2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)

Improving Automatic C-to-Rust Translation with Static Analysis

Jaemin Hong
School of Computing
KAIST
Daejeon, South Korea
jaemin.hong@kaist.ac.kr

***Abstract*—While popular in system programming, C has been infamous for its poor language-level safety mechanisms, leading to critical bugs and vulnerabilities. C programs can still have memory and thread bugs despite passing type checking. To resolve this long-standing problem, Rust has been recently developed with rich safety mechanisms, including its notable ownership type system. It prevents memory and thread bugs via type checking. By rewriting legacy C programs in Rust, their developers can discover unknown bugs and avoid adding new bugs. However, the adaptation of Rust in legacy programs is still limited due to the high cost of manual C-to-Rust translation. Rust's safe features are semantically different from C's unsafe features and require programmers to precisely understand the behavior of their programs for correct rewriting. Existing C-to-Rust translators do not relieve this burden because they syntactically translate C features into unsafe Rust features, leaving further refactoring for programmers. In this paper, we propose the problem of improving the state-of-the-art C-to-Rust translation by automatically replacing unsafe features with safe features. Specifically, we identify two important unsafe features to be replaced: lock API and output parameters. We show our results on lock API and discuss plans for output parameters.**

I. Introduction

C is a widely-used system programming language, but its poor language-level safety mechanisms have resulted in serious bugs and vulnerabilities in software systems. Approximately half of the reported open-source vulnerabilities in the past decade are in C programs [29]. Despite being statically typed, C's weak type system cannot eliminate the majority of memory and thread bugs. Even a program that passes type checking is not guaranteed to be memory- and thread-safe. This places a burden on C programmers to manually prevent bugs, which is time-consuming and error-prone.

Rust [3], [28] is a newer system programming language that addresses the limitations of C. Its ownership type system ensures both memory and thread safety [23]. Rust also offers high-level abstractions not available in C, such as closures [24, §13.1], generics [24, §10.1], and traits [24, §10.2]. These features make the code more concise and reduce the need for unsafe type casts, leading to fewer bugs.

The design of Rust, which prioritizes safety while still promising good performance, provides an opportunity for the system programming community to improve the safety of legacy system software. By rewriting C programs in Rust, people can find previously unknown bugs with the help of the Rust type checker. For example, over half of cURL's known security vulnerabilities would have been easily detected if its developers had rewritten it in Rust [22]. Furthermore, the risk of introducing new bugs while adding new features decreases once the software has been ported to Rust.

Noticing the benefits of Rust in enhancing software safety, programmers have been re-implementing critical system software in Rust. For example, Mozilla developed the Servo web browser in Rust to replace certain modules of Firefox with those of Servo. Rust's safety mechanisms enabled the correct implementation of complex HTML renderers [17]. Operating systems, the most complex category of system programs, are also adopting Rust. The latest release of the Linux kernel now supports Rust code [10]. Android and Fuchsia are using Rust in their implementations [2], [35].

Unfortunately, the high cost of manual C-to-Rust translation prevents wider adoption of Rust in system programs. The translation process is not only labor-intensive but also difficult to perform correctly due to the differences between C and Rust. Rust provides features for safety, which are different from their C counterparts both syntactically and semantically. For example, Rust has *references*, pointers guaranteeing safe access, but the use of references has stricter restrictions compared to pointers in C. These discrepancies require programmers to have a precise understanding of their program's behavior and the Rust features.

This poses a need for automatic C-to-Rust translation, but only limited work has been done in this area. C2Rust [40], the most widely used automatic C-to-Rust translator, performs only syntactic translation. It takes advantage of the fact that Rust provides *unsafe* features that coincide with C's features, in addition to the safety-guaranteeing features. For example, Rust has *raw pointers* whose access may not be safe, similar to C's pointers. C2Rust-generated programs utilize these unsafe features and therefore cannot ensure safety via type checking. Programmers are expected to manually replace the unsafe features with safe features after C2Rust's translation. Emre et al. [12] proposed an approach to replace raw pointers with references in C2Rust-generated programs, but other unsafe features still remain.

In this paper, we propose to improve the state-of-the-art automatic C-to-Rust translation by replacing the unsafe features in C2Rust-generated programs with safe features. Since

979-8-3503-2263-7/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE-Companion58688.2023.00074

the safe features are semantically different from the unsafe features, we need static analyses to compute the behavior of the unsafe features in the target program and replace them. Each unsafe feature has its own characteristics, requiring the design of a specialized static analysis. This work identifies two important unsafe features to be replaced: lock API and output parameters (§III). We show our results on replacing the C lock API with the Rust lock API (§IV) and discuss plans for output parameters and other important unsafe features. (§V).

II. Background and Related Work

C2Rust [40] is the most widely used C-to-Rust translator. Its translation is syntactic. It translates each feature used by C code into a possibly unsafe equivalent feature of Rust to resolve the syntactic discrepancies between C and Rust.

C2Rust maintains all the C types and calls to C library functions during the translation. The following is example C code and its translation by C2Rust:

```
void f(int x) { if (x==1) printf("Hello, world!"); }
```

```
fn f(mut x: libc::c_int) {
  if x == 1 as libc::c_int {
    printf(b"Hello, world!\0"
      as *const u8 as *const libc::c_char); }}
```

While the function definition and conditional statement are syntactically translated, the type `int` and the function `printf` remain. Rust programmers prefer the following code:

```
fn f(x: i32) { if x==1 { print!("Hello, world!"); }}
```

which uses Rust's primitive type `i32` (which varies depending on the hardware) and the `print!` function provided by the Rust standard library.

C2Rust also retains C pointers by translating them into raw pointers in Rust. Raw pointers are equivalent to C pointers and do not guarantee safe access. The following is an example of C code using a pointer and C2Rust's translation:

```
void f(int *p) { *p = 1; }
```

```
fn f(mut p: *mut libc::c_int) {
  *p = 1 as libc::c_int; }
```

where `*mut` represents raw pointers. To ensure memory safety, Rust provides a safe alternative to raw pointers: references. They have more restrictions but are always safe to access. Programmers prefer references, as in the following code:

```
fn f(p: &mut i32) { *p = 1; }
```

where `&mut` represents references.

CRustS [26] enhances the output of C2Rust through syntactic rewrites, which are defined by 220 syntactic replacement rules written in the TXL transformation language [9]. For instance, it can substitute `libc::c_int` with `i32`.

Emre et al. [12] proposed an approach to replace raw pointers in C2Rust-generated code with references. Blindly replacing all the raw pointers with references does not work for two reasons: some raw pointers cannot be replaced due to the restrictions on references; references require their lifetimes, i.e., how long they are valid, to be annotated when the compiler cannot infer them. They address these issues by utilizing the compiler's feedback. They start by replacing all the raw pointers with references and then modify the code, either by converting some references back to raw pointers or adding lifetime annotations, based on the compiler's error messages. This continues until the code successfully compiles.

Recently, language models capable of code translation have been proposed [5], [6], [13], [19], [25], [31], [32], [36], [38], [39]. While they can translate C code into Rust, their translation cannot ensure semantics preservation.

Apart from C-to-Rust translation, many safe substitutes for C and (semi-) automatic translation to those languages have been proposed [7], [8], [11], [14]–[16], [18], [27], [30], [33]. However, they guarantee only limited forms of memory safety, unlike Rust. For example, Cyclone [18] prevents dangling pointer dereference; Checked C [11] guarantees the absence of null pointer dereference and out-of-bound accesses.

III. Proposed Problem

We propose to improve the state-of-the-art automatic C-to-Rust translation by replacing unsafe features in C2Rust-generated programs with safe features. We discuss two important unsafe features: lock API and output parameters.

A. Lock API

Locks are widely used to prevent data races, which occur when multiple threads read and write to the same memory address simultaneously. To avoid data races, each thread acquires and releases a lock before and after accessing shared data. Unfortunately, if threads acquire wrong locks, acquire locks too late, or release locks too early, data races can still persist despite the use of locks.

The most popular lock API of C is pthreads [4], which provides the `pthread_mutex_lock` and `pthread_mutex_unlock` functions. They acquire and release the lock given as an argument, respectively. Threads have to call them before and after accessing shared data, as demonstrated in the following example:

```
int n = ...; pthread_mutex_t m = ...;
void inc() { pthread_mutex_lock(&m);
  n += 1; pthread_mutex_unlock(&m); }
```

where `n` is a shared integer and `m` is a lock protecting `n`. Each thread holds `m` when accessing `n`.

Programmers are responsible for properly using locks when working with pthreads, as the API does not validate the correctness of lock usage in programs. They can make two kinds of mistakes when using locks: a *data-lock mismatch*, i.e., an acquisition of an incorrect lock, and a *flow-lock mismatch*, i.e., an acquisition of a lock at a wrong time. Below is an example of a data-lock mismatch.

```
pthread_mutex_lock(&m2);
n1 += 1; pthread_mutex_unlock(&m2);
```

where the other parts of the program acquire `m1` when accessing `n1`. The above code incorrectly acquires `m2`, instead of `m1`. Below is an example of a flow-lock mismatch.

```
void f1() { n += 1; pthread_mutex_lock(&m); ... }
void f2() { ... pthread_mutex_unlock(&m); n += 1; }
```

where the program uses `m` to protect `n`. The function `f1` accesses `n` before acquiring `m`, and `f2` accesses `n` after releasing `m`, which are both wrong.

In contrast, the lock API of Rust ensures that the use of every lock is correct [23]. To prevent data-lock and flow-lock mismatches, the lock API makes *data-lock relations*, i.e., which lock protects which data, and *flow-lock relations*, i.e., which lock is held at which program point, explicit in programs. First, it couples a lock with shared data; this reveals data-lock relations. Each Rust lock can be considered as a C lock with shared data. The following code creates a lock initially containing `0` and names it `m`:

```
static m: Mutex<i32> = Mutex::new(0);
```

To access shared data, threads must acquire the lock coupled with the data, eliminating the possibility of data-lock mismatches. Second, it introduces the notion of a *guard* to expose flow-lock relations. The `lock` method of a lock returns a guard, which is a special pointer to the protected data. Threads can access the data only by dereferencing guards. When the `drop` function deallocates a guard, the connected lock is automatically released. Below shows the creation, use, and deallocation of a guard:

```
let mut g = m.lock().unwrap(); *g += 1; drop(g);
```

Due to the ownership type system of Rust, a function can use a variable only after its initialization and before it is passed to another function. Thus, threads can use guards only when the locks are held, and flow-lock mismatches never happen.

The C lock API in C2Rust-generated code is an important unsafe feature to be replaced. Since C2Rust preserves all the C types and library functions, the use of the C lock API remains in C2Rust-generated programs. It hinders the programs from benefiting from the thread safety guaranteed by the Rust lock API. The main challenge is to efficiently compute precise data-lock and flow-lock relations through static analysis. The replacement is impossible without this information because the Rust lock API requires these relations to be explicit.

B. Output Parameters

Output parameters are parameters for output rather than input. Unlike some languages supporting output parameters as a primitive language feature, C mimics output parameters with pointers. To use an output parameter, a function inputs a pointer and writes a value to it without reading it. There are two main reasons for using output parameters in C.

One reason is to return multiple values. In languages with tuples, a function returns multiple values by returning a tuple. Since C does not have tuples, a function has to return a single value as a return value and the other values through output parameters. For example, the following function returns both quotient and remainder of division using an output parameter:

```
int div(int n, int d, int *r) { int q = n / d;
  *r = n - q * d; return q; }
```

The other reason is to implement semipredicates. A *semipredicate* is a function that may fail, e.g., a division function fails when the divisor is zero. A common solution to implement semipredicates is to use exceptions. A semipredicate throws an exception when it fails, and its caller recognizes the failure with an exception handler. However, C does not have exceptions, and programmers often use output parameters to implement semipredicates. A semipredicate is implemented as a function with an output parameter. On success, the function writes the result to the output parameter and returns a value indicating success. On failure, the function returns a value indicating failure without writing to the output parameter. For example, the following function returns the quotient of division only when the divisor is nonzero:

```
int div(int n, int d, int *q) {
  if (d == 0) return -1; *q = n / d; return 0; }
```

While being useful in C, it is recommended to avoid output parameters if possible [1]. Output parameters harm readability. Parameters are for input, not output. A function producing a value through an output parameter is more difficult to understand than a function simply returning its result. Moreover, output parameters also harm safety. The caller of a function with an output parameter often defines an uninitialized variable and passes a pointer to the variable to the function. However, if the function is a semipredicate, the variable may not be initialized even after the function returns, and reading the variable without care would cause undefined behavior.

Rust provides safe features to remove the use of output parameters: tuples and options. A function returning multiple values can return a tuple like below:

```
fn div(n: i32, d: i32) -> (i32, i32) {
  let q = n / d; let r = n - q * d; return (q, r); }
```

A semipredicate can return an option, which is `Some` or `None`. `Some` indicates success and has an inner value; `None` indicates failure and does not contain any value. Below is an example of using an option to implement a semipredicate:

```
fn div(n: i32, d: i32) -> Option<i32> {
  if d == 0 { return None; }
  let q = n / d; return Some(q); }
```

It is important to remove output parameters in C2Rust-generated code by replacing them with tuples and options. Since C2Rust does not add or remove parameters of functions, output parameters remain during the translation, causing harm to both readability and safety of the generated Rust code. This problem is challenging because finding output parameters is nontrivial. In addition, identifying return values indicating success and failure is another challenge in the case of semipredicates. It is impossible to correctly transform the callers' failure handling routines without such information. We need efficient static analysis to find output parameters and discover the meaning of the return values of semipredicates.

IV. Achieved Results

To replace the C lock API in C2Rust-generated code with the Rust lock API, we proposed Concrat, which consists of C2Rust, a static analyzer, and a Rust code transformer [20]. Fig. 1 illustrates the workflow of Concrat. The static analyzer

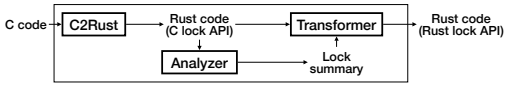

Fig. 1: Workflow of Concrat

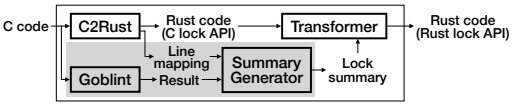

Fig. 2: Workflow of Concrat$_G$ (gray: differences from Concrat)

performs our novel dataflow analyses on the C2Rust-generated code to produce a lock summary describing data-lock and flow-lock relations. The transformer replaces the lock API according to the lock summary.

To compare our analyzer with Goblint [34], [37], the state-of-the-art static analyzer for concurrent C programs, we additionally built Concrat$_G$, which uses Goblint instead of ours. Fig. 2 shows the workflow of Concrat$_G$. Since Goblint analyzes C, our summary generator converts Goblint's analysis result to a lock summary using the C-to-Rust line mappings produced by C2Rust.

We evaluate Concrat with 46 real-world concurrent C programs collected from GitHub. Our experiments are on an Ubuntu machine with Intel Core i7-6700K (4 cores, 8 threads, 4GHz) and 32GB DRAM. Our implementation and evaluation data are publicly available [21].

Our evaluation shows that the transformer is scalable, widely applicable, and correct. As Fig. 3 shows, the transformation time is proportional to the code size, and it takes only 2.5 seconds to transform 66 KLOC. We consider the transformer applicable to a program if the program is compilable after the transformation. Among 46, 29 are compilable, 5 are compilable requiring manual fixes of a few lines, and 12 are not. The reasons for the 12 failures are classified into three categories: functions conditionally acquiring locks, function pointers, and pointers to locks. We consider the transformer correct if the program passes all of its test cases after the transformation. Among the 34 compilable programs, 20 have test cases. Among the 20, 1 fails before C2Rust's translation, 1 fails after C2Rust's translation, 17 succeed after our transformation, and 1 fails after our transformation. While we do not have a formal correctness proof, our design that transforms a `lock` function call to a `lock` method call and an `unlock` function call to the drop of a guard whose destructor unlocks the connected lock informally justifies the correctness.

Our evaluation shows that the analyzer is more scalable and more precise than Goblint. The analyzer requires only 4.3 seconds to analyze 66 KLOC, while Goblint fails to analyze 27 programs due to internal errors or the 24-hour time limit and analyzes the other 19 programs 1.1× to 3923× more slowly than ours, as shown in Fig. 4. We consider an analyzer precise if its lock summary leads the transformer to produce compilable code because an imprecise lock summary makes the transformed code uncompilable. Among the 19 programs Goblint can analyze, 2 transformed programs do not compile when using ours, but 6 do not when using Goblint.

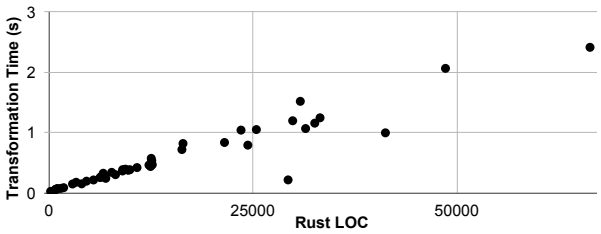

Fig. 3: Transformation time according to Rust LOC

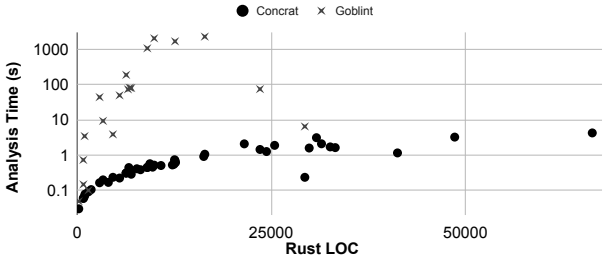

Fig. 4: Analysis time according to Rust LOC

V. Future Plans

We will propose an approach to replace output parameters with tuples and options. It requires static analyses that detect the use of output parameters and discover the meaning of the return values of semipredicates. We also need to design a Rust code transformer using the results of the static analyses. We are currently collecting C code using output parameters to identify common code patterns of output parameter usage in real-world code. We will be able to develop the analyzer and the transformer based on the identified patterns. We believe that we can evaluate the work similar to our previous work on lock API. We will evaluate the scalability, applicability, and correctness of the transformer. We will also evaluate the scalability and precision of the analyzer. We expect to finish this work by the end of 2023.

After resolving the output parameter problem, we will identify another important unsafe feature in C2Rust-generated code and replace it with a proper safe feature. While we do not have a specific feature yet, possible candidates include replacing `malloc` with `Box`, functions taking `void *` pointers with generic functions, and loops with iterators and higher-order functions. We cannot describe a concrete evaluation plan yet, but we anticipate that a similar evaluation process to our previous work will be possible. We expect to finish this work by the end of 2024.

Acknowledgment

This research was supported by National Research Foundation of Korea (NRF) (Grants 2022R1A2C200366011 and 2021R1A5A1021944), Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (2022-0-00460), and Samsung Electronics Co., Ltd (G01210570).

REFERENCES

[1] CA1021: Avoid out parameters. https://learn.microsoft.com/previous-versions/visualstudio/visual-studio-2015/code-quality/ca1021-avoid-out-parameters.

[2] Fuchsia guides: Rust. https://fuchsia.dev/fuchsia-src/development/languages/rust.

[3] The Rust programming language. http://rust-lang.org/.

[4] IEEE standard for information technology—portable operating system interface (POSIX). *IEEE Std. 1003.1-2017*, 2017.

[5] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. https://arxiv.org/abs/2103.06333, 2021.

[6] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Summarize and generate to back-translate: Unsupervised translation of programming languages. https://arxiv.org/abs/2205.11116, 2022.

[7] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, page 290–301, New York, NY, USA, 1994. Association for Computing Machinery.

[8] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In Rocco De Nicola, editor, *Programming Languages and Systems*, pages 520–535, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[9] James R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006. Special Issue on The Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA '04).

[10] Sergio De Simone. Linux 6.1 officially adds support for Rust in the kernel. https://www.infoq.com/news/2022/12/linux-6-1-rust, dec 2022.

[11] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. Checked C: Making C safe by extension. In *2018 IEEE Cybersecurity Development (SecDev)*, pages 53–60, 2018.

[12] Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. Translating C to safer Rust. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[13] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. https://arxiv.org/abs/2002.08155, 2020.

[14] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28(6):1035–1087, nov 2006.

[15] David Gay and Alex Aiken. Memory management with explicit regions. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, page 313–323, New York, NY, USA, 1998. Association for Computing Machinery.

[16] David Gay and Alex Aiken. Language support for regions. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, page 70–80, New York, NY, USA, 2001. Association for Computing Machinery.

[17] Manish Goregaokar. Fearless concurrency in Firefox Quantum. https://blog.rust-lang.org/2017/11/14/Fearless-Concurrency-In-Firefox-Quantum.html, nov 2017.

[18] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, page 282–293, New York, NY, USA, 2002. Association for Computing Machinery.

[19] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. GraphCodeBERT: Pre-training code representations with data flow. https://arxiv.org/abs/2009.08366, 2020.

[20] Jaemin Hong and Sukyoung Ryu. Concrat: An automatic C-to-Rust lock API translator for concurrent programs. https://arxiv.org/abs/2301.10943, 2023.

[21] Jaemin Hong and Sukyoung Ryu. Concrat: An automatic C-to-Rust lock API translator for concurrent programs (artifact). https://doi.org/10.5281/zenodo.7573490, January 2023.

[22] Tim Hutt. Would Rust secure cURL? https://blog.timhutt.co.uk/curl-vulnerabilities-rust/, jan 2021.

[23] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

[24] Steve Klabnik and Carol Nichols. *The Rust programming language*. https://doc.rust-lang.org/book.

[25] Marie-Anne Lachaux, Baptiste Roziere, Marc Szafraniec, and Guillaume Lample. DOBF: A deobfuscation pre-training objective for programming languages. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 14967–14979. Curran Associates, Inc., 2021.

[26] Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James R. Cordy, and Ahmed E. Hassan. In Rust we trust – a transpiler from unsafe C to safer Rust. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 354–355, 2022.

[27] Aravind Machiry, John Kastner, Matt McCutchen, Aaron Eline, Kyle Headley, and Michael Hicks. C to Checked C by 3C. *Proc. ACM Program. Lang.*, 6(OOPSLA1), apr 2022.

[28] Nicholas D. Matsakis and Felix S. Klock. The Rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103–104, New York, NY, USA, 2014. Association for Computing Machinery.

[29] Mend. What are the most secure programming languages? https://www.mend.io/most-secure-programming-languages/.

[30] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, page 128–139, New York, NY, USA, 2002. Association for Computing Machinery.

[31] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 20601–20611. Curran Associates, Inc., 2020.

[32] Baptiste Roziere, Jie M. Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. Leveraging automated unit tests for unsupervised code translation. https://arxiv.org/abs/2110.06773, 2021.

[33] Andrew Ruef, Leonidas Lampropoulos, Ian Sweet, David Tarditi, and Michael Hicks. Achieving safety incrementally with Checked C. In Flemming Nielson and David Sands, editors, *Principles of Security and Trust*, pages 76–98, Cham, 2019. Springer International Publishing.

[34] Michael Schwarz, Simmo Saan, Helmut Seidl, Kalmer Apinis, Julian Erhard, and Vesal Vojdani. Improving thread-modular abstract interpretation. In *Static Analysis: 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17–19, 2021, Proceedings*, page 359–383, Berlin, Heidelberg, 2021. Springer-Verlag.

[35] Jeff Vander Stoep and Stephen Hines. Rust in the Android platform. https://security.googleblog.com/2021/04/rust-in-android-platform.html, apr 2021.

[36] Marc Szafraniec, Baptiste Roziere, Hugh Leather, Francois Charton, Patrick Labatut, and Gabriel Synnaeve. Code translation with compiler representations. https://arxiv.org/abs/2207.03578, 2022.

[37] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. Static race detection for device drivers: The Goblint approach. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 391–402, New York, NY, USA, 2016. Association for Computing Machinery.

[38] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R. Lyu. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 382–394, New York, NY, USA, 2022. Association for Computing Machinery.

[39] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. https://arxiv.org/abs/2109.00859, 2021.

[40] Frances Wingerter. C2Rust is back. https://immunant.com/blog/2022/06/back/, jun 2022.