

On the Translation Between C and Rust

^{1st} Daniel Dean

The College of Information Sciences and Technology
The Pennsylvania State University
State College, PA, United States
dpd5518@psu.edu

Abstract—C and C++ have been dominant languages in the systems programming space since the 1970s. Both of these languages are immensely powerful, but this power comes at the cost of security. Memory security issues have plagued this style of programming since its inception. The tradeoff between speed and security is not necessary, however. Rust is a programming language with the speed of C/C++ and the memory safety guarantees of a garbage collected languages. The tradeoff that Rust makes is for complexity and a slower speed of development. The problem: how can we transfer from a C dominated space to a safer Rust dominated space? This paper will give a background on the security mechanisms of Rust, the current state of automatic translation from C to Rust, and discuss what the future may entail for security minded individuals in the systems programming space.

Index Terms—C/C++, Rust, Memory Safety, C2Rust

I. INTRODUCTION

In the sub-field of systems development, C/C++ are the typical choice when it comes to writing complex yet highly efficient systems. While this offers fast software with relatively expedient speed of development, it comes with one major cost—safety. More specifically, C has had known memory security problems that allow for full take over of a process for more than 40 years [1]. Some of these memory issues include buffer overflow attacks, and use-after-free bugs.

There have been many proposed solutions to these memory safety issues. But the simplest is re-implementing code bases written in C in a memory safe language. While this could be any memory safe language, Rust has recently gained popularity, evident in its appearance in the Linux kernel [2]. Furthermore, in early 2024 the United States Government released a report on the dire nature of software security as it relates to national cyber security goals. The report emphasizes the importance that programming languages play in the development of secure software. They offer the Rust programming language as a potential solution to decades of dubious memory management caused by the C languages [3]. This may signify a push from the government to require that new applications must be written in Rust, or legacy C applications must be rewritten in Rust – a simple solution, but a logistical nightmare.

While Rust offers high performance memory safety, many of the use-cases in which it is needed already have a C implementation. The transition from C to Rust poses a massive logistical problem. This problem is compounded by Rust’s relatively steep learning curve [4], limiting access to talent that

could feasibly implement these C programs in Rust. Attempts to fully automate this process have not yet proven fruitful.

II. BACKGROUND

The reason that code-bases such as the Linux kernel are written in C is the high level of control these languages offer over the machine, allowing for highly performant code. Two principal aspects of C that aid in performance are the manual memory management and low runtime overhead. The traditional approach to ensure memory safety is to automate away memory management with a garbage collector; Lisp and Java are examples of garbage collected languages. Notwithstanding bugs in the implementation of the garbage collector, this guarantees memory safety, but results in more overhead, as the garbage collector must exist at runtime, causing performance degradation. There are a plethora of algorithms for achieving garbage collection, all of which are outside the scope of this paper. Some of the traditional techniques can cost four times the amount of CPU cycles as opposed to the non-garbage-collected program, while others can have negligible effects on the program performance [5]. More recent techniques have brought this upper bound down to closer to two times the CPU cycles, which is still not ideal [6]. Outside of the performance costs, garbage collectors add a runtime to the execution of a language, which could result in unintended program side effects when developing at the systems-level. This approach to memory management creates a false dichotomy between memory safety and performance.

A. The Borrow Checker

Rust achieves memory safety without a garbage collector through the enforcement of strict compile time rules: Lifetimes, Ownership, Borrowing, and Exclusive Mutability, all of which are checked at compile time [7]. Lifetimes simply denote how long a variable is valid in a program, which the rust compiler’s borrow checker typically implicitly derives. For more complex lifetimes, the borrow checker will fail to automatically denote the lifetime and thus it must be done manually, which is a point of contention for new Rust programmers [4]. Borrowing and ownership are two related ideas in Rust where each variable owns a value, which changes when another variable takes ownership, or the value is passed to a function. Borrowing is the passing of a reference to a variable or function, without changing ownership. Exclusive

mutability is ensured through the compiler through the following invariant: there is either one mutable reference or an arbitrary amount of immutable references [4], [8].

It is worth noting that there are also checks that are compiled into the machine code for each Rust program, which could be considered overhead. Performance wise, the most impactful of these is the bounds checker, which checks for out-of-bounds accesses [7]. This suite of compile time checks guarantee memory safety so long as they are not deactivated using an `unsafe` block [8].

B. Unsafe Rust

The Borrow Checker was designed to be conservative by nature. This assures that any program that passes the Borrow Checking phase of compilation (Borrow Checkable) will be memory safe. It is not the case, however, that every memory safe program will pass by the Borrow Checker's standards; this is intended [9](see Fig. 1). This unsafe feature can be encapsulated in safe functions, which is extensively used in the Rust project itself [10]. This feature makes Rust capable of compiling any program, and can introduce memory bugs. Ideally, however, these unsafe blocks will be small, and will be under high levels of scrutiny.

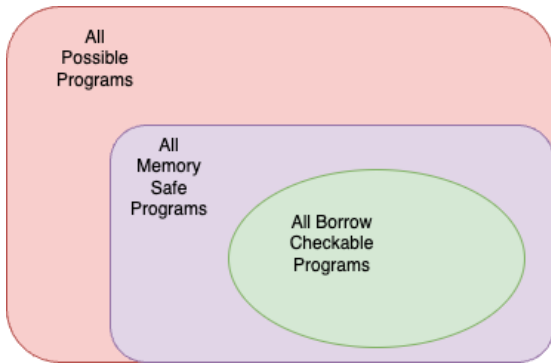


Fig. 1. Conceptual map of the Borrow Checkers conservative nature

III. EXAMPLE OF VULNERABILITIES

In this section, we will explore examples of some common C vulnerabilities. We will consider two examples: stack buffer overflows, and use-after-free. This section will include code, compilation errors, and explanations. All code for this section is available on GitHub [11].

It is also worth noting the difference between crashing an throwing exceptions, or errors, as this is a crucial difference between Rust and C/C++. Behavior like crashing and throwing exceptions may allow for a programmer to easily find and triage the bug, or allow for the analysis of a crash report. The difference between a crash and an exception is that a crash is completely out of the program's control—it is a system level problem. Exceptions, on the other hand, are still within a program and can often be caught and handled inside the program at runtime. An example of a crash would be a segmentation fault, whereas a an exception would be failing

a bounds check for an array. Crashes can be used to mount attacks, whereas errors cannot.

A. Stack Buffer Overflow

A buffer overflow is a vulnerability where the bounds of a buffer are overwritten. This can be used to overwrite the return address of the current stack frame, allowing for arbitrary code execution. Consider the following proof of concept that takes in a user-provided string, and compares it with a hard-coded password. It is certainly feasible that a C programmer may assume that the `const char secret` will remain what they set it to at compilation time.

```
#include <stdio.h>
#include <string.h>

#define MAX_BUF 20

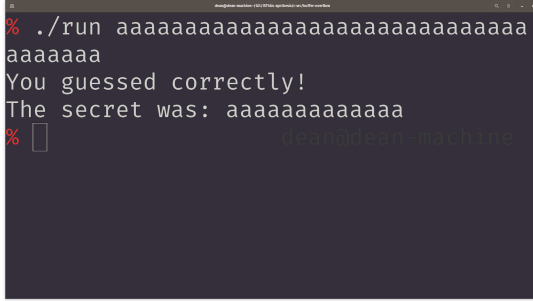
int main(int argc, char **argv) {
    const char secret[] = "letmein";
    char buf[MAX_BUF];
    if (argc < 2){
        printf("Usage: ./run <guess>\n");
        return 1;
    }
    if (argc > 1) {
        strcpy(buf, argv[1]);
    }
    if (0 == strcmp(buf, secret, sizeof(secret))) {
        printf("You guessed correctly!\n");
        printf("The secret was: %s \n", secret);
        return 0;
    }
    printf("' %s' was not the secret.\nTry again later!\n", buf);
    return 0;
}
```

```
use std::env;
fn main() {
    let argv: Vec<String> = env::args().collect();
    if argv.len() < 2 {
        println!("Usage: cargo run <guess>");
        return;
    }
    let secret: String = String::from("letmein");
    let guess: String = argv[1].clone();
    if secret == guess {
        println!("You guessed correctly!");
        println!("The secret was: {} ", secret);
    } else {
        println!("'{}' was not the secret.", guess);
        println!("Try again later!\n");
    }
}
```

Fig. 2. Example C code with a buffer overflow vulnerability and the equivalent Rust code

The program works as expected with sufficiently short input. As stated earlier, this program is vulnerable to a buffer overflow attack. This is due to the lack of bounds checking on the buffer `buf`, which can be overwritten by `strcpy`. By the construction of this program, we are able to overwrite `secret`

in such a way that the comparison between the two strings always passes (Fig.3). Importantly in Fig.3, the **secret** variable has been completely overwritten by the user's input—which is a side effect of the attack, and may have down-stream effects in the program (i.e. if this variable is used multiple times). This type of vulnerability is especially nefarious, as the program doesn't crash or throw an exception.



```
% ./run aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaa
You guessed correctly!
The secret was: aaaaaaaaaa
% 
```

Fig. 3. A successful overflow attack

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct String {
    char *s;
    int len;
};

void consume(struct String * s){
    printf("string struct: {'%s',
        len=%d}\n",s->s,s->len);
    free(s);
}

int main(int argc, char **argv) {
    struct String *s = malloc(sizeof(struct String));
    if (argc < 2) {
        printf("Usage: ./run <string>\n");
        return 1;
    }
    s->s = argv[1];
    s->len = strlen(s->s);
    consume(s);
    consume(s);
    return 0;
}
```

B. use-after-free

A use-after-free vulnerability is when a programmer allows for the use of heap allocated memory after it has already been freed, this may result in a crash, or a malicious corruption of memory [12]. Consider the code block in Fig. 4. This is a simple example that always crashes, but it compiles with no warnings. The reason it crashes is the the function **consume**, consumes the variable **s** in the sense that it frees the allocated variable. It succeeds the first time, but fails the second time, because the memory that the pointer **s** points to is already free (from the first call). We see the code failing in Fig. 5

C. Rust Implementations

We can rewrite these programs to show how they some of these crashes are solved by re-implementing the code in Rust. In Fig 2, the resulting Rust program does not have a buffer overflow vulnerability. This is because it uses the Rust String type, which is different than the array of characters that C uses. Furthermore, this String type is dynamically allocated on the program's heap [13]. Next, we see in Fig. 6, that the code provided does not compile. The reason for this is that the ownership of **s** is moved into **consume**, and not returned, thus dropped. The purpose of this example is to show that writing the bug the same way as we wrote in C is not possible in safe Rust. This also means that any automatic translation must use unsafe Rust to correctly capture the functionality of the program, or lack thereof.

IV. LITERATURE REVIEW

A. C2Rust

C2Rust is a designed with the express purpose of generating a usable Rust implementations from code written in C [14]. To that end, the project is not concerned with implementing the C code in safe Rust, rather it is principally concerned with the initial translation between the languages. It succeeds in this

```
use std::env;

fn consume(s: String) {
    println!("String: ('{}', {})", s,s.len());
}

fn main() {
    let argv: Vec<String> = env::args().collect();
    if argv.len() < 2 {
        println!("Usage: cargo run <string>");
        return;
    }
    let s: String = argv[1].to_string();
    consume(s);
    consume(s);
}
```

Fig. 4. Example C code with use-after-free bug and the equivalent Rust code

goal. It achieves this through extensive use of unsafe Rust, which allows the automatic translation to mirror exactly what the C code is doing, barring some limitations [15]. The next logical step is to transform the unsafe Rust into safe Rust if it is possible. This, however, is not trivial and is the target of ongoing research.

Much of the research on a safer translation has been focused on static analysis, such as the CROWN tool [17], which is able to produce safer code than c2rust. This tool leverages ownership analysis to infer the owners of each variable, allowing for a safer translation, barring some constraints. Another static analysis approach iterates on the C2Rust translation as well. The approach taken by Hong attempted to replace the unsafe Lock APIs output parameters generated by the C2Rust output, and replace them with safe Rust APIs [16]. Lock API is a concurrent programming API that is used to prevent race conditions. Passing output parameters is a technique used in

```

$ ./run "hello world"
string struct: {'hello world', len=11}
[2] 62229 segmentation fault (core dumped) ./run "hello world"
139

```

Fig. 5. C program from Fig. 4 unconditionally crashing

```

$ cargo run
    Compiling use-after-free v0.1.0 (/home/dean/Git/STDA-synthesis/rust-ry/use-after-free)
error[E0502]: cannot borrow `x` as mutable because it is already borrowed
  --> src/main.rs:15:13
   |
13 | let s: String = arg(1).to_string();
   |               ^^^^^^^^^^^^^^^^^^^^
14 |     move occurs because `x` has type `String`, which does not implement the `Copy` trait
   |     ~~~~~
15 |     consume(s);
   |     ^^^^^^^^^ value moved here
   |
   = note: value moved here after move

help: consider changing this parameter type in function `consume` to borrow instead if owning the value isn't necessary
  --> src/main.rs:15:13
   |
13 | let s: String = arg(1).to_string();
   |
14 |     consume(s);
   |     ~~~~~
15 |     consume(s);
   |     ^^^^^^^^^ value moved here after move

help: consider changing this parameter type in function `consume` to borrow instead if owning the value isn't necessary
  --> src/main.rs:15:13
   |
13 | let s: String = arg(1).to_string();
   |
14 |     consume(s);
   |     ~~~~~
15 |     consume(s);
   |     ^^^^^^^^^ value moved here after move

help: consider cloning the value if the performance cost is acceptable
  --> src/main.rs:15:13
   |
13 | let s: String = arg(1).to_string();
   |
14 |     consume(s);
   |     ~~~~~
15 |     consume(s);
   |     ^^^^^^^^^ value moved here after move

For more information about this error, try `rustc --explain E0502`.
error: could not compile `use-after-free` (lib) due to previous error

```

Fig. 6. Rust program from Fig. 4 failing to compile

C where a result is written directly to a location pointed to by a parameter of the function. Hong’s work is promising, yet preliminary. A slightly different approach is to generate a control-flow graph (CFG) of the C program, then convert that C CFG into a Rust CFG [18]. This offers a distinct approach to that of C2Rust, yet is not able to solve the unsafe problem.

The most effective tool to date is CRustS, which can achieve safe to unsafe ratios ranging from 60% to 99% given the output of C2Rust [19]. This is achieved through the use of a source-to-source translator with hand crafted rules to optimize for safety. Importantly, this tool can be adjusted to adapt to the code base that it is being used on [19].

B. Use of LLMs

Recently, the programming world has been adopting Large Language Models (LLMs) to aid in programming. This is a very common use of LLMs, however it poses some problems. Chiefly, the current generation of LLMs is extremely sensitive to slight differences in prompting [20] [21]. This is only compounded by the design of LLMs, in which they sample from distribution of responses.

LLMs have also been investigated in work more pertinent to this paper—fixing Rust compilation errors. RustAssistant is a tool that leverages LLMs to solve Rust compilation problems [22]. This paper finds a significant increase in success in solving compilation errors from GPT-3 to GPT-4 [22], implying that this tool will only become more prevalent in the future of software development. This LLM powered translation is not perfect, however. Large real-world projects offer problems for LLMs as the amount of context to consider

could be massive, and some is unrepresentable, such as file structure [23]. Overall the main problem with LLMs is their unpredictability and relatively high degree of randomness as opposed to traditional translation tools. One possible method to remedy this is the use of formal verification tools such as Aeneas [24] to prove correctness at each end of the translation process.

V. C++ SECURITY

In this section, we investigate the development of C++ security mechanisms as an alternative to transferring away from the C language family. This is a favorable approach to the absolute abandonment of C/C++, as it may prove to be more feasible than a complete rewrite, not to mention the benefits that come with the years of experience accrued by engineers writing C/C++ for their entire careers.

Many projects have made progress in C/C++ memory safety including (but not limited to) SafeC, CCured, Cyclone, and Iron Clad C++ [25]. Furthermore frameworks to guarantee security at compile time, similar to the borrow checker, have been designed for C++ [26]. This is preliminary work, however, as it does not deal with all classes of memory issues. Code Pointer Integrity (CPI) is another method concerned with securing function pointers in a C/C++ program. There are different levels of guarantees associated with CPI, some of which offer strong amounts of safety guarantees with low amounts of overhead [27]

Unfortunately, however most modern C++ code has not made significant improvements to memory safety from its predecessor C [25]. The lack of compile-time guarantees is a large problem, as any other method will introduce some amount of overhead during execution. More broadly, C++ was designed to be an object oriented version of C, not a safer version of C. Rust is capable of sidestepping this issue as it was designed with safety in mind, not as an afterthought. This approach to memory safety is much more realistic than being able to refactor existing C/C++ code into Rust, however. Furthermore, C/C++ developers could implement some of the ideas from Rust, such as ownership and borrow checking (albeit in a heavily modified form) into their standards moving forward. The main benefit of this would be maintaining the status quo of the C family languages being dominant.

VI. DISCUSSION AND CONCLUSION

We have shown in this paper that we are trending towards a very near future where it becomes feasible to completely write all C/C++ code bases in Rust, considering the current state of CRustS, as well as the rapidly developing field of LLMs for coding assistance. While these breakthroughs sounds like it would lead to a utopia of bug-free code, there still may be some barriers break down before we get to this point in software’s history. In this scenario, it may be unwise to completely change decades old code bases for safety, if it comes at the price of maintainability. In the current state of software development, most code is written by hand, and thus it would be extremely unwise to abandon this precedent to

prefer a generated code base, even if it does result in more memory safe code. While the transition into a status quo that no longer produces unsafe code has been happening slowly, this may be by necessity. A quick adoption of any technology is bound to present unforeseen challenges that otherwise would have otherwise been worked out over a slower evolution.

While the complete rewrite all of the memory unsafe software in existence may not happen, it is certainly possible to stop writing new unsafe software. Rust is a perfect candidate to supplant the C family of languages in the future of software development. This is shown through its safety features, speed, interoperability with C code, and the current tooling that allows for transfer between languages. The transition into a Rust-dominated space should be executed with the same caution that adopting any new technology requires.

This paper has discussed the security mechanisms of Rust, and how these mechanisms affect performance. We demonstrated different security vulnerabilities, and how they are averted when developing in Rust. We then summarized the current state of the art in translation between C/C++ and Rust, as well as research into C/C++ memory safety to potentially negate the need for translation. Finally, we conclude that while the translation between C/C++ and Rust feasible in the present/very near future, it must be effectuated with caution and a deep consideration of how it will affect the maintainability of large code bases.

REFERENCES

- [1] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," IEEE Symposium on Security and Privacy, May 2013, doi: <https://doi.org/10.1109/sp.2013.13>
- [2] L. Torvalds, "Linux/Rust at master," GitHub, 2024. Available: <https://github.com/torvalds/linux/tree/master/rust>. [Accessed: Mar. 18, 2024]
- [3] The United States Government, "Back to The Building Blocks: A Path Towards Secure and Measurable Software," 2024. Available: <https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf>
- [4] S. Zhu, Z. Zhang, B. Qin, A. Xiong, and L. Song, "Learning and programming challenges of rust," Proceedings of the 44th International Conference on Software Engineering, May 2022, doi: <https://doi.org/10.1145/3510003.3510164>
- [5] M. Hertz and E. D. Berger, "Quantifying the performance of garbage collection vs. explicit memory management," Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Oct. 2005, doi: <https://doi.org/10.1145/1094811.1094836>[4]Z.
- [6] Cai, S. M. Blackburn, M. D. Bond, and M. Maas, "Distilling the Real Cost of Production Garbage Collectors," 2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), May 2022, doi: <https://doi.org/10.1109/ispass55109.4022.00005>. Available: <https://users.cecs.anu.edu.au/~steveb/pubs/papers/lbo-ispass-2022.pdf>. [Accessed: Feb. 24, 2023]
- [7] Y. Zhang, Y. Zhang, Georgios Portokalidis, and J. Xu, "Towards Understanding the Runtime Performance of Rust," 37th IEEE/ACM International Conference on Automated Software Engineering, Oct. 2022, doi: <https://doi.org/10.1145/3551349.3559494>
- [8] H. Xu, Z. Chen, M. Sun, Y. Zhou, and M. R. Lyu, "Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs," ACM Transactions on Software Engineering and Methodology, vol. 31, no. 1, pp. 1–25, Jan. 2022, doi: <https://doi.org/10.1145/3466642>
- [9] Rust Foundation, "Unsafe Rust - The Rust Programming Language," doc.rust-lang.org. Available: <https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html>
- [10] Rust Foundation, "rust-lang/rust," GitHub, Apr. 15, 2024. Available: <https://github.com/rust-lang/rust/tree/master>. [Accessed: Apr. 15, 2024]
- [11] D. Dean, "dandeandean/masters-capstone," GitHub, Apr. 15, 2024. Available: <https://github.com/dandeandean/masters-capstone>. [Accessed: Apr. 15, 2024]
- [12] OWASP, "Using freed memory — OWASP," owasp.org. Available: https://owasp.org/www-community/vulnerabilities/Using_freed_memory
- [13] The Rust Programming Language, "Strings - Rust By Example," doc.rust-lang.org. Available: <https://doc.rust-lang.org/rust-by-example/std/str.html>. [Accessed: Apr. 16, 2024]
- [14] C2Rust, "C2Rust Demonstration," c2rust.com, Jun. 2019. Available: <https://c2rust.com/>
- [15] Immunant, "Known Limitations of Translation," GitHub. Available: <https://github.com/immunant/c2rust/wiki/Known-Limitations-of-Translation>. [Accessed: Apr. 16, 2024]
- [16] J. Hong, "Improving Automatic C-to-Rust Translation with Static Analysis," May 2023, doi: <https://doi.org/10.1109/icse-companion58688.2023.00074>
- [17] H. Zhang, C. David, Y. Yu, and M. Wang, "Ownership Guided C to Rust Translation," Lecture Notes in Computer Science, pp. 459–482, Jan. 2023, doi: https://doi.org/10.1007/978-3-031-37709-9_22
- [18] X. Han, B. Hua, Y. Wang, and Z. Zhang, "RUSTY: Effective C to Rust Conversion via Unstructured Control Specialization," Dec. 2022, doi: <https://doi.org/10.1109/qrs-c57518.2022.00122>
- [19] M. Ling, Y. Yu, H. Wu, Y. Wang, J. R. Cordy, and A. E. Hassan, "In rust we trust," May 2022, doi: <https://doi.org/10.1145/3510454.3528640>
- [20] M. Acher, José Galindo Duarte, and Jean-Marc Jézéquel, "On Programming Variability with Large Language Model-based Assistant," Aug. 2023, doi: <https://doi.org/10.1145/3579027.3608972>
- [21] M. Macedo, Y. Tian, F. R. Cogo, and B. Adams, "Exploring the Impact of the Output Format on the Evaluation of Large Language Models for Code Translation," arXiv (Cornell University), Mar. 2024, doi: <https://doi.org/10.1145/3650105.3652301>
- [22] P. Deligiannis, A. Lal, N. Mehrotra, and A. Rastogi, "Fixing rust compilation errors using llms," arXiv preprint arXiv:2308.05177, 2023.
- [23] R. Pan et al., "Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code," 2024, doi: <https://doi.org/10.1145/3597503.3639226>. Available: <https://arxiv.org/pdf/2308.03109.pdf>. [Accessed: Feb. 07, 2024]
- [24] S. Ho and J. Protzenko, "Aeneas: Rust verification by functional translation," Proceedings of the ACM on programming languages, vol. 6, no. ICFP, pp. 711–741, Aug. 2022, doi: <https://doi.org/10.1145/3547647>
- [25] C. DeLozier, "How Close Is Existing C/C++ Code to a Safe Subset?," Journal of Cybersecurity and Privacy, vol. 4, no. 1, pp. 1–22, Mar. 2024, doi: <https://doi.org/10.3390/jcp4010001>. Available: <https://www.mdpi.com/2624-800X/4/1/1>. [Accessed: Apr. 17, 2024]
- [26] M. Busi, P. Degano, and L. Galletta, "Towards effective preservation of robust safety properties," Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, Apr. 2022, doi: <https://doi.org/10.1145/3477314.3507084>
- [27] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Codepointer integrity," in The Continuing Arms Race: CodeReuse Attacks and Defenses, 2018, pp. 81–116.