Analyzing and Lifting Legacy Software To Aid Rewriting (ALLSTAR)

Galois, Inc.
421 SW 6th Avenue, Suite 300
Portland, Oregon 97204

Immunant, Inc.
3333 Michelson Dr., Suite 300
Irvine, CA 92612

December, 13th 2024

FINAL TECHNICAL REPORT FOR PERIOD December 21, 2021 – December 17, 2024

Prime Contract Number HR0011-22-C-0020

Summary **3**
Task Objectives **3**
Technical Problems **3**
General Methodology **4**
Technical Results **4**
Static Analysis 4
Dynamic analysis 6
Code Rewriter 10
Code Amalgamation **10**
Transpiler Improvements 11
LLM experiments 12
Important Findings and Conclusions **13**
Significant hardware/software development **16**
Special comments **16**
Implications for further research **17**

Summary

The ALLSTAR project, a part of the DARPA LiLaC-SL program, builds on the existing C2Rust toolchain (https://c2rust.com). The baseline C2Rust tool, *c2rust-transpile*, converts almost any C application into structurally-equivalent Rust code. However, the resulting code is not safe, ie. statically guaranteed to be free of undefined behaviors by the Rust type-checker. In ALLSTAR, we have developed *c2rust-analyze*, a tool that uses a mix of techniques to improve the safety of code generated by *c2rust-transpile*.

This report describes the results of the ALLSTAR project. To summarize, we have developed three linked capabilities: (1) static type inference; (2) dynamic runtime analysis; and (3) code rewriting support for Rust. We have integrated these capabilities into the *c2rust-analyze* prototype and tested them on the *lighttpd* benchmark, as well as other benchmarks selected by DARPA. We have also made improvements to the *c2rust-transpile* partially in response to issues and requests filed by the community. Finally, we have evaluated LLM-based AI tools as a potentially complementary technology for use in transpilation tasks.

Task Objectives

The ALLSTAR project is structured into four tasks:

1. Development of a static type inference component that infers safe Rust types for the pointers in an unsafe Rust program. This static process takes hints from the dynamic analysis (see 2).
2. Development of a dynamic analysis that instruments unsafe Rust programs so as to collect information about how pointers flow through the program.
3. Development of a source code rewriting system that takes the result of type inference component (see 1) and uses it to transform unsafe raw pointers into safe Rust types.
4. Evaluation of LLMs as a component of the C2Rust pipeline.

We explain the technical work on each of these tasks in the section Technical Results, below.

Technical Problems

Problem 1: implicit code properties. Converting C code to safe Rust is a difficult task. It requires that many properties that are implicit in the C code be determined, then encoded explicitly in the Rust type system. For example, a function that takes a pointer argument might access the pointed-to allocation only temporarily, or it might take ownership of the allocation and eventually deallocate it. In C, these functions would have the same signature, but in Rust, the distinction between borrowing and ownership is encoded in the argument type. Selecting an appropriate pointer type for the translation to safe Rust thus requires inferring these implicit properties through interprocedural program analysis.

Problem 2: gap between C and Rust idioms. Additionally, although the C and Rust languages operate at similar levels of abstraction, many code patterns that are idiomatic in C are unidiomatic and difficult to express in Rust due to Rust's extensive static checking. For example, C code may track reference counts or array lengths implicitly to avoid memory overhead; in Rust, these counters are used for essential run-time safety checks, and it may be impossible to omit them without using unsafe code.

Problem 3: lack of suitable tools for supporting transpilation to Rust. Currently, few tools are available for analyzing and manipulating Rust code. Obtaining a complete and precise representation of the program typically requires integrating with the internal APIs of the Rust compiler. Instrumenting code for dynamic analysis also requires compiler integration. The integration itself is difficult because many of the APIs involved are not designed for external use and frequently change without notice as part of ongoing development of the Rust compiler.

General Methodology

Our methodology on the ALLSTAR project was an iterative prototype development process that worked as follows:

- We designed and built prototype capabilities as part of the *c2rust-analyze* tool.
- We tested those capabilities on our test suite, including our ‘model organisms’, the *lighttpd*[1] and *cFS*[2] libraries. This typically revealed bugs or novel code patterns that the new prototype could not handle.
- We evaluated these outcomes, and used the resulting insights to drive the next iteration of design and prototyping.

Labor on the project was shared between Immuant and Galois, with Immunant leading the dynamic analysis and LLM experiments, and Galois leading the static analysis and rewriting support.

Technical Results

Static Analysis

The static analysis component of *c2rust-analyze* comprises four discrete analyses, which together provide the information required to rewrite pointers to safe reference types.

Analysis 1: interprocedural dataflow analysis. First, the interprocedural dataflow analysis computes a set of "permissions" for each pointer, which represent operations that may be performed on the pointer. These permissions constrain the choice of rewritten type for each pointer. For example, if pointer `p` is used to write to the pointee in a statement like `*p = x;`,

[1] https://git.lighttpd.net/lighttpd
[2] https://github.com/immunant/cfs-rust/pull/1

then `p` must have the `WRITE` permission, and it would be invalid to rewrite `p` to a non-writable pointer type like `&T`.

The permissions computed by the interprocedural dataflow analysis include:

- `READ`: The pointer can be used to read the pointee. This includes the ability to borrow the pointee immutably, as in `&(*p).field`.
- `WRITE`: The pointer can be used to write the pointee. This includes the ability to borrow the pointee mutably.
- `OFFSET_ADD` and `OFFSET_SUB`: The pointer can be used in pointer arithmetic to advance it the forward and backward directions respectively.
- `FREE`: The pointer can be passed to `free`. This implies that the pointer either is null or points to a heap allocation. The pointer must be rewritten into an owned, heap-allocated type such as `Box<T>`.
- `UNIQUE`: The pointer is used in a manner consistent with Rust's borrow checking rules, avoiding mutable aliasing. This allows the pointer to be rewritten into `&mut T`; non-`UNIQUE` writable pointers must instead be rewritten into `&Cell<T>`.
- `NON_NULL`: The pointer is never null. Pointers without this permission must be rewritten into a type wrapped in `Option` to handle the null case.

The `READ`, `WRITE`, `OFFSET_ADD`, `OFFSET_SUB`, and `FREE` permissions are all propagated backward. For example, if pointer `q` has the `FREE` permission, and there is an assignment `q = p`, then `p` must also have the `FREE` permission; the assignment will transfer ownership of the heap allocation. The `UNIQUE` and `NON_NULL` permissions are instead propagated forward: if `p` is nullable (lacks the `NON_NULL` permission), then the assignment `q = p` means that `q` must also be nullable.

Analysis 2: Polonius. Second, the Polonius borrow checker is used in conjunction with the dataflow analysis to determine which pointers can have the `UNIQUE` permission. When invoking Polonius, *c2rust-analyze* treats raw pointers as if they were references. If this results in an error, then some of the pointers involved are mutably aliased, and *c2rust-analyze* must remove the `UNIQUE` permission from those pointers. The dataflow analysis then propagates this change to other pointers, including pointers in other functions.

Analysis 3: pointee type analysis. Third, the pointee type analysis tries to determine the concrete type of data pointed to by each pointer, which may be different from the pointer's declared type. This is particularly necessary for C code that uses `void*` or `char*` pointers to erase the concrete type of data being manipulated. This analysis infers the concrete type based on the type used at dereferences. For example, given the statement `*(p as *mut i32) = 123;`, the analysis will recognize this as writing a value of type `i32` into `p` (regardless of `p`'s declared type) and record `i32` as a possible pointee type for `p`. Pointee types are propagated backward through pointer assignments: if `p` must point to an `i32` value to satisfy a later dereference, and there is an earlier assignment `p = q`, then `q` must also point to an `i32` value.

During rewriting, if a pointer p has exactly one inferred pointee type, then it will be rewritten into a pointer to that type, such as &mut [i32]. (A pointer may have more than one pointee type if the data it points to is interpreted in different ways at different dereference sites; this behavior is legal in C and Rust but is currently unsupported by our analysis.)

Some operations, notably memcpy, interact with pointee values without specifying a concrete type for the access. For example, memcpy(q, p, size) copies some number of values from *p to *q, but unlike the assignment statement *q = *p, the concrete type of value being copied is not known at the site of the call. We handle this through unification. At the site of the memcpy, the pointee type analysis introduces a new inference variable T0 and treats the memcpy as if it dereferenced both *p and *q at type T0. The type T0 is propagated through assignments as normal; however, once propagation is complete, the analysis attempts to unify each pointer's set of pointee types down to a single type, potentially resolving inference variables in the process. If p's set of pointee types contains both i32 and T0, then unifying these produces i32 and resolves T0 = i32; if T0 also appears in q's pointee type set, then this unification allows information to propagate from p to q, even if neither pointer is assigned to the other.

Analysis 4: last-use analysis. Fourth, the last use analysis determines which expressions represent the last use of the value in a variable, meaning that either the variable is unused afterward or its value is overwritten before a subsequent access. During rewriting, the last use of a value is given special treatment: the value can be moved or consumed instead of being borrowed. This is necessary to satisfy the borrow checker in certain cases. For example, given p: Option<&mut T>, the inner reference can be accessed either as p.as_deref_mut().unwrap(), which borrows p, or as p.unwrap(), which moves p; the two expressions have the same type &mut T, but the borrowing version produces a reference with a shorter lifetime, tied to the local variable p.

Dynamic analysis

Static analysis must be conservatively correct when assigning permissions to pointers. To get the ground truth of each application and improve the scalability of our rewrites, we extended the static analyzer with a run-time dynamic instrumentation and analysis step. We first instrument the target program with handlers that catch and record a series of events involving pointers:

AddrOfLocal(x)	Take the address of a local variable x
AddrOfSized(x)	Take the address of a sized pointee x
Alloc(ptr, size)	A new memory allocation that produces a pointer, e.g., malloc
BeginFuncBody	Record function entry
Free(ptr)	A free of a previously allocated pointer

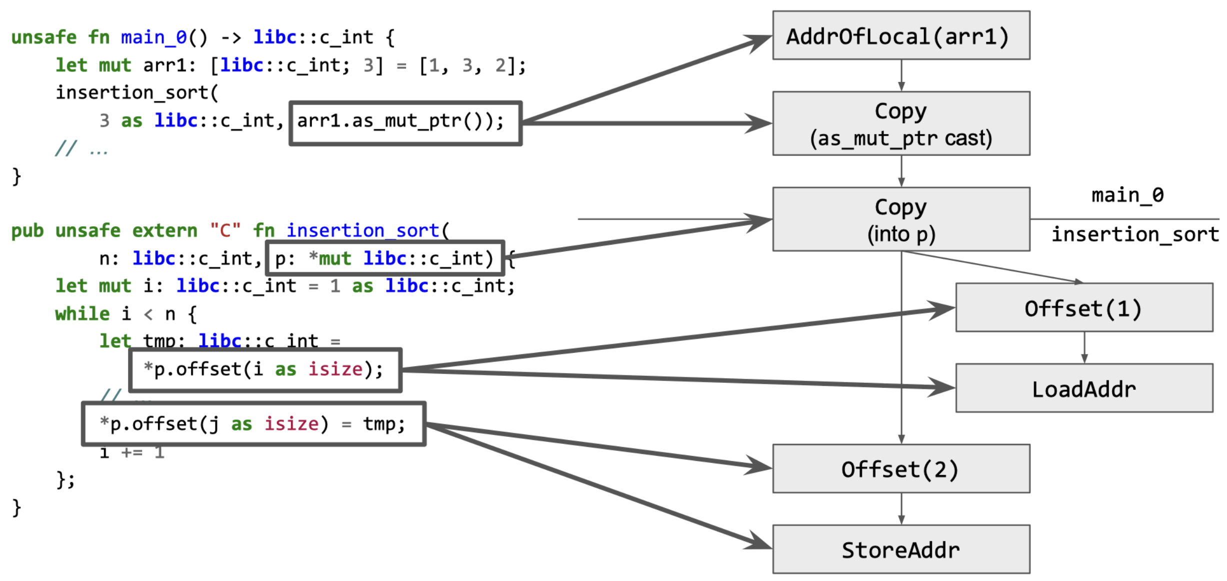
`FromInt(x)`	Create pointer out of integer `x`
`Copy(ptr)`	The copy of a pointer into another local, e.g., `q = p`
`Done`	Record end of program execution
`Offset(ptr, idx)`	Compute a new pointer at a given offset from a base address
`LoadAddr(ptr)`	Load the memory at the given address
`StoreAddr(ptr)`	Store at the given memory address
`StoreAddrTaken(ptr)`	Store at the memory address of `ptr`. Unlike `StoreAddr(ptr)`, this does not imply that `ptr` needs write permissions.
`LoadValue(ptr)`	Load a pointer value from memory
`StoreValue(ptr)`	Store a pointer value to memory
`Project(ptr, ptr, key)`	Compute new pointer via field projection. Key is used to disambiguate when different projections can produce the same pointer[3].
`Realloc(old, sz, new)`	Reallocate `old` pointer to `new` with size `sz`
`Ret(ptr)`	Record function return to `ptr`
`ToInt(ptr)`	Create integer out of pointer `ptr`

Each event encodes the relevant pointer by its memory address. The address therefore becomes a sort of temporary identity for each object, from the moment it is allocated to the corresponding `free` call. For now, we only perform this tracking for heap allocations; tracking stack variables is left for future work. Newly allocated objects with equal addresses but non-overlapping lifetimes are considered distinct. There are separate events for using a pointer as an address or as a value, e.g., `*q = p` is both a `StoreAddr` event for the `q` pointer and also a `StoreValue` event for the `p` pointer. This lets us track pointers across stores and loads from memory.

Running the instrumented program on a representative input (or several) produces a linear stream of pointer events in chronological order (even in the multi-threaded case, where our runtime serializes the events coming from multiple threads using a concurrent queue[4]). We then convert this stream into a Pointer Derivation Graph (PDG). The PDG is a collection of directed subgraphs, where each subgraph traces the lifetime of that pointer throughout the execution of

[3] https://github.com/immunant/c2rust/blob/master/analysis/runtime/src/events.rs#L36
[4] Introduced in https://github.com/immunant/c2rust/pull/1091

the program. Each node corresponds to an event from the stream, and each edge represents a “direct use” relationship at the MIR level. The figure below illustrates the PDG for a simple function.



```
unsafe fn main_0() -> libc::c_int {
    let mut arr1: [libc::c_int; 3] = [1, 3, 2];
    insertion_sort(
        3 as libc::c_int, arr1.as_mut_ptr());
    // ...
}

pub unsafe extern "C" fn insertion_sort(
        n: libc::c_int, p: *mut libc::c_int) {
    let mut i: libc::c_int = 1 as libc::c_int;
    while i < n {
        let tmp: libc::c_int =
            *p.offset(i as isize);
        // ...
        *p.offset(j as isize) = tmp;
        i += 1
    };
}
```

Both the static and dynamic analysis implementations operate on Rust MIR. The PDG preserves the MIR operation associated with each event and consequently each node. This lets us map all PDG subgraphs to their corresponding MIR locations and pointers in the MIR used by the static analyzer. Our dynamic analysis implementation builds the PDG and feeds it into the analyzer, with the goal of improving its results based on run-time observations. We extended the static analyzer to allow the PDG to override its findings by adding a “forbidden updates set” of permissions for each pointer. Whenever the data-flow algorithm would try to propagate a member of this set, the algorithm instead uses the override from the PDG.

We have also extended the dynamic analysis with support for `NON_NULL`. The implementation turned out fairly simple: since the PDG identifies each pointer by its address, all null pointers are consolidated into a single graph with pointer address `0x0`. For any given pointer in the static analysis, we only had to extend the dynamic analyzer to support one simple query: “is this pointer in the singleton `0x0` graph?” This is the opposite of the `NON_NULL` permission, so we negate the answer and pass it to the analyzer.

The additional information that the PDG currently provides to the analyzer consists of:

- An `is_null` flag for every graph marking whether the pointer for that graph is null. As stated earlier, this flag is negated and fed directly into the `NON_NULL` permission.
- An additional `NodeInfo` structure for every graph node with the following information:
    - A `unique` boolean flag that marks whether the current node can be used as a mutable reference. This flag is used to remove the `UNIQUE` permission from a static analysis pointer.

    - Optional “flows-to” edges for every node, used to add additional permissions to MIR pointers. A node `A` is said to “flow into” `B` if it is the transitive “source” of `B`, and `B` is one of the following operations (one separate edge per operation):
        - `load` for both `LoadAddr` and `LoadValue`; maps directly to the `READ` permission in the static analyzer
        - `store` for both `StoreAddr` and `StoreValue`; maps to `WRITE`
        - `pos_offset` for `Offset(x)` with positive `x`; maps to `OFFSET_ADD`
        - `neg_offset` for `Offset(x)` with negative `x`; maps to `OFFSET_SUB`

For debugging purposes, we have added an output mode that dumps the PDG in a linearized text format. The listing below shows the PDG for a small test function:

```
g is_null=false {
    n[0]: alloc        _    => _2  @ bb1[2]:  fn simple_analysis;  _2 = malloc(move _3);
    n[1]: copy         n[0] => _1  @ bb2[1]:  fn simple_analysis;  _1 = move _2 as *mut pointers::S (Misc);
    n[2]: project[0]@0 n[1] => _   @ bb2[5]:  fn simple_analysis;  ((*_1).0: i32) = const 10_i32;
    n[3]: addr.store   n[1] => _   @ bb2[5]:  fn simple_analysis;  ((*_1).0: i32) = const 10_i32;
    n[4]: project[0]@0 n[1] => _   @ bb2[18]: fn simple_analysis;  _10 = ((*_1).0: i32);
    n[5]: addr.load    n[1] => _   @ bb2[18]: fn simple_analysis;  _10 = ((*_1).0: i32);
    n[6]: copy         n[1] => _13 @ bb3[7]:  fn simple_analysis;  _13 = _1;
    n[7]: copy         n[6] => _12 @ bb3[8]:  fn simple_analysis;  _12 = move _13 as *mut libc::c_void (Misc);
    n[8]: free         n[7] => _11 @ bb3[10]: fn simple_analysis;  _11 = free(move _12);
}
nodes_that_need_write = [3, 1, 0]
```

The last line starting with `nodes_that_need_write` shows another piece of information that the PDG provides.

We have to be mindful of soundness issues when combining results from a static and dynamic analysis. Just because the dynamic analysis never observed a pointer having a null address does not mean that a pointer can never be null. Forcing `NON_NULL` on such a pointer can cause undefined behavior if this assumption is ever false at run time. For this reason, we support two modes of operation:

- The first is a test mode where we remove the `NON_NULL` permission if it is present, but do not add it by default. This tests the correctness of our system since there is either a problem with the static analysis or a problem with the integration of dynamic and static results if a pointer marked `NON_NULL` is observed with a null address.
- To explicitly enable using the results of the dynamic analysis to add the `NON_NULL` permission, we added an extra `PDG_ALLOW_UNSOUND` parameter to the analyzer which needs to be manually set. The reason is that the rewriter does not always perform a checked conversion of a raw pointer into a Rust reference. For instance, unchecked and unsafe conversions happen on the FFI boundary. If we were not successful in analyzing or rewriting all functions to use a Rust reference, the rewriter inserts shims that also perform unchecked conversions. These are limitations with our prototype, not with the general approach, and could be lifted with additional engineering effort.

Code Rewriter

The rewriter allows us to modify the target Rust code based on the analysis phases. Based on the results of the static and dynamic analyses, the rewriter first computes a new type for each pointer and then inserts casts in places where the source and destination of an assignment (or pseudo-assignment, such as passing an argument to a function) now have different types.

The rewriter supports several kinds of rewrites depending on the type of the original C code:

- `&T` / `&mut T` for ordinary references originating from non-nullable C pointers.
- `Result<Box<T>, ()>` for owned heap allocations.
- Option<_> wrapping any of the above to produce nullable reference types.

The rewriter builds in special cases for a number of common functions. We use these to replace unsafe *libc* calls with equivalent safe Rust code. Supported functions include: `malloc` and related memory allocation functions such as `calloc` and `realloc`, `free`, `memcpy`, `memset`, and Rust's `offset` function (corresponding to C pointer arithmetic).

The rewriter operates on all the available code at once, with limited support for targeting it at particular portions of the codebase. We support manually-written filter lists provided by the user, which allow us to exclude code that should not be rewritten. However, this coarse-grained support for rewriting does not allow us to target different sets of rewrites at different functions, or sequence rewrites. We think these features may be useful in future versions of the tool.

Code Amalgamation

C and clang process distinct translation units with headers textually included in source files. As a consequence, the C2Rust transpiler inherits this behavior and processes each translation unit independently. This is different from Rust, which has modules whose items can be imported. As a result, *c2rust-transpile* generates duplicate definitions for `#included` items in each source file. This presents a problem during static analysis, as we cannot infer that these items are the same. In future, we plan to fix this directly in the transpiler, but while fixing this in the common case is easy, the general case is more difficult. There is no guarantee that different `#includes` of the same file will expand to the same output, as prior definitions can affect the output.

For example, `dav1d` compiles many source files twice with different defines each as a way to be generic over certain values. However, duplicated items presented a huge problem for static analysis, so we decided on an alternative solution for our model organisms (*lighttpd* and *cFS*). Essentially, we “amalgamated” all translation units into a single one and then transpiled this single source file. Thus, there would only be a single definition of every item because there was only a single translation unit, and no imports would be needed to link all the items to each other. However, this did require some manual fixes, as some code was unable to be `#included` in a single translation unit, such as static functions with the same name.

Transpiler Improvements

Our new *c2rust-analyze* tool depends on the existing *c2rust-transpile* tool. We made several improvements to *c2rust-transpile* in the ALLSTAR project. The major transpiler improvement was removing the transpiler's dependence on `rustc` compiler internals. These tied the transpiler to a specific unstable Rust compiler version and made upgrading very hard. Now the transpiler uses the `syn` crate instead and is able to build on a stable toolchain. (#374)

Most of the other transpiler improvements were small fixes and translation improvements, such as:

- (#374) Rust changed how inline assembly works. We updated our support for C inline assembly, which is now translated to Rust inline `asm!` instead of the prior `llvm_asm!`.
- (#347, #385) Correctly preserving and translating more attributes, like `packed` into `#[repr(packed)]`.
- (#415) Preserving labeled blocks from C, which keeps the C and Rust more similar.
- (#612) Support for the variadic macro `va_copy` did not work in all cases, like when passing a pointer to a `struct` field, so we made support more robust to handle these cases.
- (#859) Some C builtin functions (e.g. `strlen`, `strchr`) were not handled, so we fixed translation for all C builtin functions.
- (#880) Improved translation of `else if` chains. `else if` used to turn into nested `else { if … }` trees, so now we translate them to normal `else if`s like in C.
- (#898) Unary operator expressions with side effects (e.x. `-func()`) were removed, so we fixed things so the side effects are preserved.
- (#1076) Allowing any integral type in C initializer lists (e.x. `unsigned int a[] = {1, 2, 3};`), not just `char` and `int`.
- (#1037) Support for `c2rust transpile src_files_*.c` vs. requiring a `compile_commands.json`. This was a commonly requested feature which makes it easier to transpile a single C source file.
- (#1030) Support for `bool` to float casts by going through `u8`.
- (#1134) Support for `bool` to `void*` casts by going through `size_t`.
- (#1128, #1163) Rust 1.80 started panicking during sorting if the order was not total, which uncovered a bug in the sorting of top-level items. We fixed the non-transitive order, so now top-level items should be sorted correctly.
- (#1170) Emitting zeroed arrays as `[0; N]` instead of `[0, 0, …, 0]`, more closely matching how they were declared in C.
- (#1185) Support `enum` compound literals, such as `(enum E) { A }`.

The rest generally were updates to support new LLVM/Clang and OS versions and improve our test workflows. The net effect is that the transpiler can now handle a far greater number of C libraries and projects on most Unix-like host systems.

LLM experiments

During the ALLSTAR period of performance, Immunant was working on a separate effort to migrate a high-performance AV1 video decoder library called dav1d[5] from C to Rust. This effort used the existing c2rust transpiler to generate the initial Rust code of the port (called rav1d[6]). Although using c2rust provided a major productivity boost because we were able to test each subsequent change against all available test vectors, the transpiler output was also much less compact and readable than the input C code due to a variety of reasons including extensive use of the C preprocessor in dav1d as well as limitations in *c2rust-transpile* that would require substantial engineering effort to overcome.

Specifically, the existing transpiler can handle all possible control flows in C (except control flows that can span activation frames such as `setjmp` /`longjmp`) but it is not always possible to map C control flows onto corresponding Rust as the latter does not support C-style for-loops or gotos for example. In cases where the C control flow cannot be mapped cleanly to Rust control flow constructs, the transpiler emits a state machine where a variable with the prefix `current_block_` is set to a numerical value and later used in a `match` statement containing a sequence of basic blocks. This code is difficult for humans to read and far from the desired end result of idiomatic Rust code. At the same time, the code that translates C control flows into Rust equivalents is quite complex; updating it without introducing errors is quite difficult and time consuming.

We therefore wanted to test whether it would be possible to use a large language model to rewrite the code in a way that is more readable and idiomatic. We used Google's Gemini 1.5 Pro model due to its large context window. We focused on a single non-trivial function: `insert_tasks` in the `thread_task` module (C version, Rust version before cleanup, Rust version after cleanup). We put the C sources and headers for the `thread_task` translation unit in the model's context and prompted the model to rewrite the transpiled Rust version to get rid of the state machine.

The output[7] was substantially more readable and included original comments which accurately described the logic of the rewritten code. Moreover, the code compiled, passed all tests, and did not regress performance. Encouraged by this initial finding, we carefully reviewed the code and found that it ultimately was not functionally correct. We had previously cleaned up the code by hand and during the review of the manually generated code change also surfaced a translation error. In other words, the Gemini 1.5 LLM performed about as well as a human programmer on this particular translation task. We ultimately ended up keeping the human translation because it was closer to the structure of the original C code and thus easier to check the correctness of. The machine translation produced more readable and compact code which we might have kept instead had similarity with the C code not been a priority.

[5] https://www.videolan.org/projects/dav1d.html
[6] https://github.com/memorysafety/rav1d
[7] https://github.com/memorysafety/rav1d/commit/368d911e143d39b7618018e9f817b268817979e1

The Gemini experiment was performed in early May of 2024. We re-ran the experiment in early December when OpenAI released their `o1` reasoning model. Interestingly, we found that the `o1` model did the smallest possible rewrite that satisfied our request ("Please rewrite the following Rust function to remove the `current_block_34` variable and make the code more idiomatic.") and left comments describing the rewrite[8]. The code compiled and passed all tests. More importantly, the fact that the `o1` model took a smaller step and inserted explanatory comments made it easier for a human to review the output. We take this as evidence that LLM capabilities are improving in ways that will make AI-driven transpilation more viable in future.

Important Findings and Conclusions

We have applied *c2rust-analyze* to *lighttpd* and other benchmarks. At a high level, these results show that:

1. A small but significant number of C functions can be translated directly into safe Rust by our static / dynamic strategy. These functions are generally 'naturally safe' in Rust, meaning that they have a structurally equivalent Rust function. In this case, we need only change the types of the pointers, and the code becomes statically safe, meeting our goal.
2. Some C functions present difficulties that are shallow in nature, and could be resolved by extending the tool. For example, if a function's control-flow is too complex, our inference may not be able to determine the correct type. However, given a more powerful static or dynamic analysis or more rewriting strategies, such functions could be supported.
3. Many C functions contain idioms that have no direct structural equivalent in Rust. In order to migrate these functions to safe Rust, we would need to transform the structure of the code. This might involve local or even global transformations. These functions cannot be translated using the approach we have used on ALLSTAR and would need some additional strategy to deal with them.

As a result of this pattern, we would expect that further investment of resources into ALLSTAR would result in a moderate increase in the number of functions that are translated into completely safe Rust. This would correspond to covering category (2) above. However, we expect there would be a 'long tail' of patterns in category (2), and that we would not be able to address category (3) by the ALLSTAR strategy.

c2rust-analyze shows promising results on two core modules of *lighttpd*.

Module	**Functions converted**	**Lines of code**
algo_md5	6/6	350
buffer	27/57	1400

[8] https://github.com/memorysafety/rav1d/commit/dd01a111d392c1a592fdc988031d0e823c2bc159

The algo_md5 module implements the MD5 hash function. It provides functions to initialize a hashing context, provide data to the hash function, and obtain the final hash value. This module uses type-erased `void*` pointers, for which *c2rust-analyze* infers a concrete underlying type to use in rewriting, and it also uses the `memcpy` and `memset` library functions, which *c2rust-analyze* rewrites into safe operations on Rust slices. *c2rust-analyze* converts the entire module to safe Rust code with no manual edits required.

The buffer module implements a heap-allocated, resizable byte array, which is used throughout *lighttpd* to store strings (such as HTTP headers) and binary data. It performs memory management using the `malloc`/`free`/`realloc` library functions, which *c2rust-analyze* translates using Rust's standard `Box` and `Vec` types, and it allows some pointer arguments and fields to be null, so *c2rust-analyze* translates these types into `Option<T>`. A small number of manual edits are required before running *c2rust-analyze* to break up access patterns that are incompatible with Rust's borrow checker and to simplify error-handling code. After applying these edits, *c2rust-analyze* automatically converts 27 functions to safe Rust code; this includes the core implementation of the buffer type in functions such as `buffer_init`, `buffer_extend`, and `buffer_free`. Higher-level helper functions like `buffer_append_strftime` are not converted to safe code as they rely on unsupported features or library functions.

To evaluate the effectiveness of *c2rust-analyze* on the entire *lighttpd* codebase, we measured the results of "pointwise" rewriting of *lighttpd*, in which each function is rewritten in isolation while all other functions remain in their original unsafe forms. This reduces the influence of complex cross-function effects on the measurement, giving more consistent results as the capabilities of *c2rust-analyze* improve over time.

The results of pointwise testing are computed as follows. For each function f, we create two copies of the codebase. In the first copy, we run *c2rust-analyze* to rewrite only f (leaving other functions unchanged), then remove the unsafe keyword from f, and finally try to compile the result. If the compilation succeeds, then we say that f is safe after rewriting. In the second copy of the codebase, we only remove the `unsafe` keyword from f and attempt to compile it. If this compilation succeeds, then we say that f is trivially safe. We then count the number of functions that are trivially safe and/or safe after rewriting.

- Safe after rewriting: 136/1620 functions (8.4%)
- Trivially safe: 222/1620 functions (13.7%)
- Improved by rewriting: 33/1398 non-trivially-safe functions (2.4%). These are cases where *c2rust-analyze* converted an unsafe function into a safe one. Note that not all of the algo_md5 and buffer functions succeed here due to limitations on cross-function rewriting in this measurement setup.
- Regressed after rewriting: 119/222 trivially-safe functions (53.6%). These are functions that manipulate raw pointers (but don't dereference them, which would be unsafe) where *c2rust-analyze* failed to convert the raw pointers to safe reference types.

As well as evaluating our new tool in total, we also evaluated whether the addition of the dynamic analysis resulted in improvements in the success of transpilation. To do this, we ran the augmented static analysis (with PDG information) against a baseline analysis on one model organism (the *lighttpd* web server). The PDG showed that the `connection_handle_close_state(mut con: Option<&Cell<connection>>)` function is never called with a `None` value (which is a reasonable conclusion since only valid connections should be shut down). This lets us rewrite its parameter and those of its callees from an `Option<&Cell>` to a simple `&Cell`, eliminating a series of `is_none` and `unwrap` checks. The following table lists a small sample of the interesting changes caused by the additional dynamic information:

```
fn connection_handle_close_state(mut con: Option<&Cell<connection>>) {
// 8813: mut con:   g810 = READ | WRITE | FREE, CELL
fn connection_handle_close_state(mut con: &Cell<connection>) {
// 8813: mut con:   g810 = READ | WRITE | FREE | NON_NULL, CELL[9]
```

```
connection_close((con).unwrap().as_ptr());
// 8816: con:   l14 = READ | WRITE | FREE, CELL
connection_close((con).as_ptr());
// 8816: con:   l14 = READ | WRITE | FREE | NON_NULL, CELL
```

```
connection_state_machine_loop((r).unwrap().as_ptr(), (con).unwrap().as_ptr());
// 10201: r:   l34 = READ | WRITE | FREE, CELL
// 10201: con:   l36 = READ | WRITE | FREE, CELL
connection_state_machine_loop((r).as_ptr(), (con).as_ptr());
// 10201: r:   l34 = READ | WRITE | FREE | NON_NULL, CELL
// 10201: con:   l36 = READ | WRITE | FREE | NON_NULL, CELL
```

```
// 10189: r:   g813 = READ | WRITE | FREE, CELL
Option::Some(&*(((*(r).unwrap()).conf).get() as *const Cell<fdlog_st>)).unwrap().as_ptr(),
// 10189: r:   g813 = READ | WRITE | FREE | NON_NULL, CELL
&*(((*r).conf).get() as *const Cell<fdlog_st>).as_ptr(),
```

```
// 10215: con:   g815 = READ | WRITE | FREE, CELL
let r: Option<&Cell<request_st>> = Some(&mut *Some((&mut
(*(con).unwrap()).request)).unwrap());
// 10215: con:   g815 = READ | WRITE | FREE | NON_NULL, CELL
let r: &Cell<request_st> = &mut *(&mut (*con).request);
```

```
// 15435: c:   l6 = READ, CELL
offset = ((*(c).unwrap())).get();
toSend = ((*(c).unwrap()).file).get() - ((*(c).unwrap())).get();
// 15435: c:   l6 = READ | NON_NULL, CELL
offset = ((*c)).get();
toSend = ((*c).file).get() - ((*c)).get();
```

To test how well rewriting based on static and dynamic analysis scales, we applied it to the `buffer` module of lighttpd[10]. We picked this module because it is relatively simple (~1000 lines of C) yet part of a non-trivial application. In total, we were able to rewrite a little under half of all the functions (27 out of 57 total). This includes all "core" functions such as `buffer_init`,

[9] `CELL` here is a different kind of attribute: it is a *flag* which is computed by a separate algorithm from the dataflow analysis that handles the other *permissions*

[10] https://github.com/immunant/lighttpd-rust/blob/385ca548101242cfb8f8bc61533f1073042b5036/src/buffer.c

buffer_free, etc., but excludes helper functions such as buffer_append_strftime, buffer_urldecode_path, etc. Additionally, a small number of edits before or after rewriting were required to change lifetimes that would otherwise prevent our rewrites. All in all, we were able to reduce the number of Option types in the buffer module from 50 to just 4. The example below shows one function from the module before and after rewriting:

```
// buffer function without NON_NULL information
fn buffer_realloc<'h0: 'h1, 'h1>(
    b: Option<&'h0 mut buffer>,
    len: size_t,
) -> Option<&'h1 mut [i8]> {
    b.as_deref_mut().unwrap().ptr = Some(DynOwned::new(/* ... */));
    b.as_deref_mut().unwrap().size = /* ... */;
    // ...
}
// buffer function with extra NON_NULL information
fn buffer_realloc<'h0: 'h1, 'h1>(
    b: &'h0 mut buffer,    // was: Option<&mut buffer>
    len: size_t,
) -> &'h1 mut [i8] {       // was: Option<&mut [i8]>
    b.ptr = Some(DynOwned::new(/* ... */));
    b.size = /* ... */;
    // was: b.as_deref_mut().unwrap().size = ...
    // ...
}
```

Significant hardware/software development

Galois and Immunant's work on ALLSTAR has primarily taken the form of developing the c2rust-analyze tool and supporting infrastructure, and applying this tool to benchmarks. All code developed on the ALLSTAR project has been released as open source and is available on the C2Rust GitHub repository[11]. The C2Rust transpiler is available through a web interface at https://c2rust.com.

In order to increase the visibility of our work on ALLSTAR, Immunant PI Per Larsen wrote a paper *"Migrating C to Rust for Memory Safety"*[12], which was published in the IEEE journal *Security & Privacy* (Jul-Aug 2024, vol. 22).

Special comments

No special comments.

[11] https://github.com/immunant/c2rust
[12] Permanent link: https://doi.org/10.1109/MSEC.2024.3385357

Implications for further research

Our main conclusions from the ALLSTAR work are:

1. Static and dynamic based tools can be very effective in improving the safety of some classes of C code—roughly the code that is already idiomatically similar to Rust.
2. Many C idioms are difficult or impossible to transpile into safe code using a purely symbolic (i.e. rule-based static and dynamic) approach.
3. LLMs and other AI-based tools are a promising direction for building more capable transpilers that can cover more classes of C code. The current generation of models seem particularly promising (see our experiments with the o1 model above)

We make the following suggestions for further research:

- Many kinds of refactoring tasks can be accomplished by pure-symbolic methods. However, it is arduous to build and debug refactorings by hand, and the process suffers from diminishing returns as the number of refactorings grows. To solve this problem, we believe what is needed is some kind of *generic refactoring layer or language for Rust* which can be used as a target for developers.
- We found that it is difficult to understand whether a set of refactorings together are correct, even when all the refactorings are relatively simple. We therefore recommend *more work on formally verified program transformation*. The refactoring language we propose in the previous point could be a target for this kind of work, which would help developers build transpilers and other tools more quickly and confidently.
- As we have stated above, pure-symbolic techniques cannot accommodate the range of C code idioms (and, beyond the scope of ALLSTAR, cannot impose human idiomaticity on generated code). We recommend *more research on combining symbolic and AI-based methods*, specifically focusing on LLMs as a strong candidate for better transpilation.
- Our experiments showed a wide range of C code patterns, and it seems unlikely that any tool will cover 100% of these in the near future. One alternative is to combine transpilation to safe Rust with dynamic safety enforcement mechanisms, for example by garbage collection. We recommend more investigation into *hybrid static/dynamic safety enforcement mechanisms* for Rust.