also be tracked which adds complexity. Complexity can also come from bounds checking for an array. Static analysis on bounds checking can become arbitrary complex if increments and decrements happen in a loop that is variably bound, for example. For situations where a pointer has variable offsets like `*(p + var)`, the tool rejects the pointer on safety concerns. It is possible, though, for the pass to track constant offsets like `*(p - 1)`.

The last major safety check comes from making sure only pointers that exhibit array-like behavior are transformed. To ensure that the pointer is not just a singleton that is never mutated or dereferenced, the pointer conservatively looks for explicit array tendencies like at least one increment, decrement, or index assignment.

4.1.3 Within-Function Transformation

This is the core, basic transformation where pointers are replaced by index variables and all of their accesses are rewritten according to the following rules. Below, every type of access is categorized and transformed with a range of examples for clarity.

4.1.3.1 Declaration

The pointer declaration becomes an `int` declaration initialized to the appropriate offset or -1 for `NULL`:

Before	After
<code>int *p = buf;</code>	<code>int p_index = 0;</code>
<code>int *p = buf + 5;</code>	<code>int p_index = 5;</code>
<code>int *p = &buf[i];</code>	<code>int p_index = i;</code>
<code>int *p = NULL;</code>	<code>int p_index = -1;</code>

4.1.3.2 Dereference

The pointer dereference and name are replaced with an array subscript:

Before	After
<code>*p</code>	<code>→ buf[p_index]</code>
<code>*p++</code>	<code>→ buf[p_index++]</code>
<code>++p</code>	<code>→ buf[++p_index]</code>
<code>*(p + 2)</code>	<code>→ buf[p_index + 2]</code>
<code>*(p - 1)</code>	<code>→ buf[p_index - 1]</code>

4.1.3.3 Structured access

Arrow access becomes dot access through subscript:

Before	After
<code>p->field</code>	<code>→ buf[p_index].field</code>
<code>p[i]</code>	<code>→ buf[p_index + i]</code>

4.1.3.4 Mutations

Pointer arithmetic becomes integer arithmetic:

Before	After
<code>p++</code>	<code>→ p_index++</code>
<code>p--</code>	<code>→ p_index--</code>
<code>p += n</code>	<code>→ p_index += n</code>
<code>p -= n</code>	<code>→ p_index -= n</code>

4.1.3.5 Assignments

Pointer assignments become index assignments:

Before	After
<code>p = buf</code>	<code>→ p_index = 0</code>
<code>p = buf + off</code>	<code>→ p_index = off</code>
<code>p = &buf[i]</code>	<code>→ p_index = i</code>
<code>p = NULL</code>	<code>→ p_index = -1</code>

4.1.3.6 Comparisons

Pointer comparisons become index comparisons. NULL is represented as -1:

Before	After
<code>p == NULL</code>	<code>→ p_index == -1</code>
<code>p != NULL</code>	<code>→ p_index != -1</code>
<code>p < buf + n</code>	<code>→ p_index < n</code>
<code>p >= buf</code>	<code>→ p_index >= 0</code>

4.1.3.7 Boolean contexts

Implicit pointer-to-boolean conversions become explicit index checks:

Before	After
<code>if (p)</code>	<code>→ if (p_index != -1)</code>
<code>if (!p && expr)</code>	<code>→ if (p_index == -1 && expr)</code>

4.1.3.8 Passing to non-transformed functions

In functions that aren't transformed (for example C library functions like `sscanf(..)`), the pointer is reconstructed from the base and index:

Before	After
<code>func(p)</code>	<code>→ func(buf + p_index)</code>

4.1.4 Special RustSlice Transformation

Beyond the direct within function transformations, there are opportunities for more aggressive reorganization across function boundaries. In Rust, arrays are typically passed to other functions as slices or references, but in C these are often represented as 2 parameters, a pointer and a length or a start pointer and an end pointer. The pass, therefore, addresses this particular problem by consolidating the 2 parameters into a single `RustSlice` struct parameter to make it more compatible with idiomatic Rust.

4.1.4.1 Detection

The pass scans across all functions looking for 2 parameters and usages that match the pattern:

```
void process(int *buf, int n) {
    int *p = buf;
    while (p < buf + n) {
        *p++ = 0;
    }
}
```

4.1.4.2 Typedef emission

The tool generates a struct type definition:

```
typedef struct { int *ptr; size_t len; } RustSlice_int;
```

4.1.4.3 Signature rewrite

The base pointer and bound parameters are replaced with a single `RustSlice`:

Before	After
<code>void process(int *buf, int n)</code>	<code>void process(RustSlice_int arr)</code>

4.1.4.4 Body transformation

Inside the function, the base array becomes `arr.ptr` and bounds become `arr.len`:

Before	After
<pre>int *p = buf; while (p < buf + n) *p++ = 0;</pre>	<pre>int p_index = 0; while (p_index < arr.len) arr.ptr[p_index++] = 0;</pre>

4.1.4.5 Call site rewriting

Every call to the transformed function is rewritten. The tool handles three cases.

4.1.4.5.1 From non-transformed code A compound literal is constructed:

Before	After
<code>process(buf, n)</code>	<code>process((RustSlice_int){buf, n})</code>

4.1.4.5.2 From another transformed function (pass-through) The slice `arr` is forwarded directly:

Before	After
<code>process(buf, n)</code>	<code>process(arr)</code>

4.1.4.5.3 From another transformed function (slice) A new slice is constructed from a sub-range and the length is shortened as needed:

Before	After
<code>process(mid + 1, hi)</code>	<code>process((RustSlice_int){arr.ptr + mid_index + 1, arr.len - mid_index - 1})</code>

4.1.4.6 Pointer pairs and inclusive end

For functions that take two pointers defining a range (e.g., `partition(int *lo, int *hi)`), the pass detects whether the end pointer is inclusive (dereferenced, or compared with `>` rather than `>=`). The length calculation is adjusted accordingly: (`hi - lo + 1` for inclusive, `end - lo` for exclusive).

4.1.4.7 Singleton callees

In some cases, a transformed function passes a pointer to a helper that does not exhibit explicit array behavior (like only dereferencing its pointer parameters without iterating). Even though the callee usually wouldn't be transformed, to maintain consistency, the callee is still modified to take the new `RustSlice` variable:

Before	After
<pre>void swap(int *a, int *b) { int tmp = *a; *a = *b; *b = tmp; }</pre>	<pre>void swap(RustSlice_int arr, int a, int b) { int tmp = arr.ptr[a]; arr.ptr[a] = arr.ptr[b]; arr.ptr[b] = tmp; }</pre>

4.1.4.8 Propagation To Other Functions in Caller/Callee Chain

RustSlice detection does not stop at individually analyzed functions, it propagates through the call graph. After the initial detection pass identifies root RustSlice functions (those with local iterating pointers), the pass examines their callers to find functions that should also be transformed. This detection runs as an iterative fixpoint loop: each iteration scans all unprocessed functions for calls to already-detected RustSlice functions where the caller's own parameters are passed as the base/end arguments. Newly detected functions are added to the set, and the loop repeats. The process terminates when an iteration discovers no new functions. This allows the transformation to propagate through arbitrarily deep call chains.

4.1.5 Full Transformation Example On Quicksort

Below shows how the pass transforms a sample quicksort implementation. It illustrates not just the within function transformations, but shows the special RustSlice modification as well as how that change was propagated to other functions. For example, `swap(..)` previously only took two pointers and swapped the values they point to which usually would not be transformed because there is no base array. However, after recognizing that `partition(..)` which calls `swap(..)` passes values that are both pointers to elements of the same array, `swap(..)` is transformed to take in a RustSlice argument. Further, the pass can make nuanced changes to the code, such as the changes in the comparison at line 19 of the before code and line 20 of the after code. The two `lo`, `hi` pointers were changed to use the length attribute of the RustSlice after it was inferred that `lo` and `hi` represent the array start and end and their comparison is a length comparison.

```
void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}

int *partition(int *lo, int *hi) {
    int pivot = *hi;
    int *i = lo;
    for (int *j = lo; j < hi; j++) {
        if (*j < pivot) {
            swap(i, j);
            i++;
        }
    }
    swap(i, hi);
    return i;
}
```

```

}
void quick_sort(int *lo, int *hi) {
    if (lo < hi) {
        int *p = partition(lo, hi);
        quick_sort(lo, p - 1);
        quick_sort(p + 1, hi);
    }
}
void sort(int *arr, int n) {
    quick_sort(arr, arr + n - 1);
}

```

Listing 4.1: Quicksort Before Pass

```

typedef struct { int *ptr; size_t len; } RustSlice_int;
void swap(RustSlice_int arr, int a, int b) {
    int t = arr.ptr[a];
    arr.ptr[a] = arr.ptr[b];
    arr.ptr[b] = t;
}
int partition(RustSlice_int arr) {
    int pivot = arr.ptr[(arr.len - 1)];
    int i_index = 0;
    for (int j_index = 0; j_index < arr.len - 1; j_index++) {
        if (arr.ptr[j_index] < pivot) {
            swap(arr, i_index, j_index);
            i_index++;
        }
    }
    swap(arr, i_index, arr.len - 1);
    return i_index;
}
void quick_sort(RustSlice_int arr) {
    if (arr.len > 1) {
        int p = partition(arr);
        quick_sort((RustSlice_int){arr.ptr, p});
        quick_sort((RustSlice_int){arr.ptr + p + 1, arr.len - (p + 1)});
    }
}
void sort(int *arr, int n) {
    quick_sort((RustSlice_int){arr, n});
}

```

Listing 4.2: Quicksort After Pass

4.2 Results

Of 21 supplied TRACTOR tests that exhibited valid pointer array behavior, all 21 were correctly transformed. The transformations of these tests included, global pointers, generating wrappers for standard library methods, and RustSlices.

The pointer pass was also tested on 5 public repositories, transforming a total of 3672 pointers and rejecting only 1.4% of candidate pointers identified. Similar to the union punning

pass, files were recompiled post translation to ensure that valid code was produced.

	# identified	# replaced	# rejected
coreutils	561	557	4
zlibng	6	6	0
stb	108	105	3
sqlite	210	202	8
ffmpeg	2840	2802	38
Total	3725	3672	53

Table 4.1: Number of Identified, Replaced, and Rejected Pointers By Repository

Of the 53 that were rejected, 49 of the rejected pointers were due to the lack of explicit array behavior, 2 were from the pointer’s address being taken, and 2 were from pointer and array type mismatch. The most common example of a pointer being related to an array but not exhibiting explicit array iterating behavior is when a pointer is set as the address after the last element of an array and then only used in comparisons. The tool rejects this because there is no mutation/valid index to represent this value, even though the pointer is inherently related to the array.

```
// se is rejected!
char *d = c->buf;
const char* s = buf;
const char* se = buf + len;
do {
    *d++ = *s++;
} while (s < se);
```

Listing 4.3: Rejected Sentinel Pointer

In terms of testing correct behavior of generated code, 59 regression tests were used to ensure that the 19 invalid examples were rejected and the 40 valid ones produced the same output before and after transformation. Invalid examples included variable lookaheads, type mismatch between pointer and array, and cases where the pointer candidate’s address was taken.

4.3 Future Work

One possible extension of the pointer pass would be to include handling the sentinel pointer pattern highlighted above. To handle these sentinels, the tool would need to detect pointers whose only uses are being derived from an array base and compared against another pointer into the same array. An idiomatic Rust solution would be to replace the sentinel with references to the array’s length.

Another direction could be improving the safety validation of a candidate pointer’s usage before transformation. There are many bugs that come from pointers being used beyond their intended buffers or ranges. Testing how the pointer actually changes over the course of

a dynamic run is very challenging to do statically. However, after transformation if the new Rust index variable goes out of bounds it would cause the program to panic and abort at runtime. Adding safety checks for simple cases could prevent proportion of these issues to ensure that only safe Rust is produced.

Chapter 5

Conclusion

This thesis presented two preparatory refactoring passes for the Tenjin C to Rust translation pipeline, each targeting a specific unsafe C memory pattern: union punning and pointer-based array manipulation. By operating on C source code before any Rust translation occurs, these passes simplify the work of later stages in the pipeline and enable the generation of cleaner, more idiomatic Rust output.

The union punning pass identifies valid type-punning patterns through static analysis on clang’s AST and CFG, validates that all source field accesses dominate all destination field accesses at a single cut point, and rewrites the code to isolate the bit reinterpretation into a recognizable `memcpy(..)` function call. This call serves as a signal for the later translation stage to emit Rust’s safe `to_bits()` and `from_bits(..)` conversions. The pointer pass similarly identifies pointers that exhibit array-like behavior, classifies every one of their accesses, and rewrites pointer operations as explicit index expressions tied to a base array. Beyond this within-function transformation, the pass also performs cross-function analysis to detect pointer-plus-length parameter pairs and consolidates them into a single `RustSlice` struct that propagates through the call graph, producing code that closely mirrors Rust’s slice abstraction.

Both passes were evaluated on the TRACTOR test suite as well as on large open source repositories. The union punning pass transformed 86% of identified candidates across `eastdc`, `musl`, and `glibc`, with the majority of rejections coming from complex control flow patterns that fall outside the current single-cut-point analysis. The pointer pass transformed over 98% of identified candidates across `coreutils`, `zlibng`, `stb`, `sqlite`, and `ffmpeg`, with rejections primarily from sentinel pointer patterns that lack explicit array iteration. Generated code compiled successfully in all cases, and regression test suites for both passes confirmed behavioral equivalence on extracted real-world patterns.

These results demonstrate the value of preparatory refactoring as a translation strategy. Rather than relying on Rust-to-Rust improvements after a direct line-by-line translation, restructuring the C code beforehand allows every downstream stage in the pipeline to take advantage of clearer logic and simpler memory patterns. The modular, pass-based design also makes it easy to extend the pipeline with additional refactorings as new unsafe C patterns are identified. With continued work on more advanced control flow handling and deeper cross-function analysis, this preparatory refactoring approach can cover an even wider range of legacy C code and bring automated C to Rust translation closer to producing the safe,

idiomatic Rust that motivated the effort in the first place.

References

- [1] Mehmet Emre et al. “Aliasing Limits on Translating C to Safe Rust”. *Proc. ACM Program. Lang.* **7**(OOPSLA1) Apr. 2023. DOI: [10.1145/3586046](https://doi.org/10.1145/3586046). URL: <https://doi.org/10.1145/3586046>.
- [2] MSRC Gavin Thomas Principal Security Engineering Manager. “A proactive approach to more secure code”. July 2019.
- [3] Jaemin Hong and Sukyoung Ryu. “Automatically Translating C to Rust”. *Communications of the ACM* **68**(11) Nov. 2025, pp. 56–65.
- [4] Hanliang Zhang et al. *Ownership guided C to Rust translation*. 2023. arXiv: [2303.10515](https://arxiv.org/abs/2303.10515) [cs.PL]. URL: <https://arxiv.org/abs/2303.10515>.