Practicing Safer C:

Provably Safe Invariants for Safe Integration

Harald Ruess

Entalus LLC
2071 Gulf of Mexico Drive
Longboat Key, FL 34228

harald.ruess@entalus.com

Natarajan Shankar

SRI International
333 Ravenswood Ave
Menlo Park, CA 94025

shankar@csl.sri.com

www.entalus.com

June 2024

The Need for Memory Safety. The *White House Office of the National Cyber Director* (ONCD) recently argued that technology vendors can prevent entire classes of vulnerabilities from entering the digital ecosystem by adopting memory-safe programming languages [1]. Many have wondered why the highest echelons of the U.S. government would be concerned with the seemingly technical housekeeping issues of computer memory layout that primarily affect programmers.

Memory safety vulnerabilities are extremely common (Figure 1) [3], and industry report indicates that 70% or more of Common Vulnerabilities and Exposures (CVEs) are memory safety vulnerabilities [30-33]. The majority of cybersecurity vulnerabilities over the past few decades have been enabled by memory safety vulnerabilities, including the Morris Worm [22], Slammer Worm [23], Heartbleed [24], Blastpass [25], and Spectre/Meltdown [26].

Failure to satisfy memory safety properties also costs us dearly [13]: *"… memory safety bugs are widespread and one of the leading causes of vulnerabilities in memory-unsafe codebases. Those vulnerabilities endanger end users, our industry, and the broader society".* In fact, cybercrime has become an $8 trillion problem in the U.S. alone [28]. It is clear that the world would already be a safer and more secure place if the basic hardware and software supply chains could be made at least memory-safe.

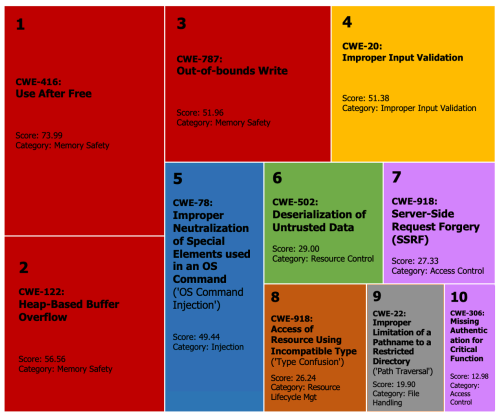

Figure 1. Top 10 Software Vulnerabilities (Source: MITRE, 2023, [13]).

Memory safety vulnerabilities affect how memory can be accessed, written, allocated, or deallocated in unintended ways. This includes violations to basic *invariant properties* such as

- Any object in memory must be written at least once before being read.
- References always point to valid memory objects.[1]
- There always is exactly one reference to a valid memory object.
- All memory objects are properly deallocated as soon as they are not used anymore.
- Access to shared global memory objects is synchronized as to prevent data races.

Any program that violates memory safety properties is considered unsafe in that it may crash (e.g., *null pointer dereferencing*), produce unintended behavior (e.g., *use-after-free*), leak sensitive information (e.g., *out-of-bounds reads*), exhibit non-deterministic behavior (e.g., *data races*), or allow program hijacking (e.g., *buffer overflows*).

Memory safety is not a single, universally accepted concept. Instead, it depends very much on the definition of the underlying programming language(s) as to what memory safety means for a program. For example, data races are defined in Java, but considered undefined behavior in C and C++.

Most system software has been and still is written in C or C++, which, although they have been greatly modernized in recent years, still place most of the burden of ensuring that safety conditions are met on the programmer. To avoid unsafe memory management, C and C++ programmers have a variety of methods, techniques, and tools at their disposal, including programming guidelines, restricting themselves to safer subsets of the programming language, and detecting and isolating memory safety violations with dynamic and static

[1] For example, a memory object is not deallocated from memory as long as one or more references point to it.

analysis. But neither can find every vulnerability, and they open other cans of worms like *false positives*.

These attempts to address the problem have only been partially successful, as two-thirds of the reported vulnerabilities in memory-unsafe languages are still related to memory issues. Consequently, there is a strong movement to move away from programming languages that are considered to be memory-unsafe [14,29].

Memory safe programming languages. Automatic memory management in the form of *garbage collection*, which periodically checks for unused memory as the program runs, is the most common technique for avoiding the vast majority of memory safety problems. As a result, most modern programming languages are based on a garbage-collecting runtime. However, garbage collection imposes a certain runtime overhead, and is therefore not used in high-performance applications, which often require direct memory access and manual memory management to improve real-time behavior and reduce memory consumption.

Essential memory safety properties can also be enforced by a type-checking compiler, since type systems ensure that well-typed programs cannot "go wrong" [4], i.e. they don't crash and their results are well defined with respect to the operational semantics of the programming language. The well-typedness of a program is enforced statically, at compile time, and/or dynamically, at run time.

Rust has a type-based approach to memory management that differs significantly from the classical approach of C or C++, where memory is managed manually through functions such as *malloc* and *free*, or by letting the garbage collector manage the resources. In contrast, its type system enforces basic memory safety properties such as *no-aliasing* through a notion of *ownership* that enforces that each object in memory is owned by at most one mutable reference, and that each memory object can be modified only by a reference that currently owns the object. When the owner reaches the end of its *lifetime*, i.e., when it is no longer used in the remainder of the computation, the object it owns is automatically *dropped*, i.e., deallocated from memory. As a result, the ownership model allows Rust to automatically and to efficiently handle memory management without the need for advanced garbage collection. For example, it has been estimated that the Rust type system would have prevented 53 out of 95 reported bugs in cURL, 47 of which are due to standard memory bugs [40]. On the other hand, Rust only protects only against 6 out of the current top 25 top CVEs, even when ’unsafe’ blocks are not in use [48].

Because Rust does not currently have a well-defined and generally agreed-upon formal semantics,[2] it is not all that clear which memory safety properties should actually be preserved by its type checker. But the Rust compiler error messages at least indicate the intended set of Rust's memory safety invariants [15], and there is progress on understanding and formalizing Rust’s ownership and borrow mechanism at least at the memory level [47].

Rust’s type system can also be used to encode program invariants beyond memory safety. First, the Rust type system enforces a certain discipline on thread programming, since a well-typed Rust program will enforce a lock on a mutex before it can be accessed, and a lock release will eventually occur after such an access. Data races are also largely prevented by Rust's

[2] Some effort has been put into formalizing various parts of Rust, such as Oxide [35], Krust [34].

ownership system, since it is impossible to alias a mutable reference ([12], Section 8.1). But, of course, the Rust type system cannot prevent general race conditions, since this is impossible. Second, certain two-state *typestates* such as *"these messages can only be sent to the client after authentication has succeeded, and not after the session has been terminated"* can be implemented directly as compile-time checks based on Rust's move semantics [20]. Third, Rust also allows for some static information flow analysis due to its ownership model, which prevents pointer aliasing problems [36]. But there is a limit to encoding data, control, and information flow invariants of programs as types, because their analysis is inherently complex or even generally undecidable.

Statically typed languages provide essential memory safety properties, but they usually also provide means to break their type systems, intentionally or not. For example, the *Obj.magic* function in Ocaml casts any object of one type to any other type. And Rust includes both a safe and an unsafe programming language, allowing the programmer to do extra things that are definitely not safe [10]. While aliasing is not an issue in safe Rust, once unsafe code blocks are involved, aliasing becomes an issue again [39]. Note also that the Rust compilation framework itself is built on a large pile of unsafe code, including unsafe programming in the Rust standard library, the C11 atomic type system, and indeed the entire LLVM compiler infrastructure for C-like languages. Thus, virtually every Rust program depends on unsafe code [41].

From C to Rust. There is currently a strong movement to adopt memory-safe and efficient programming languages such as Rust [14,17,29]. However, the sheer size and complexity of existing software stacks poses significant challenges to a smooth transition to memory safety.

A manual reimplementation effort of existing C and C++ code into a memory-safe language is costly, error-prone, and doesn't scale well. The obvious question, then, is whether we can automate much of the translation and rewriting process to make legacy migration practical and scalable with minimal manual effort. Previous attempts to automatically translate C programs to Rust have been purely syntactic, producing memory and thread-unsafe Rust code that closely mimics the original C code and explicitly bypasses the Rust compiler's safety checks [18,19]. While these tools may provide a good starting point for automated translation, they leave the hard work of manually reasoning about the safety properties of the translated program and rewriting the code to allow the Rust compiler to verify those properties to the developer.

An empirical study of the unsafety in translated Rust programs and its causes concludes that it is currently rather unclear how, if at all, to systematically translate a memory unsafe language like C into safe Rust, and how much manual effort is required to do so [38]. Moreover, specifying and verifying the correctness of a C-to-Rust translator, safe or unsafe, or even validating the correctness of individual translations is a daunting task in itself.

Instead, we propose what appears to be a more practical way to integrate C with Rust in a way that is memory safe. The first step is to establish essential memory safety invariants over the C code. Second, this memory-safe C code is syntactically transformed into, say, unsafe Rust using existing C-to-Rust tools. The memory safety provisions in the first step guarantee that the unsafe C-like blocks do not compromise the memory safety of the safe Rust. The important thing to remember is that the unsafe code is encapsulated in a safe abstraction [39]. This alternative approach from C to Rust relies heavily on ontic typing to establish essential memory safety invariants for C programs.

Ontic typing. The key idea of *ontic typing* [21] is that a type is not just a collection of entities, but that it captures a *phenomenal* aspect of the thing being represented, so that we can operate meaningfully on the representation. Technically, ontic types form a subtype lattice over a ghost tag associated with a datum. The operational semantics of expressions and statements of a programming language is extended to include a precondition on the possible ghost tags and a transition relation to determine possible effects on the ghost tags.

In their simplest form, ontic types can be thought of as *typestates* that determine valid sequences of operations that can be performed on an instance of a given type [16]. But ontic types extend *typestates* considerably in that values can have ontic tags along multiple dimensions, which can be interdependent in the sense that information can flow across different dimensions of ontic types.

Ontic type analysis accumulates all possible ghost tags at each reachable control location of the program, and checks that the collection of preconditions for these ghost tags holds at each control location of the program. In this way, ontic type analysis reduces to a terminating fixed-point iteration for well-founded programs. It is decidable if all the branching conditions and applicable lattice operators are decidable. Alternatively, ontic type analysis can be formulated as *Constrained Horn Clauses* (CHC). Ontic type analysis is much more precise than typical *dataflow* or *taint analysis* because it uses contextual information and the combination of decision procedures for solving constraints over the datatypes of the programming language under consideration. Ontic types are also expressive for *hyperproperties*, since the analysis of typical information flow properties is based on Reynold's *self-composition* trick for symbolically relating different executions of a program [16].

Ontic Typing for C. We now provide preliminary examples of how ontic typing is used to ensure some basic *memory safety invariants* for C-like programs. For ontic type analysis to be effective, the underlying program is assumed to terminate.

Take null dereferencing, for example. Its "inventor" calls it, in a typical case of British understatement, a *billion-dollar mistake* [42]. An ontic type for fixing null dereferencing simply keeps track of whether each pointer is *NULL* or *NONNULL*. The dereference precondition now checks that its argument pointer has *NONNULL* as its only possible ghost value. Since memory allocation is abstracted to succeed or fail non-deterministically, the postcondition on the ghost value of this operation is *{NULL, NONNULL}.* Dereferencing a pointer with this nondeterministic ghost value will fail, so a nullity check is required.

An ontic type is also used to express lifetime invariance between references and values. For example, a ghost variable *refcount* is used to keep track of the number of references pointing to this value. For example, an assignment like **p = v* has the ontic precondition *(p.refcount = 0),* which expresses that no pointer points to the value v. In this case, the postcondition is *(p'.refcount = p.refcount + 1)*. Furthermore, the deallocation of memory has the precondition *(p.refcount = 0),* which enforces that there are no dangling pointers. It should be clear that we are essentially encoding reference counting for C, with the notable difference that it is not used for dynamic runtime memory management, but for static checking of ontic invariants.

The ontic ownership type tracks whether a value is owned by the reference to which it is bound [21]. Ownership can also be transferred from one owner to another. When ownership is transferred, the ontic type of the variable is marked as stale, with the caveat that a stale value cannot be reused and does not need to be finalized at the end of the scope of the reference. Note that mutability interacts with ownership, since an immutable value can be transferred to a mutable variable by changing the mutability flag. Finally, references to owned values can be borrowed by other references as long as (1) the scope of the borrowing is contained within the scope of the owner, and (2) the mutable borrows are exclusive. Rust's borrowing system can, in principle, be coded as an ontic type, since for any statement at any control point, a transfer function can be defined to tell us which borrows it brings in or out of scope (cf. Rust RFC2094).

Ontic types keep also track of program counters to ensure that certain operations are performed in a timely manner. For example, such an ontic type might require that a file be closed at most *N* steps after its last access, which is expressed in Rust slang as the end of its lifetime.

An ontic type can distinguish between mutable and immutable references as in Rust. Based on this information, a variant of the reference-counting ontic type can, like Rust, ensure the memory safety property that there is at most one mutable reference pointing to any value at any time. Unlike Rust, however, one does not necessarily have to declare statically whether a reference is mutable or immutable, and a reference may be determined by an ontic type to be mutable for some parts of the code and immutable for other parts of the code. In these cases, there is an implicit flow of information from the mutable to the reference-counting ontic type.

Previous work has also attempted to provide C programmers with greater memory safety using *fat pointers [45,46],* which store some additional data besides just the address of the object being pointed to. *Fat* pointers are captured by corresponding ghost tags of an ontic type, thereby eliminating the runtime penalty of fat pointers. Similarly, an ontic type captures hardware-based capability-enhanced memory management, such as the *capability hardware enhanced RISC instructions* (CHERI) architecture [43,44,48], which replaces memory addresses with capabilities including permissions and tags. Bounds checking is performed on memory access, and capabilities can only be modified monotonically, which is now enforced by the ontic type instead of the underlying hardware.

Data races are prevented using ontic types by ensuring that globally shared variables (between threads) are protected by mutexes, that each write is preceded by a lock, and that mutexes are always unlocked in a timely manner after use. Arguably, ontic types for preventing data races are much simpler, more transparent, and more flexible than Rust's rather complicated ownership, borrow, and lifetime-based type system, along with smart pointers and unsafe library code for ensuring similar invariants.

These examples are intended to demonstrate the expressiveness, flexibility, compositionality, and open-endedness of ontic typing for making C a safer programming language. But there are many more useful ontic types such as input taint, initialized/uninitialized variables, dimensional analysis, freshness, security levels, authentication, encryption/decryption, authorization, declassification, and space/time complexity [21].

Approach. Ontic type analysis for C enforces essential memory safety properties that are sound and relatively complete with respect to a well-defined subset of Rust's intended memory safety invariants. However, the ontic C type checker, while clearly motivated by Rust's ownership, does not duplicate Rust's type system. Instead, it ensures a practically useful subset of memory safety invariants that do not require additional type annotations. At the very least, these invariants provide a safe abstraction for embedding unsafe code in Rust. There are three main challenges to supporting safer C:

1. Specify a minimal set of ontic memory safety types that is sound and relatively complete with respect to standard C semantics and Rust's intended set of memory safety invariants [15]. Soundness is only argued informally, given the lack of precise formal semantics for both C and Rust. Formal completeness would also require the exploration of dark and largely unexplored corners of C semantics.
2. Design an efficient and useful ontic type analyzer based on efficient symbolic execution combined with precise and lightning fast decision procedures for a combination of basic C types. Computational speedup and *modular ontic type analysis* can be achieved by *ontic summaries*, which are exact ontic pre- and postconditions of larger code fragments.
3. Design useful error messages and goal-directed interactions to identify and locate root causes of ontic type violations, as well as possible repair suggestions.

A variant of the ontic type checker for C may also be based on the LLVM intermediate representation, or alternatively on MLIR, since the semantics of these intermediate languages are easier to capture for the definition of succinct ontic types.

Benefits. The ontic type checker for safer C is an essential tool for making the programming world at least a memory-safe place. It provides the provably correct safe interface abstraction for safely integrating the unsafe Rust that results from translating that C code into Rust. Another use of ontic C type checking is to analyze critical legacy C code in the software supply chain, and it is also used to incrementally transform and repair memory-unsafe C programs so that they may be suitable for more automated translation into memory-safe Rust.

Compiling a version of the C program that explicitly maintains ghost tags is also helpful for debugging existing implementations, since ontic types are now interpreted as runtime monitors. These runtime checks are essential for detecting, for example, side-channel attacks such as physically manipulated random generators. Similarly, *fuzzers* are tuned to generate refined test input that matches the ontic type. By clearly separating the ontic roles of the data used, we also expect ontic types in C to lead to a more principled programming discipline.

References.

[1] White House Office of the National Cyber Director (ONCD), Back to the Building Blocks: A Path towards Secure and Measurable Software, February 2024. (https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf)

[2] See generally Morris worm - Wikipedia. See also National Vulnerability Database, Department of Commerce, National Institute of Standards and Technology, available here. See also BLASTPASS: NSO Group iPhone Zero-Click, Zero-Day Exploit Captured in the Wild , University of Toronto, Munk School of Global Affairs and Public Policy, September 2023, available here.

[3] *"Microsoft: 70 percent of all security bugs are memory safety issues". ZDNET. Retrieved 21 September 2022.*

[4] *"CVE-2014-0160". Common Vulnerabilities and Exposures. Mitre. Archived from the original on 24 January 2018. Retrieved 8 February 2018.*

[5] *Goodin, Dan (4 February 2020). "Serious flaw that lurked in Sudo for 9 years hands over root privileges". Ars Technica.*

[6] *"Fish in a Barrel". fishinabarrel.github.io. Retrieved 21 September 2022.*

[7] *Crichton, Will. "CS 242: Memory safety". stanford-cs242.github.io. Retrieved 22 September 2022.*

[8] Klabnik, Steve. *Memory Safety is a Red Herring, Dec 2023 (https://steveklabnik.com/writing/memory-safety-is-a-red-herring)*

[9] Milner, Robin, *A Theory of Type Polymorphism in Programming,* Journal of Computer and System Sciences, 17 (3): 348–375, 1987, doi:10.1016/0022-0000(78)90014-4.

[10] https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html

*[11] The Rustonomicum, The Dark Arts of Advanced and Unsafe Rust Programming. (*https://web.mit.edu/rustlang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/nomicon/meet-safe-and-unsafe.html)

[12] Alex Rebert, Christoph Kern, *Secure by Design: Google's Perspective on Memory Safety,* Mar, 2024 (https://research.google/pubs/secure-by-design-googles-perspective-on-memory-safety/)

[13] MITRE, CWE: Top 25 Most Dangerous Software Weaknesses, 2023. (https://cwe.mitre.org/top25/index.html)

[14] NSA, Software Memory Safety, Cybersecurity Information Sheet, Apr 2023. (https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF)

[15] Rust Compiler Error Index, Jun 2024, (https://doc.rust-lang.org/error_codes/error-index.html)

[16] *Strom, Robert E.; Yemini, Shaula. Typestate: a programming language concept for enhancing reliability.* IEEE Transactions on Software Engineering. 12. IEEE: 157–171, 1986.

[17] Wu, Yuchen, Hauck, Andrew. *How we built Pingora, the proxy that connects Cloudflare to the Internet. The Cloudfare Blog, Sept, 2022. (https://blog.cloudflare.com/how-we-built-pingora-the-proxy-that-connects-cloudflare-to-the-internet/#:~:text=Pingora%20is%20faster%20in%20production)*

[18] Mertens, Eric*. C2Rust, Aug, 2018. (https://galois.com/blog/2018/08/c2rust/)*

[19] Immunant Inc. *Introduction to C2Rust,* Aug, 2019. *(https://immunant.com/blog/2019/08/introduction-to-c2rust/)*

[20] Cliffle Blog, *The Typestate Pattern in Rust,* 2019-06-05. (*https://cliffle.com/blog/rust-typestate/)*

[21] Natarajan Shankar, Ontic Typing, Invited Talk, 2020.

[22] https://alumni.cornell.edu/cornellians/morris-worm/

[23] https://www.netscout.com/blog/asert/remembering-sql-slammer

[24] https://heartbleed.com/

[25] https://citizenlab.ca/2023/09/blastpass-nso-group-iphone-zero-click-zero-day-exploit-captured-in-the-wild/
[26] https://meltdownattack.com/

[27] NASA Engineering and Safety, *Technical Support to the National Highway Traffic Safety Administration (NHTSA) on the Reported Toyota Motor Corporation (TMC) Unintended Acceleration (UA) Investigation, Jan, 2011. (https://www.nhtsa.gov/sites/nhtsa.gov/files/nasa-ua_report.pdf)*

[28] https://cybersecurityventures.com/cybercrime-to-cost-the-world-8-trillion-annually-in-2023/

[29] CIS, NSA, FBI, ASD's ACS, CCCS, NCSC-UK, NCSC-NZ CERT-NZ. *The Case for Memory Safe Roadmaps. Why Both C-Suite Executives and Technical Experts Need to Take Memory Safe Coding Seriously. (https://media.defense.gov/2023/Dec/06/2003352724/-1/-1/0/THE-CASE-FOR-MEMORY-SAFE-ROADMAPS-TLP-CLEAR.PDF)*

[30] MSRC Team, Security Research & Define, *A proactive approach to more secure code,* July 2019. (https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/)

[31] Chromium, Memory Safety, Chromium Security, (https://www.chromium.org/Home/chromium-security/memory-safety/)

[32]. Diane, Hosfelt, Implications of Rewriting a Browser Component in Rust, Feb 2019. (https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/)

[33] Stone, Maddie, Google Project Zero, The More You Know, The More You Know You Don't Know, Apr, 2022. (https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html)

[34] Wang, Feng, et al. *Krust: A formal executable semantics of Rust.* In: 2018 International Symposium on Theoretical Aspects of Software Engineering (TASE). IEEE, 2018. pp. 44-51, 2018.

[35] Weiss, A., Gierczak, O., Patterson, D., & Ahmed, A. *Oxide: The essence of Rust*. arXiv preprint arXiv:1903.00982, 2019.

[36] Balasubramanian, A. et. Al., System programming in rust: Beyond Safety. In Proceedings of the 16th Workshop on Hot Topics in Operating Systems, pp. 156–161, 2017.

[37] Njor, Emil Jørgensen; Gustaffson, Hilmar. Static taint analysis in Rust. Master's Thesis, 2021.

[38] Emre, M., et al. Translating C to Safer Rust. Proceedings of the ACM on Programming Languages, 2021, 5. Jg., Nr. OOPSLA, pp. 1-29, 2021.

[39] Jung, R. et. Al.,. *Stacked borrows: An aliasing model for Rust*. Proc. ACM Program. Lang., 4(POPL), Dec, 2019.

[40] Hutt, T. Would Rust secure cURL?, Jan, 2021. (http://blog.timhutt.co.uk/curl-vulnerabilities-rust/)

[41] Jung, Ralf, Understanding and Evolving the Rust Programming Language, PhD thesis, Saarbrücken, Aug, 2020 (https://publikationen.sulb.uni-saarland.de/bitstream/20.500.11880/29647/2/thesis-screen.pdf)

[42] Hoare, Tony. *Null References: The Billion-Dollar Mistake.* Presentation at QCON, London, 2009. (https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/)

[43] Zaliva, Vadim, et al. *Formal Mechanised Semantics of CHERI C: Capabilities, Undefined Behaviour, and Provenance.* In: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, pp. 181-196, 2024.

[44] Woodruff, Jonathan, et al. *The CHERI capability model: Revisiting RISC in an age of risk.* ACM SIGARCH Computer Architecture News, 2014, 42. Jg., Nr. 3, pp. 457-468, 2014.

[45] T. Jim, J. G. Morrissett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, *Cyclone: A safe dialect of C*, in ATEC '02: Proceedings of the USENIX Annual Technical Conference, pp. 275–288, 2002.

[46] G. C. Necula, S. McPeak, and W. Weimer, CCured: *Type-safe retrofitting of legacy code*, in ACM SIGPLAN Notices, vol. 37, no. 1, pp. 128–139. 2002.

[47] Kan, S., Chen, Z., Sanán, D., & Liu, Y.. *Formally understanding Rust's ownership and borrowing system at the memory level.* Formal Methods in System Design, 1-37, 2024.

[48] Watson, R. N., Chisnall, D., Clarke, J., Davis, B., Filardo, N. W., Laurie, B., ... & Woodruff, J. *CHERI: Hardware-Enabled C/C++ Memory Protection at Scale.* IEEE Security & Privacy, 2024.

[49] Gasiba, T. E., & Amburi, S. *I Think This is the Beginning of a Beautiful Friendship-On the Rust Programming Language and Secure Software Development in the Industry*, CYBER, 2023.