Porcelain: A Semantic Framework for Representing and Analyzing Memory Safety Techniques

Pedro Henrique Boniatti Colle
phbcolle@inf.ufrgs.br
Universidade Federal do Rio Grande do Sul
Porto Alegre, Rio Grande do Sul, BR

Rodrigo Machado*
rodrigo.machado@inf.ufrgs.br
Universidade Federal do Rio Grande do Sul
Porto Alegre, Rio Grande do Sul, BR

ABSTRACT

At the start of the millennium, the Cyclone language was developed as a solution to the memory safety shortcomings of the C language. Since then, many programming languages have emerged with their own memory safety strategies. But how can we validate those strategies? This paper proposes Porcelain: a semantic framework for representing and analyzing memory safety techniques. Introducing its definitions, use cases, and limitations whilst giving an overview of the current and future work.

KEYWORDS: Memory Safety, Rust, Cyclone, Checked C

1 Introduction

The C programming language [8] is the foundation of systems programming. Up to the adoption of Rust in early 2024 [10], it was the (only) language of the Linux kernel. Despite this great importance, C is still prone to many types of memory safety violations, with very little guarantees from the compiler.

At the start of the millennium, Cyclone [6] started the trend of low-level programming languages with safe memory systems without the broad use of Garbage Collection (GC). This was followed by CCured [13], Rust [9] and more, each with its own memory safety solution.

Given a memory safety solution, it is important to formalize and prove its correctness, as projects such as Jung et al. [7] and Ho et al. [4] aimed for with Rust's borrow checker. With that, this work introduces Porcelain: a Semantic Framework for Representing and Analyzing Memory Safety Techniques. Its aim is to establish a back-end language (PCL_{back}) to which other languages could compile to, as a basis for proving memory safety techniques. For that, this paper also defines a borrow checking front-end language (PCL_{front}) defined via compilation to PCL_{back} as a use case for this system.

2 Memory Errors

Memory safety bugs, as it concerns low level languages, can be subdivided in 5 classes: Spacial, Temporal, Type, Initialization and Data-Race Safety [2, 14]. Even though memory safety bugs can be subdivided in many more ways [3, 18], each of those classes maps to a distinct mechanism for solving them. Spacial safety can be solved with dependent types [16] and runtime bounds checks [6]; Temporal safety can be solved with Garbage Collection [12], memory regions [17], a static alias analyzer [15], key-lock systems [13, 21], and more. Both Type and Initialization can be solved by imposing stricter constraints on declarations and type conversions. Data-Race safety is beyond the scope of this project.

3 Back-End Language

This work introduces PCL_{back}, a C-like core language designed to detect memory safety violations and to classify them. It is designed to be simple, for ease of developing proofs, but expressive enough to express common memory errors in C. Its syntax is defined as such:

$$
\begin{aligned}
\mathit{Locals} \ni l &::= l^m \mid l^p \\
\mathit{Value} \ni v &::= n \mid l \\
\mathit{BinOp} \ni \mathit{op} &::= + \mid - \mid * \mid < \mid > \mid = \mid \wedge \mid \vee \\
\mathit{Expression} \ni e &::= x \mid v \mid e \ \mathit{op} \ e \mid !e \mid f(\bar{e}) \mid {}^*e \mid \&x \\
&\quad \mid e; e \mid \{\, e \,\} \mid \texttt{let}\ x[n] \mid e := e \\
&\quad \mid \texttt{malloc}(e) \mid \texttt{free}(e, e) \\
&\quad \mid \texttt{if}(e)\ e\ \texttt{else}\ e \mid \texttt{while}(e)\ e \\
&\quad \mid \texttt{panic}\ \mathit{pcode} \\
\mathit{Function} \ni F &::= \texttt{let}\ f(\overline{x})\ e \mid \texttt{let}\ ()\ e \\
\mathit{Globals} \ni G &::= \texttt{global}\ x[n]\ G \mid F
\end{aligned}
$$

where x and f are meta variables that range over variable names and function names, respectively.

In the Syntax, `panic` *pcode* signals an early termination of the program in which the *pcode* indicates the type of memory safety violation that occurred. Those violations are: *Out of Bounds Read* (OBR) and *Out of Bounds Write* (OBW), for spatial safety; *Use After Free* (UAF), *Free Memory Not On Heap* (FMNOH), *Partial Free* (PF) and *Double Free* (DF), for temporal safety; *Uninitialized Access* (UA) for initialization safety; *NullPtr Dereference* (ND) is a domain-specific bug, because PCL_{back} is modeling C; *User Error* is when the user wants to terminate the execution, such as an ‘exit(1)‘ in C.

*Contributed as the advisor of the undergraduate thesis project associated with this paper.

3.1 Memory Model

The interplay between permanent allocations, ones on the *heap*, and scoped allocations, ones on the *stack*, is an important component to reproduce memory-related bugs. For the purpose of this modeling, the memory in PCL_{back} uses two disjunct environments, p, for modeling a memory *stack*, and m, for modeling a memory *heap*.

With that distinction, the locations l, elements of the language responsible for indexing the memory, hold a tag to specify which memory component is being indexed, as in l^p and l^m. These locations also hold metadata used for error detection. A location l has an index i, the position in the memory; a size s, the space indexable; an offset o, the element where additions and subtractions from the base are stored; and a unique key k, the checker value for temporal safety bugs.

3.1.1 Stack. The stack is a list of elements, which can be values $v \in \{l, n, \bot, -\}$ or control codes $c_{ctrl} \in \{\textbf{stack}, \textbf{func}\}$. The functions $\textbf{pop}_{stack}$ and $\textbf{pop}_{func}$ use these control codes to know up to which element to remove, simulating the drop of variables at the end of a scope.

The values $\bot$ and $-$ are used to detect initialization errors, meaning uninitialized and not allocated respectively, such as in [20]. Every element of the stack is paired with a lock, which is used for temporal errors. That results in the *stack* being defined as $p := [(v \mid c_{ctrl}, lock)]$. Notation similar to Haskell's [5] is used to manipulate the top of the stack, with $p(i)$ and $p[i \mapsto v]$ representing indexing and attribution in the structure.

3.1.2 Heap. The *heap* is similar to the *stack*, but without the control codes. Therefore a heap m is defined as $m := [(v, lock)]$. In PCL_{back}, the heap is manually deallocated with **free** and allocated with **malloc**. This structure can be conceptualized as an infinite list with all positions initialized as $-$, given that Porcelain does not cover memory limitation issues.

3.2 Operational Semantics

The language is defined by means of structural operational semantics, which specifies a one-step relation between configurations (states). One writes $\mathbb{F} \vdash \langle e, a, p, m\rangle \rightarrow \langle e', a', p', m'\rangle$ to describe the transition of state $\langle e, a, p, m\rangle$ towards state $\langle e', a', p', m'\rangle$ under a global function definition environment $\mathbb{F}$. Within the state, e is the program, a is the names environment, p is the stack, and m is the heap.

One example of the language's derivations is the access of a variable's value, as the following rules show.

$$\frac{a(x) = l^p\{i, k, o, s\} \quad \neg(0 \le o < s)}{\mathbb{F} \vdash \langle x, a, p, m\rangle \rightarrow \langle \textbf{panic OutOfBoundsRead}, a, p, m\rangle} \text{ (Var-obr)}$$

$$\frac{a(x) = l^p\{i, k, o, s\} \quad 0 \le o < s \quad p(i+o) = (v_p, k_p) \quad k \neq k_p}{\mathbb{F} \vdash \langle x, a, p, m\rangle \rightarrow \langle \textbf{panic UseAfterFree}, a, p, m\rangle} \text{ (Var-uaf)}$$

$$\frac{a(x) = l^p\{i, k, o, s\} \quad 0 \le o < s \quad p(i+o) = (v_p, k_p) \quad k = k_p \quad v_p = \bot}{\mathbb{F} \vdash \langle x, a, p, m\rangle \rightarrow \langle \textbf{panic UninitializedAccess}, a, p, m\rangle} \text{ (Var-ua)}$$

$$\frac{a(x) = l^p\{i, k, o, s\} \quad 0 \le o < s \quad p(i+o) = (v_p, k_p) \quad k = k_p \quad v_p \neq \bot}{\mathbb{F} \vdash \langle x, a, p, m\rangle \rightarrow \langle v_p, a, p, m\rangle} \text{ (Var)}$$

3.3 Fault Detection

In order to detect these kinds of faults, the metadata attached to the pointers needs to uphold certain invariants. If the offset o of a pointer is below 0 or equal to or greater than its size s ($0 > o \ge s$), then it results in a *Out of Bounds Read*, such as in (VAR-OBR), which is a Spacial Safety bug. If it is within bounds, but the key k in the pointer does not match the lock k' at the position ($k \neq k'$), then it results in a *Use After Free*, such as in (VAR-UAF), which is a Temporal Safety bug. With all that, if the value at position is $\bot$, then it results in a *Uninitialized Access*, which is an Initialization safety bug.

There are a few other faults in the language that were omitted for space, but all of them follow the same principle. This way of approaching faults not only expects certain invariants when they are present, but also defines those faults by the upholding of these invariants. A *Use After Free* fault is defined by an access where $k \neq k'$, and a case of *Use After Free* only indicates that $k \neq k'$, showing that **UseAfterFree** $\leftrightarrow$ $k \neq k'$. This concept can be extrapolated to the other memory safety violations, as shown in Table 1.

Type	Fault	Invariant
Spacial	OBR	$0 > o \ge s$
	OBW	$0 > o \ge s$
Temporal	UAF	$0 \le o < s\ k \neq k'$
	FMNOH	$l = l^p \vee i = 0$
	PF	$o \neq 0 \vee s \neq n$
	DF	$o = 0\ s = n\ k \neq k'$
Initialization	UA	$0 \le o < s\ k = k'\ v = \bot$
Extra	ND	$i = 0\ k = 0\ o = 0\ s = 0$

Table 1. Program Invariants

4 Use Case: Borrow Checker

One of the most researched solutions for memory bugs has been Rust's borrow checker [9]. It promises a zero-overhead Temporal Safety solution by constraining the ability to alias and adding code annotations to validate temporal memory

access, although properly validating its claims has been a challenge [4, 7].

In the case of PCL_{back}, there is an interesting bijection between the key locks used to check for temporal safety and the lifetimes used for computing the correctness of an access. One could postulate that, given a front-end language with a borrow checking system (PCL_{front}), it could be proven that no *Use After Free* errors would occur in the compiled PCL_{back} program, given a correctly typed program and a sound compilation.

5 Front-End Language

The front-end language, PCL_{front}, aims to model a borrow checking system. It differs from Rust's borrow checker due to different constraints and to ease compilation to PCL_{back}. This borrow checker directly uses linear types [1, 19] instead of Rust's *Copy-Clone* subtextual fling with the concept; and allows for multiple mutable references, called *alias* ($@\tau\,'a$).

The aim with this language is to specifically avoid the temporal memory fault of *Use After Free*. Therefore, when compiled to PCL_{back}, its objective is to never reach a derivation of `panic UseAfterFree`. The other temporal safety issues (FMNOH, DP, PF) are related to the corruption of the allocator structure and are not necessarily of concern of this borrow checker. For simplicity, its inclusion was reserved for future work. The syntax of the language follows:

$$
\begin{aligned}
\mathit{Types} \ni \tau &::= \texttt{int} \mid {}^*\tau \mid @\tau\,'a \mid (\bar{\tau}) \to \tau \\
\mathit{Locals} \ni l &::= l^m \mid l^p \\
\mathit{Value} \ni v &::= n \mid l \\
\mathit{BinOp} \ni op &::= + \mid - \mid * \mid < \mid > \mid = \mid \wedge \mid \vee \\
\mathit{Expression} \ni e &::= x \mid v \mid e\ op\ e \mid !e \mid e; e \mid \{e\} \mid {}^*e \mid \&x \\
&\quad \mid \texttt{alias}\ x \mid \texttt{alias}^*\ x \mid x \mathrel{+}= e \mid x \mathrel{-}= e \\
&\quad \mid \texttt{var}\ x : \tau := e \mid e_1 := e_2 \mid x_1 :=: x_2 \mid x_1 :=:^* x_2 \\
&\quad \mid \texttt{if}(e)\ \{e\}\ \texttt{else}\ \{e\} \mid \texttt{let}\ x : \tau = f(\bar{e}) \\
&\quad \mid \texttt{let}\ x : \tau = \texttt{new}(e) \mid \texttt{delete}(x, e) \\
&\quad \mid \texttt{stop} \mid \texttt{nullprt<}\tau\texttt{>} \mid \texttt{nullalias<}\tau\texttt{>} \\
\mathit{Function} \ni F &::= \texttt{fn}\ f(\overline{x : \tau}) \to \tau\ \{e\}\ F \mid \textbf{let}\ ()\ e
\end{aligned}
$$

where $'a$, x and f are meta variables ranging over regions in the program, variable and function names, respectively. Tuple and recursive types, along with while loops and global declarations, were omitted for simplicity and time constraints.

5.1 Type checking

The borrow checker implemented is based on the Polonius proposal [11, 15]. In it, each borrow has an associated region and each region has a set of associated loans according to the control flow graph (CFG) of the program. Invalidating the terms of any loan in that set invalidates the access of an alias associated with that region.

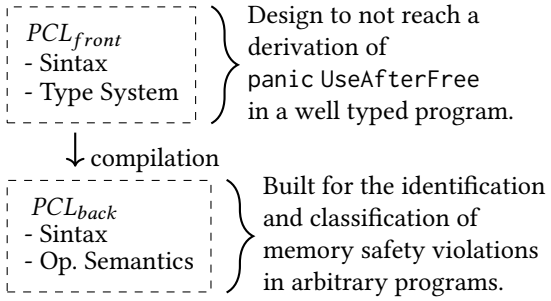
In PCL_{front}, each alias type ($@\tau\,'a$) has a region associated $'a$. During the type checking, whenever an alias is accessed, it is validated that its region is alive (has a valid set of loans). Accesses through the path of a loan invalidate it, which can happen in at many points of the evaluation tree. This requires that a node in the derivation tree alter a state which will be used for the sequent assessment of other nodes. This imposes an evaluation order of depth first, in the form $\Delta \vdash e : \tau \mid \Delta'$, with Δ representing the environments used for type checking.

5.2 Compilation

The compilation from PCL_{front} to PCL_{back} is very close to a simple process of type erasure. A few operands have to compile to explicitly return an $Unit$ type, defined in PCL_{front} as 1. The most complicated transformations, the ones for :=: and :=:*, only required the creation of variables with unique names (s_{uniq}) and to execute a swap, as: $x_1 :=: x_2 \mapsto$ `let` $s_{uniq}[1]; s_{uniq} := x_1; x_1 := x_2; x_2 := s_{uniq}$.

6 Conclusion/Future Work

The Porcelain framework aims to allow the modeling of memory safety techniques and can be visualized as:



Safety (sketch): If $p \in PCL_{front}$ and p is well-typed, then the evaluation of $compile(p) \in PCL_{back}$ does not trigger specific memory errors.

At this moment, PCL_{back} has a stable, complete formal specification and prototype implementation in Haskell (parser and evaluator). PCL_{front} has a first version of its syntax, type system and compilation semantics, and it is currently being validated. The next step is to express and prove the safety of the compilation of correct code in PCL_{front} by means of standard techniques such as structural induction on the type derivation.

REFERENCES

[1] Samson Abramsky. 1993. Computational interpretations of linear logic. *Theor. Comput. Sci.* 111, 1–2 (April 1993), 3–57. doi:10.1016/0304-3975(93)90181-R

[2] Security Research Apple. 2022. *Towards the next generation of XNU memory safety: kalloc_type.* Apple. Retrieved June 14, 2025

from https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety/

[3] Content Team CWE. 2023. *CWE CATEGORY: Comprehensive Categorization: Memory Safety*. MITRE. Retrieved June 14, 2025 from https://cwe.mitre.org/data/definitions/1399.html

[4] Son Ho, Aymeric Fromherz, and Jonathan Protzenko. 2024. Sound Borrow-Checking for Rust via Symbolic Semantics. *Proc. ACM Program. Lang.* 8, ICFP, Article 251 (Aug. 2024), 29 pages. doi:10.1145/3674640

[5] Graham Hutton. 2016. *Programming in Haskell* (2nd ed.). Cambridge University Press, USA.

[6] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATEC '02)*. USENIX Association, USA, 275–288.

[7] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. doi:10.1145/3158154

[8] Brian W. Kernighan and Dennis M. Ritchie. 1988. *The C Programming Language* (2nd ed.). Prentice Hall Professional Technical Reference.

[9] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language*. No Starch Press, USA.

[10] Hongyu Li, Liwei Guo, Yexuan Yang, Shangguang Wang, and Mengwei Xu. 2024. An empirical study of rust-for-Linux: the success, dissatisfaction, and compromise. In *Proceedings of the 2024 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC'24)*. USENIX Association, USA, Article 27, 19 pages.

[11] Niko Matsakis. 2018. An alias-based formulation of the borrow checker. https://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/

[12] John McCarthy. 1960. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM* 3, 4 (April 1960), 184–195. doi:10.1145/367177.367199

[13] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.* 27, 3 (May 2005), 477–526. doi:10.1145/1065887.1065892

[14] Alex Rebert and Christoph Kern. 2024. *Secure by Design: Google's Perspective on Memory Safety*. Google Security Engineering Technical Report. Google.

[15] Amanda Stjerna. 2020. *Modelling Rust's Reference Ownership Analysis Declaratively in Datalog*. Master's thesis. Uppsala University, Department of Information Technology.

[16] David Tarditi, Archibald Samuel Elliott, Andrew Ruef, and Michael Hicks. 2018. Checked C: Making C Safe by Extension. In *IEEE Cybersecurity Development Conference 2018 (SecDev)*. IEEE, 53–60. https://www.microsoft.com/en-us/research/publication/checkedc-making-c-safe-by-extension/

[17] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. 2004. A Retrospective on Region-Based Memory Management. *Higher-Order and Symbolic Computation* 17 (09 2004), 245–265. doi:10.1023/B:LISP.0000029446.78563.a4

[18] Katrina Tsipenyuk, Brian Chess, and Gary McGraw. 2005. Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors. *IEEE Security and Privacy* 3, 6 (Nov. 2005), 81–84.

[19] Philip Wadler. 1990. Linear Types can Change the World!. In *Programming Concepts and Methods*. https://api.semanticscholar.org/CorpusID:58535510

[20] Nienke Wessel. 2019. *The Semantics of Ownership and Borrowing in the Rust Programming Language*. Bachelor's Thesis. Radboud University.

[21] Jie Zhou, John Criswell, and Michael Hicks. 2023. Fat Pointers for Temporal Memory Safety of C. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 86 (April 2023), 32 pages. doi:10.1145/3586038