Cedar: Declarative Data Layouts for C

K. Chinchilla Flores 1526925
Supervisor: Dr. Christine Rizkallah

2025-11-03

Declaration of Authorship

I certify that

- This thesis does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain material previously published or written by another person where due reference is not made in the text.
- No clearance was necessary from the University Ethics committee.
- The thesis is 11145 words in length (excluding text in images, table, bibliography, and appendices).

Abstract

When programmers write low-level systems code, such as device drivers or network protocol implementations, they need to be able to control exactly how data structures are stored in memory. In the C programming language, these details are usually implied by the order of fields, alignment directives, or pragmas that are specific to the compiler. These kind of tools are powerful, but can also be error-prone and layout problems are still a big cause of systems crashes and security issues.

Cedar is a declarative programming language that addresses this problem by making memory layouts clear and easy to reason about in the C programming environment. Cedar builds on previous work, most notably Dargent, a verified data description language built into Cogent. It aims to follow the same declarative principles while being usable directly in C development. It treats data layouts as first-class constructs, meaning programmers can describe structure and placement at the byte level while still being able to use existing compilers.

Cedar provides a practical basis for reliable data-layout specification by making intent clear and easy to be checked statistically. This does not require a whole new programming environment, and shows that language design alone may provide many benefits learned from formal methods, such as clarity and predictability. Cedar is a lightweight expressive framework that increases the transparency and reliability of C data representations. It fills the gap to increase trustworthiness in low-level systems programming without leaving a familiar ecosystem.

Acknowledgements

I would like to express my deepest gratitude to **Dr. Christine Rizkallah**, my supervisor, for her invaluable guidance, patience, and encouragement throughout this project. Her insight and feedback shaped both the technical direction and clarity of this thesis.

I would also like to thank **George Anderson**, whose original prototype of Cedar's concept formed the foundation on which this work was built. His prior contributions mae this research possible.

Finally, I extend my thanks to **Vincent Jackson** for his thoughtful and patient discussions and advice during the development of this work

Contents

Declaration of Authorship **2**

Abstract **3**

Acknowledgements **4**

1 Introduction **8**

1.1 Motivation 8
1.2 Problem Context 9
1.3 Research Aim and Question 10
1.4 Approach and Methodology 11
1.5 Contributions 12
1.6 Thesis Structure 12

2 Background and Related Work **13**

2.1 Overview 13
2.2 The C Memory Model and Layout Challenges 14
2.3 Verified Compilers 16
2.3.1 Verified Compilation and the C Memory Model 16
2.3.2 From Verified Compilers to Verfied Systems Languages 17
2.3.3 Declarative Layouts in Cogent: The Dargent Extension 17
2.3.4 Extending the Lineage 18
2.4 Declarative and Domain-Specific Layout Languages . . 19
2.4.1 Academic Layout Description Languages 19
2.4.2 Industrial Serialization Frameworks 19
2.4.3 Comparing Declarative Layout Languages . . . 20
2.5 Runtime Safety and Analysis Tools 21
2.5.1 Dynamic Analysis Tools 21
2.5.2 Static Analysis and Type-Based Approaches . . 21
2.5.3 Safer Systems Languages 22
2.5.4 Hardware-Assisted Memory Safety 22
2.5.5 Positioning Cedar 23
2.6 Summary and Research Gap 23

3 Methodology and Implementation **24**

3.1 Foundations: CompCert and Dargent 25
3.1.1 CompCert verified C semantics as a foundation 25

3.1.2 Dargent: declarative layouts and type-layout separation 25
3.1.3 Bridging the Two influences 26
3.2 Language Overview 26
3.2.1 C types 26
3.2.2 Supported Constructs 27
3.2.3 Primitive and Derived Types 27
3.2.4 Layout Representation 28
3.2.5 Scope and Limitations 29
3.3 Cedar Language Design and Core Features 29
3.3.1 Core Syntax 29
3.3.2 Layout-Type Matching 30
3.3.3 Well-formedness 31
3.3.4 Design Considerations 31
3.3.5 Supported Directives Summary 32
3.4 Compiler Architecture 32
3.4.1 Parsing and Resolution 32
3.4.2 Static Checking 33
3.5 Code Generation 34
3.5.1 Code Generation Example 34
3.5.2 In-memory representation 35
3.6 Flattening of nested records and naming 35
3.6.1 Scalars and Arrays 36
3.6.2 Emdianness Policy 36
3.6.3 Byte-part helpers and masking discipline 38
3.6.4 Example: offsets and word mapping 38
3.6.5 Compilation Targets 38
3.6.6 Current codegen limitations 38
3.7 Design Rationale 39
3.8 Achievements and Scope 39
3.8.1 Implemented Features 39
3.8.2 Implementation Coverage 40
3.8.3 Practical Outcome 40
3.9 Summary 41

4 Evaluation and Results **41**
4.1 Experimental Procedure 42
4.2 Nested Record Layouts 43
4.2.1 Discussion 45
4.3 Representation of the 64-bit ELF Header 45

4.3.1 Discussion 48
4.4 Discussion of Findings and Limitations 48
4.5 Summary 50

5 Future Work 51
5.1 Language and Compiler Features 51
5.2 Towards a Verified System 52

6 Conclusion 53

Appendix 58

List of Tables

1 Implementation coverage summary 40

List of Figures

1 Example of implicit padding in a C structure, illustrating how compiler alignment rules insert invisible bytes between fields. Even simple definitions can produce unexpected layouts, motivating the need for declarative, checkable layout descriptions. 9

2 The Cogent refinement framework (adapted from [28]). Each box represents an embedding of the same Cogent program within Isabelle/HOL, and arrows denote refinement proofs that establish semantic equivalence between successive representations. This verified foundation underpins later extensions such as Dargent, which introduces declarative data-layout descriptions within the Cogent toolchain. 18

3 Cedar's type universe, aligned with CompCert C. . . 26

4 Surface syntax of Cedar layouts. Users describe field order, byte offsets, and endianness declaratively. 28

5 * 28

6 Core syntax of Cedar layouts after parsing and offset resolution. Greyed forms are shown for context only. . 30

7 Layout–type matching rules in Cedar. Each layout ℓ corresponds to a well-typed region of memory for type τ. 30

8 Well-formed layouts in Cedar. A layout is well-formed if its fields occupy disjoint bit ranges, variant tags are unique, and all sublayouts are themselves well-formed. 31

9 The Cedar compiler processing pipeline, adapted from Anderson's original proposal [2]. Phase 1 performs lexing, parsing, and semantic analysis of the input source code, while Phase 2 performs code generation to produce C output. 33

1 Introduction

Programming systems in C are ever an important part of computer infrastructure [4, 7, 36]. Operating systems, device drivers, and performance-critical software all depend on the C language's ability to access memory directly [38, 24]. Yet the same freedom that makes code written in C efficient also makes it fragile: tiny inconsistencies in layout can lead to huge runtime errors [24, 29, 12]. Layout-related faults, incorrect padding, misaligned fields, or unintended aliasing, manifest as memory corruption and undefined behavior. These faults are notoriously difficult to detect, especially when they happen in hand-crafted structures where delicate layouts must match exactly for hardware or protocol standards.

1.1 Motivation

Layout defects are widespread,even though there has been significant work on tools, dynamic sanitizers, and formal verification efforts. This is because errors tend to occur *below* the level of most abstractions. Tools such as **Valgrind Memcheck** [36] and AddressSanitizer [35] are tools that designed to find uninitialized values, and invalid accesses. However, these tools only detect issues *after* they occur. At the same time, real-world studies on real systems demonstrate that **memory layout choices have significant effects both performance and reliability**, impacting cache locality, concurrency, and even power consumption [15, 44].

Verified programming languages like **Cogent** [9] have shown that it is possible for a functional, type-safe systems code to be compiled to efficient C with formal correctness guarantees. However, Cogent and similar languages exists in their own ecosystems. The **Dargent** framework within Cogent partly addresses this issue by allowing the specification of *data layouts* alongside high-level data types, generating C structs and accessors that follow those specifications.

Despite its promise, Dargent exposes an inherent challenge: layouts must be low-level enough to manage bits and bytes, yet compositional enough to reason and verify. Its design is also intimately tied to Cogent's compiler, which makes reuse outside that context difficult. These limitations drive the development of a standalone, declarative language for describing and generating correct low-level layouts: A language that can serve both as a research vehicle for reasoning about

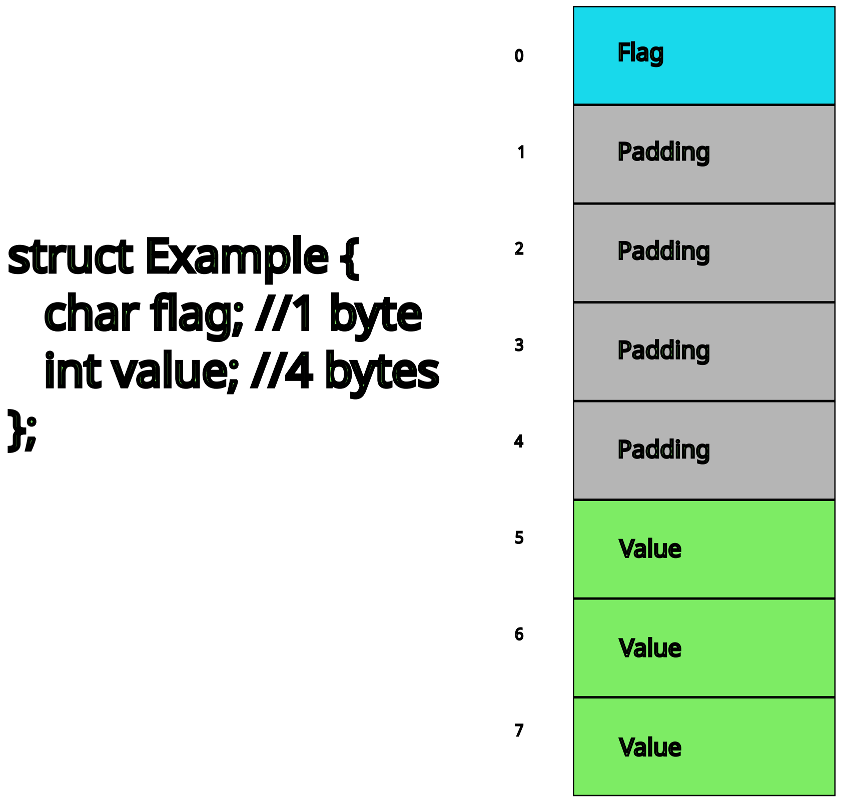

Figure 1: Example of implicit padding in a C structure, illustrating how compiler alignment rules insert invisible bytes between fields. Even simple definitions can produce unexpected layouts, motivating the need for declarative, checkable layout descriptions.

memory representations and as a practical tool for C programmers.

This paper addresses the gap between *abstract correctness* and *representation*. Cedar is a declarative language for specifying data layout as a first-class concern, that allows programmers to reason about byte-accurate specifications.

1.2 Problem Context

The flexibility of C's memory model, while powerful, complicates formal reasoning. *Type-punning*, *bit-fields*, and *packed structs* are common ways for C programmers to control binary representations, but the standard gives only practical guarantees about how these constructs behave. Compiler optimizations may reorder data, introduce padding, or interpret aliasing differently among architectures [18].

This design creates a big trust gap between language semantics and hardware behavior. Layout mismatches can manifest in several ways: **Errors in padding and alignment**, which happens when the compiler adds bytes that the programmer did not intend, and may break formats or protocol expectations. **Truncation or overlap of Bit-fields**, which causes packed data to become corrupted, often silently. **Endianness and offset miscalculation**, especially in cross-platform programs. **Aliasing violations**, where multiple pointer types reference the same region with different assumptions about its state.

These problems exist even when the layout is properly defined. It is difficult hard to find these issues because they tend to rarely crash programs directly; instead, they cause hidden problems that show up under specific compiler versions or architectures. As studies of memory errors in production software show, **layout-level faults are among the most persistent categories of low-level defects** [45].

Cedar addresses this category of issues by offering a distinct, declarative layer capable of specifying how a value is represented in memory, irrespective of its logical structure. Cedar externalizes this layout logic as a *layout description language*. Then, the compiler then turns this specification into accessors and struct declarations that can be checked or verified.

1.3 Research Aim and Question

This thesis examines the capacity of a compact, declarative language to describe memory structures while maintaining integrity of correctness reasoning. The project contributes with a programming language, **Cedar**, initially proposed by Anderson [2], which handles data layouts as a first-class concern. Cedar allows programmers to describe how values of structured types occupy memory, separating layout from logical structure.

The central **research question** guiding this work is:

> **How can we design a data layout language that is both practical for systems programming, and sound in its treatment of C memory models?**

Addressing this question involves both theoretical and practical components:

1. **Theoretical Model and Language Design**
    - Add support for nested layouts and arrays to Cedar's syntax and internal representations.
    - Formalize layout semantics in terms of offset-size pairs and compositional alignment rules.
    - Draw parallels with declarative frameworks, to inform trade-offs with techniques such as serialization.
2. **Implementation and Code Generation**
    - Implement a compiler in Haskell, producing C headers and definitions from Cedar programs.
    - Focus on generating *cohesive* code.
    - Ensure the pipeline remains modular so that each stage could, in the future, could be linked to a verified backend (e.g., CompCert).
3. **Evaluation Examples**
    - Reproduce examples from Dargent.
    - Evaluate in terms of *expressivity* (what can be represented), *correctness* (generated C compiles and behaves as Intended) and *usability* (readability, simplicity, and correspondence with C conventions).

The thesis does not aim to produce a fully verified compiler, but to establish the *architecture* and *design principles* that would allow one to be verified later. In this sense, Cedar serves both as a language design experiment, and a bridge between research prototypes and industrial serialization tools.

1.4 Approach and Methodology

The research follows a *design-and-evaluation* methodology typical of programming-language research [27]. The work proceeds in three interlocking stages:

1. **Language Design and Formalization**: analyzing the previous Cedar proposal and revising its intermediate representations (L $\to$ LR $\to$ CR) to better capture nested records, arrays, and bit-fields.
2. **Implementation and Code Generation**: enabling the Cedar compiler to emit cohesive, self-contained C code, and test this

code generation on examples so that it may be directly compared to similar tools or alternative approaches.

3. **Assessments of Examples**: running the Cedar compiler on examples to evaluate how expressive, correct, and consistent the generated code is.

During these stages, the design was informed by principles from declarative programming, as it is implemented in a functional style using Haskell.

1.5 Contributions

This project makes several contributions to the study of data-layout languages:

- **Cedar Language Design**: It establishes an expanded declarative syntax for memory layouts in C, The design isolates layout description from computation, which lets programmers think about representation by itself.
- **Implementation Contribution**: It provides a Haskell implementation of Cedar, that includes semantics and code generation. The compiler shows how that declarative layouts can be translated into valid C code without the need to write special alignment code.
- **A Compiler**: A working Cedar compiler that works, implemented in Haskell. It includes parsing, type checking layout-matching, and code generation.
- **Evaluation via Examples**: Assessing Cedar on two representative layout examples.
- **Discussion of Design Trade-offs and Future Verification**: Establishes the distinction among declarative specification, compiler automation, and machine-checkable proof of correctness.

These results outcomes turn Cedar from a concept into a more complete research language.

1.6 Thesis Structure

This paper is is composed of the following chapters:

- **Chapter 2** reviews background material on C memory layouts, verified compilation, the Cogent and Dargent frameworks, and the origins of Cedar.
- **Chapter 3** describes the research methodology and language design, detailing Cedar's intermediate representations and translation pipeline.
- **Chapter 4** presents the implementation and evaluation, including examples.
- **Chapter 5** discusses design implications, limitations, and directions for future verification and usability improvements.

2 Background and Related Work

Low-level systems software depends on correct algorithms but also on how data is represented in memory. Choices about field order, alignment, and bit layout can affect performance, portability, and correctness. This chapter surveys research and industrial work that informs Cedar's design. It begins with an overview of the problem space, followed by a discussion of C's memory model and layout hazards. Then we talk talk about verified compilers, and industrial tools that follow a similar concept, as well as tools designed for runtime analysis. The last subsection summarizes these findings and identifies the gap that this project aims to fill.

2.1 Overview

One of the most important aspects of systems programming is the programmer's ability to regulate how data is stored in memory precisely. Programming languages such as C give programmers direct access to memory through pointers and explicit layout constructs. These techniques make it possible to write high-performance code, as well as code that needs to sit close to hardware [44, 7]. This flexibility, however, introduces some risks: implicit padding, inconsistent alignment across architectures, and aliasing behaviors that are not completely defined by the C standard [37, 13]. Even if the program logic is correct, representation errors can lead to subtle bugs, security vulnerabilities, or slower cache performance [38, 29, 15].

Research that deals with these challenges has gone in several directions. **Verified compilation** projects such as CompCert and Cogent look to ensure correctness of compiler translations via automated machine-generated proofs. However, they typically assume that input programs already have well-defined memory layouts. **Declarative schema languages** such as PADS, DataScript, and industrial tools like FlatBuffers or Cap'n Proto make data representation explicit for serialization, though they operate through generated APIs rather than arbitrary in-place C structures [17, 3, 21, 32] Meanwhile, **runtime tools** such as Valgrind and AddressSanitizer detect invalid memory accesses dynamically, identifying many errors only after execution [36, 35].

Cedar builds insights from these areas to explore a **declarative, C-oriented approach** to describing layouts directly. The remainder of the chapter sets out the background required to understand that design: the characteristics of C's memory model, the verified-compiler lineage, declarative layout framework, and complementary runtime safety tools.

2.2 The C Memory Model and Layout Challenges

Systems programming depends on the programmer being able to change data structures directly in memory. C makes this trivial because it contains tools such as pointers, address arithmetic, and low-level layout control, which allow programmers to express data representations to map efficiently to hardware [33, 44]. This flexibility allows precise control and high performance, but it also puts the burden of correctness to the programmer. C itself does not require uniform object representations. Different compilers may vary in how they align, pad, and order fields, and the standard intentionally does not specify these details. [37, 13].

In reality, these uncertainties cause problems such as implicit padding, misaligned fields, and bit fields that overlap. All of these concerns can produce silent data corruption or access errors. For example, structure layouts may vary between architectures or compilers even when their source definitions are identical [14, 11]. The C99 and C11 standards allow compilers to insert padding bytes for alignment, but do not specify how or where these bytes should go in the memory model. This results in uncertainty that complicates binary interoperability, particularly for programs that need to communicate through shared memory, files, or hardware registers.

The effects of representation choices go beyond just being correct. Data layout has a big effect on **cache locality** and, by extension, overall performance [15, 19, 26]. Reorganizing fields or changing arrays-of-structures into structures-of-arrays can significantly improve performance, demonstrating that representation is both a semantic and performance concern [31].

To control layout, programmers use mechanisms such as `struct` for field ordering bit-fields and `union`, often combined with compiler-specific directives like `#pragma pack` or `__attribute__((packed))` [10]. These features are essential when implementing device drivers, network stacks, or file-system components that must match predefined binary formats [8, 39]. Yet their semantics are fragile: even small changes in type definitions or compiler options can alter offsets or alignment unexpectedly. Because the correctness of these constructs is not checked statically, developers resort to **trial-and-error inspection**—printing offsets or using debuggers to verify field positions—rather than relying on language guarantees.

The consequences of layout mistakes are well documented. Misaligned accesses can crash embedded firmware; incorrect field boundaries in protocol headers can expose vulnerabilities exploitable through malformed input [38, 38]; and inconsistent packing across architectures breaks cross-platform file or network formats. Each example reflects a deeper issue: the lack of a formal abstraction boundary between a type's *semantic structure* and its *physical representation* in memory. Without such a boundary, reasoning about correctness or portability becomes extremely difficult, both for programmers and for automated analysis tools.

C's weak memory model compounds this difficulty. The standard defines memory in terms of abstract "objects" and "effective types", but optimizations rely on assumptions such as the **strict aliasing rule** which are frequently violated in low-level code [13, 37]. These rules can cause apparently safe pointer casts to yield undefined behavior, particularly in code that manipulates raw binary data. Formal memory models developed for verified compilers, such as CompCert's block-and-offset model, explicitly exclude such cases from their semantics [23]. As a result, even when C code is verified at a higher level, layout mismatches or aliasing violations remain outside the verified envelope.

These challenges highlight the need for **explicit, verifiable layout specifications**. A language mechanism that clearly separates logical types from their physical representation could prevent many classes of low-level bugs, improve portability, and enable reasoning about performance effects. Later sections of this chapter review verified and declarative approaches that aim to provide such abstractions.

2.3 Verified Compilers

Formal verification has become a central technique for establishing trust in low-level software systems. Rather than detecting errors through testing, verified compilers and proof assistants attempt to **prove semantic preservation**: that the compiled program behaves exactly as specified by its source semantics [23]. This approach has produced verified toolchains such as **CompCert**, and, more recently, high-level verified systems languages such as **Cogent** [1].

2.3.1 Verified Compilation and the C Memory Model

CompCert is a formally verified C to assembly whose proof, mechanized in Coq, ensures functional equivalence between source and target programs under a rigorously defined model [23, 22, 41]. CompCert's *block-and-offset* model treats memory as a collection of abstract objects rather than a flat byte array, allowing it to reason about pointers and aliasing while avoiding undefined behavior. The model deliberately abstracts away physical layout: byte order, field alignment, and padding are not represented. As a consequence, CompCert's correctness theorem applies only to programs that never depend on the concrete arrangement of bytes in memory [41]. Programs that perform bit-level reinterpretation of data or depend on packed representations fall outside its verified envelope.

This abstraction limits the reach of formal verification for systems code that must match **external binary specifications**—for example, device drivers or network protocol stacks that manipulate precise bit layouts. Verified compilation ensures that a compiler will not introduce new errors, but it does not guarantee that a given structure definition correctly corresponds to its hardware or wire-format specification. Addressing this gap requires reasoning not just about program behavior, but about the **correspondence between logical types and concrete memory representations.**

2.3.2 From Verified Compilers to Verfied Systems Languages

The **Cogent** language was developed to make verification of systems components practical by constraining features and integrating compilation with theorem-proving [1]. Cogent is a **total, strongly typed, purely functional language** that compiles to C and simultaneously generates a formal model in Isabelle/HOL. This dual output allows refinement proofs showing that the generated C implementation faithfully realizes the semantics of the Cogent program. Cogent's design enforces properties such as **linearity of state** and **absence of undefined behavior**, enabling verified systems components like file systems and kernel modules to be produced.

However, Cogent's abstract data types still required manual, adhoc mappings to the concrete C layouts they referenced. The need to express these mappings precisely led to the development of **Dargent**.

2.3.3 Declarative Layouts in Cogent: The Dargent Extension

Dargent extends Cogent with a **declarative language for specifying data layouts**. Introduced as "Bringing Effortless Refinement of Data Layouts to Cogent" [28], it allows programmers to define how algebraic data types should be represented at the bit level within memory, elimination much of the manual marshalling previously required. Layouts are expressed separately from type definitions and are checked for **well-formedness**, ensuring that fields neither overlap nor violate alignment constraints.

In its later formalization [9], Dargent was fully integrated into Cogent's type system and semantics. The authors proved that layout accessors and updates generated by the compiler preserve the abstract semantics of the data type, bridging the gap between logical specification and binary representation. Dargent's key innovation is the explicit association of a *layout descriptor* with each type, allowing verified reasoning about both functional behavior and physical placement in memory.

Together, Cogent and Dargent demonstrate that it is possible to achieve **verified data-layout correctness** within a restricted functional environment. Yet the restrictions that make this verification tractable—totality, purity, and the absence of unrestricted mutation—also limit applicability to general-purpose systems programming. Dargent's layout declaration are tied to Cogent's type system and proof infrastructure; they cannot be directly used within arbitrary C codebases.

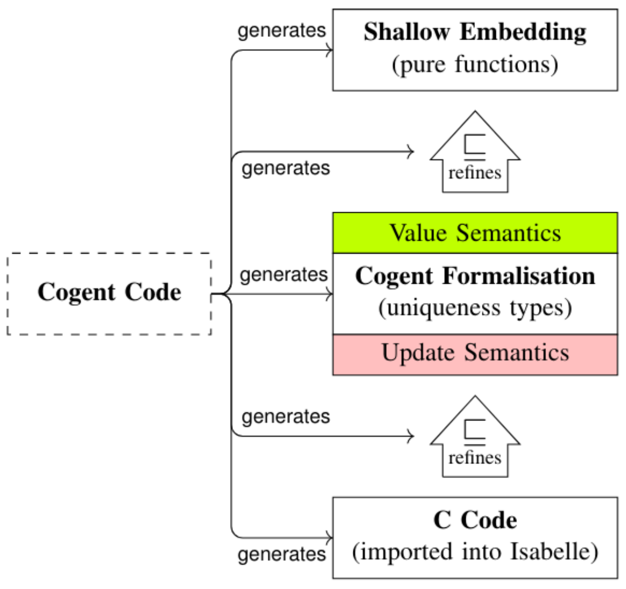

Figure 2: The Cogent refinement framework (adapted from [28]). Each box represents an embedding of the same Cogent program within Isabelle/HOL, and arrows denote refinement proofs that establish semantic equivalence between successive representations. This verified foundation underpins later extensions such as Dargent, which introduces declarative data-layout descriptions within the Cogent toolchain.

2.3.4 Extending the Lineage

Earlier foundational work by **Tuch** (2008) formalized C's pointer and heap semantics and Isabelle, laying the groundwork for verified reasoning about memory at the byte level [42]. More recently, efforts such as a **Cornucopia** for CHERI architectures [16] push forward verified compilation toward more realistic representations of machine memory.

Cedar builds directly on this insight. It seeks to extend the extend the idea of **type-layout separation** and layout well-formedness checking to the broader C ecosystem, without imposing Cogent's functional restriction or proof obligations. In this sense, Cedar generalized the principles pioneered by Dargent, bringing declarative layout specifications to conventional systems programming.

2.4 Declarative and Domain-Specific Layout Languages

While verified compilers focus on proving that compilers focus on proving that compilation preserves meaning, a complementary tradition in language design makes the data *representation* explicit through **declarative schema or layout languages**. These languages allow programmers to describe binary formats at a high level, from which parsers, serializers, and accessors, can be generated automatically. Their goal is not proof of semantic equivalence, but **automation, maintainability, and safety** in specifying how structured data maps to bytes.

2.4.1 Academic Layout Description Languages

Early research systems such **PADS** and **DataScript** pioneered declarative specifications of binary data [17, 3]. PADS introduced a system for ad-hoc data formats, enabling automatic parser generation and providing basic static checks such as field coverage and alignment consistency. **DataScript** [3], developed at Stanford, treated binary layouts as first-class constructs: programmers defined bit-level field structures and constraints, and the compiler generated Java classes to read and write them. These languages were influential in showing that low-level binary parsing could be expressed declaratively rather than through manual bit-twiddling code.

2.4.2 Industrial Serialization Frameworks

Many of these academic ideas re-emerged in modern industrial frameworks designed for cross-platform serialization and inter-process communication. Tools such as **FlatBuffers** [21] and **Cap'n Proto** [32] use **interface definition languages** (IDLs) to describe data schemas from which efficient, type-safe code is generated automatically. These frameworks guarantee binary compatibility across architectures and languages, reducing the likelihood of layout mismatches between communicating components.

The **Cap'n Proto** and **FlatBuffers** tools focus on zero-copy access: data can be read directly from the serialized buffer without immediate unmarshalling. This design aligns with the performance goals of systems programming but constrains data access to API-generated methods rather than arbitrary pointer arithmetic. A recent survey by **Vioti and Kinderkhedia (2022)** catalogues these binary specifications and highlights trade-offs between expressiveness, schema evolution, and safety [43].

Recent work has begun on examining these tools from a **security pers**. **Lilly Sell's (2024)** analyzes the schema semantics of FlatBuffers and its implementation to identify classes of layout vulnerabilities and unsafe pointer usage [34]. The study emphasizes that, despite their strong type systems, industrial IDLs provide no formal guarantees about in-memory safety or field alignment once data is accessed through unsafe code. These findings underline that even modern frameworks remain vulnerable when low-level layout details escape the boundaries of their generated APIs.

2.4.3 Comparing Declarative Layout Languages

Declarative layout frameworks share several design principles:

1. they separate *logical structure* from *physical representation*,
2. they generate accessors and parsers automatically; and
3. they provide at least partial checks for field consistency or alignment.

However, they typically operate in **controlled environments**, either within managed runtimes or through generated API layers, and are not designed for **arbitrary pointer-level operations** common in systems code. Their safety guarantees therefore stop at the interface boundary: once raw memory is accessed directly, correctness again depends on the programmer.

Cedar builds on this lineage by re-introducing declarative layout control *within C* itself. It inherits Dargent's separation between logical physical layouts but targets a more permissive, imperative environment. Where existing DSLs stop at schema generation, Cedar amis to provide static well-formedness checking and direct compatibility with verified C toolchains such as CompCert. In doing so, it connects the two worlds of declarative layout description and verified low-level systems programming.

2.5 Runtime Safety and Analysis Tools

When low-level layout errors escape compile-time detection, developers rely on **runtime instrumentation** and **static or dynamic analysis** to locate faults. These tools do not prevent misuse of memory, but they help identify common classes of errors such as buffer overflows, uninitialized reads, and invalid pointer dereferences.

2.5.1 Dynamic Analysis Tools

Dynamic instrumentation frameworks such as **Valgrind/Memcheck** [36] monitor every memory access to detect invalid reads and writes at byte granularity. Compiler-based sanitizers—including **AddressSanitizer (ASAN)**, **UndefinedBehaviorSanitizer (UBSAN)**, and **MemorySanitizer (MSan)**—insert checks into generated code to catch access errors and violations of the C standard at runtime [45, 35]. These tools have been widely adopted in testing and continuous-integration pipelines because they can reveal elusive bugs that arise from undefined behavior, alignment faults, and mis-computed offsets.

However, such techniques are inherently **reactive**. They only detect errors along exercised execution paths and their effectiveness depends heavily on the completeness of test coverage. In addition, instrumentation typically incurs in non-trivial runtime overhead—Valgrind can slow execution by an order of magnitude—making these tools unsuitable for production or embedded environments. While they are indispensable for debugging, they provide **no guarantee** that untested code is free from layout mismatches or that binary representations remain consistent across architectures.

2.5.2 Static Analysis and Type-Based Approaches

Complementary static analyses attempt to infer memory usage and pointer relationships directly from source code. Tools such as **Cclyzer** perform **points-to and structure-sensitive analysis** to detect potential type violations [6]. These methods can flag probable bugs before execution, but their results depend on heuristics and may yield false positives or incomplete coverage for low-level bit-manipulation code. Static analyses are particularly challenged by C's weak type system and unrestricted pointer arithmetic, which blur the boundary between logical structure and physical representation.

2.5.3 Safer Systems Languages

Alongside external tools, modern systems languages incorporate safety directly into their type systems. **Rust**, for example, enforces memory safety through ownership and borrowing rules that prevent data races and use-after-free errors at compile time [20, 5]. Attributes such as `#`

$$repr(C)$$

allow Rust structures to maintain binary compatibility with C while preserving the language safety guarantees [30]. However, Rust's model does not eliminate layout variability entirely: padding and alignment remain platform-dependent, and many performance-critical routines still resort to `unsafe` code when manipulating bit-fields or packed data [12].

Emerging projects such as **Carbon** aim to modernize C++ while improving safety and tooling [40]. Yet their treatment of low-level representation is still evolving; fine-grained layout control and verified layout correctness are not current design goals.

These language-level solutions demonstrate the growing demand for safer systems programming, but they also show how difficult it is to combine **fine-grained control** with **formal assurance**. Cedar occupies a complementary niche: it does not replace C or enforce a new ownership mode, but instead introduces **declarative layout specification** within existing C codebases, addressing correctness at the representation layer.

2.5.4 Hardware-Assisted Memory Safety

Recent research explores hardware extensions to enforce spatial and temporal safety directly in the architecture. The **Capability Hardware Enhanced RISC Instructions (CHERI)** architecture introduces bounds and permissions to pointers, preventing many classes of buffer overflow and use-after-free errors[16]. Building on this model, **Cornucopia** introduces efficient mechanisms for temporal safety in CHERI heaps. Such approaches complement compiler instrumentation by catching illegal accesses in hardware, greatly reducing overhead and enabling fine-grained isolation between components.

Despite their effectiveness, these mechanisms do not reason about **representation correctness**: they guarantee that accesses stay within allocated bounds, not that the bytes being accessed correspond to the intended logical structure. A program can be spatially safe but still misinterpret a binary layout if field offsets or bit-widths are wrong. Hardware capabilities thus solve one aspect of memory safety—protecting allocation boundaries—while leaving layout correctness to the programmer.

2.5.5 Positioning Cedar

Runtime and static tools collectively enhance robustness but they remain *after-the-fact* defenses. They identify violations of memory safety rather than enforcing *design-time correctness* of layouts. Cedar approaches the problem from the opposite direction: by **declaring layouts explicitly** and checking their compatibility with corresponding C types before compilation, it aims to prevent entire categories of representation errors that dynamic tools can only detect later. Where Valgrind and sanitizers ensure that a memory access is valid, Cedar, ensures that the access corresponds to the *right bytes*. In this sense, Cedar complements runtime verification and hardware enforcement, providing a static foundation on which these broader safety mechanisms can rely.

2.6 Summary and Research Gap

This chapter has reviewed the landscape work related to data-layout control and memory safety in systems programming. The discussion began with the **C memory model**, where representation details such as alignment, padding and field order remain underspecified and vary across compilers and architectures. These ambiguities complicate reasoning about correctness and performance in context such as device drivers and protocol stacks. Subsequent sections examined how **verified compilers** and **languages**, exemplified by CompCert, Cogent and Dargent, have provided rigorous guarantees about program semantics and layout correctness—but only within constrained or abstracted settings. Other efforts, from **declarative layout DSLs** like PADS and DataScript, to **industrial frameworks** such as FFlatBuffers or Cap'n Proto, have demonstrated that high-level schema descriptions can improve maintainability and portability, albeit without formal assurance for in-place C data structures. Finally, the chapter considered **runtime tools, hardware capabilities**, and **safer systems languages** such as Rust, which mitigate many classes of memory errors but do not address the root of the problem of explicitly specifying and verifying layout semantics.

Across these approaches, a consistent limitation emerges. Verified compilers assume layouts are correct but cannot check them. Declarative frameworks desribe layouts but operate outside the C toolchain. Runtime tools detect violations only after execution. None provide a **language-integrated, static guarantee** that a C data structure's physical representation corresponds to its intended logical form.

Cedar aims to bridge this gap. It brings declarative layout philosophy of Dargent into the C ecosystem, allowing programmers to define and check binary layouts directly alongside their types. By separating logical structure from physical representation, performing static well-formedness checks and generating cohesive C artifacts, Cedar seeks to make low-level layout design both expressive and reliable. The next chapter presents the **design methodology** and **systems architecture** underpinning this approach, outline how Cedar's compiler pipeline and language features were developed and evaluated.

3 Methodology and Implementation

This chapter presents the design principles and implementation of **Cedar**, a language and compiler for describing C layouts declaratively. The goal of the project is to give programmers **greater trust** that in-memory representations of data structures corresponds to their intent. Cedar achieves this by introducing a small external language for layout declarations, paired with a static checker that validates those declarations against C type definitions and generates layout consistent C code.

While we previously reviewed related work, this chapter focuses on what was built: the theoretical foundations on which Cedar stands, the kind of layouts it supports, and the design of its grammar and compilation pipeline. The work draws on two main sources of inspiration—**CompCert** and **Dargent**—whose ideas on form the conceptual basis for Cedar's semantics and tooling.

3.1 Foundations: CompCert and Dargent

Cedar's design is built upon the semantic foundations provided by **CompCert** [**leroy2009formal**], and the declarative layout principles introduced by **Dargent** [9]. Understanding these two systems explains both the motivation and structure of Cedar.

3.1.1 CompCert verified C semantics as a foundation

CompCert [22] is a formally verified compiler of the C language Its significance lies in **formal definition of the C memory model** [23], which specified how C objects are represented as contiguous blocks of bytes, how pointer arithmetic and aliasing work, and under what condition two memory states can be considered equivalent. Although Cedar is not yet built *within* CompCert, it adopts the same conceptual views on C memory: objects are typed, laid out in addressable bytes, and subjected to well-defined alignment and padding rules.

By grounding Cedar's reasoning in this model, layout checking can be interpreted consistently within verified C semantics. For example, when Cedar computes bit-ranges, and checks for non-overlapping fields, it effectively enforces that the resulting layout would be **well-formed under CompCerts memory abstraction**. The connection ensures that, if Cedar layouts were eventually compiled through CompCert, their structural assumptions would remain sound.

3.1.2 Dargent: declarative layouts and type-layout separation

While CompCert provides the semantic bedrock, Cedar's language design follows the philosophy of **Dargent** [9]. Dargent introduced a declarative approach to specifying how algebraic data types are represented in memory, separating the physical layout from the logical structure.

Cedar adopts key ideas—**type-layout separation** and **explicit field placement**—but targets them for plain C rather than Cogent. Unlike Dargent, Cedar does not require a restricted functional language, or a theorem prover; it provides lightweight static checking that enforces the same well-formedness properties. These adaptations transform Dargent's research concept into a more practical, C-oriented tool.

3.1.3 Bridging the Two influences

Cedar can thus be viewed as a **bridge between CompCert and Dargent**: From CompCert, it inherits a precise understanding of how C objects occupy memory; from Dargent, it inherits a declarative notation for describing that occupation explicitly. Cedar's goal is to close the gap between declarative layout specifications in self-contained languages, and the C ecosystem, producing verifiable code that remain compatible with existing C compilers.

The remainder of this chapter builds on these foundations. We continue by describing the subset of C types supported by Cedar and how layouts are represented. Later we detail the language patterns and features. Finally, we summarize the compiler architecture that realizes it.

3.2 Language Overview

Cedar operates over a well-defined subset of the C type system that reflect the representations defined by CompCert. The current compiler focuses on data structures whose layout is statically determinable: **structures, arrays, and tagged unions**. These cases cover the majority of layout-sensitive code in low-level programming while remaining tractable for static checking.

3.2.1 C types

$$
\begin{array}{llcll}
\text{C types (CompCert subset)} & \tau & ::= & \mathrm{int}(n, sg) & (n \in \{8, 16, 32, 64\};\ \text{signed/unsigned}) \\
 & & \mid & \_Bool & \\
 & & \mid & \mathrm{float}(p) & (p \in \{32, 64\}) \\
 & & \mid & \tau[n] & (\text{array of } n) \\
 & & \mid & \mathrm{struct}\ \{\, f_i : \tau_i \,\} & (\text{struct}) \\
 & & \mid & \mathrm{union}\ \{\, f_i : \tau_i \,\} & (\text{unions not yet implemented}) \\
 & & \mid & \tau\, * & (\text{pointers not yet implemented}) \\
\text{Field names} & & \ni & f &
\end{array}
$$

Figure 3: Cedar's type universe, aligned with CompCert C.

Figure **??** summarizes the core typing rules that relate Cedar layouts (ℓ) to C types (τ). Cedar's typing system aligns closely with CompCert's representation of C types [23]. Figure 3 summarizes the supported C types (τ) and their syntax.

3.2.2 Supported Constructs

Structures

Cedar treats C `struct` as ordered collections of named fields occupying contiguous regions of memory. Each field has a known size and alignment derived from its base type. A layout declaration for a structure describes the physical order and spacing of its fields using placement directives such as `at`, `before`, and `after`. During checking, Cedar ensures that:

- All fields of the corresponding C `struct` are represented exactly once,
- No two fields overlap in their bit ranges, and
- The overall size and alignment conform to the base types requirements.

This model allows Cedar to detect and properly handle changes that could otherwise be silently mishandled by compiler-specific padding or unordered field declarations—one of the most common causes of data-layout bugs in C.

Arrays

Arrays are represented as repeated layouts of a uniform element type. Cedar supports both fixed-length and statically known arrays. The layout checked verifies that the element layout is well-formed and then replicates its size and alignment across the declared array length. Array support enables Cedar to express memory regions such as device register tables, or network buffers that consist of repeated records.

Primitive and Derived Types

3.2.3 Primitive and Derived Types

Base types such as `char`, `short`, `int`, `long`, and their fixed-width equivalents (`int8_t`, `uint32_t`, etc.) are treated as atomic elements with known bit sizes and alignment constraints. Cedar assumes the same type sizes as the host compiler, which can be configured when the tool is built.

Pointers are not yet handled.

Each primitive type has its endianness specified in the layout declaration.

Layouts (surface syntax)	ℓ	$::=$	`record` $\{ f_i : \ell_i \}$	record layout
		$\mid$	@ n`B`	absolute offset (bytes)
		$\mid$	`after` f n`B` $\mid$ `before` f n`B`	relative offset
		$\mid$	$\ell[n]$	fixed-length array
		$\mid$	`endian` ω ℓ	endianness annotation
		$\mid$	$()$	empty / unit layout
Endianness	ω	$::=$	`LE` $\mid$ `BE` $\mid$ `ME`	little / big / machine

Figure 4: Surface syntax of Cedar layouts. Users describe field order, byte offsets, and endianness declaratively.

3.2.4 Layout Representation

```
layout Student = record {
  name   : U8[16] @ 0B,
  id     : U32    @ 16B endian LE,
  grades : record {
    maths   : U16 @ 0B  endian BE,
    physics : U16 @ 2B  endian BE
  } @ 20B
} @ 0B;
```

Figure 5: *
Cedar layout specification example

Internally, every Cedar layout is reduced to a **set of non-overlapping bit-ranges** withing a contiguous address space. Each range corresponds to a field array element and is annotated with its logical name, type, and alignment. The student example at Figure 5 illustrates this mapping for a simple `struct` example.

The compiler computer explicit offsets—`name` [0-255], `id` [256-287], `grade`[288-295]—and verifies that these regions are contiguous and non-overlapping.

This representation follows the same notion of "memory chunk" used in CompCert, where each object occupies a sequence of bytes and field addresses are computed as offsets within that chunk. By maintaining this correspondence, Cedar's checked layouts can be compiled through any C toolchain, including CompCert, without violating its memory-safe assumptions.

3.2.5 Scope and Limitations

The current compiler omits constructs whose semantics complicate static layout reasoning, such as: bitfields, flexible array members, recursive layouts, and untagged unions. These features will be considered in future work once the core checking mechanisms are stable. Despite these restrictions, the implemented subset is expressive enough to capture common data representations used in device drivers, network protocols, and binary file formats.

3.3 Cedar Language Design and Core Features

Cedar extends C with a minimal set of declarative constructs for specifying how data structures are laid out in memory. Its syntax follows C's lexical conventions but introduces new top-level `layout` blocks that describe the physical placement of fields. These layouts declarations are currently written in separate `.cedar` source file and linked to existing C type definition. The language is intentionally small: its expressiveness comes from composable placement directives rather than complex new forms.

3.3.1 Core Syntax

After parsing, Cedar lowers the surface syntax (Figure 4) into a simpler internal representation. Relative offsets are resolved into absolute byte positions. Endianness annotations are preserved for code generation.

Each `layout` declaration associates a name (matching a C type) with a collection of fields. Each field declaration specifies the type and placement of a field relative to the layout's origin or to other fields.

The `placement` clause may be absolute () or relative (`before`/`after`). Recursive or parametric layouts are excluded.

Layouts (core syntax)	ℓ	$::=$	$()$	unit / empty layout
		$\mid$	$\langle o, s \rangle$	bit-range starting at offset o of size s bits
		$\mid$	$\texttt{record}\ \{\, f_i : \ell_i \,\}$	nested record layout
		$\mid$	$\ell[n]$	array of n identical elements
Offsets	o	$\in$	$\mathbb{N}$	byte offsets (resolved statically)
Sizes	s	$\in$	$\mathbb{N}$	bit lengths (multiples of 8)

Figure 6: Core syntax of Cedar layouts after parsing and offset resolution. Greyed forms are shown for context only.

In this example, the `Student` layout begins at offset 0. The field `name` is an array of 32 characters occupying the first 256 bits; `id` is placed immediately after it, and `grade` follows `id`. Offsets are computed automatically during the layout-checking phase. If two fields were to overlap, or if a field were missing compared to the associated C type, the checked would reject the layout.

3.3.2 Layout-Type Matching

$$\text{Q}\vdash \ell \sim \tau \quad \frac{\text{bits}(T) = s \qquad T \text{ is a primitive integer or boolean type}}{\mathcal{Q} \vdash \langle o, s \rangle \sim T}\ \textsc{PrimMatch}$$

$$\frac{\text{for each } i,\ \mathcal{Q} \vdash \ell_i \sim \tau_i}{\mathcal{Q} \vdash \texttt{record}\ \{\, f_i : \ell_i \,\} \sim \texttt{struct}\ \{\, f_i : \tau_i \,\}}\ \textsc{RecordMatch}$$

$$\frac{\mathcal{Q} \vdash \ell \sim \tau}{\mathcal{Q} \vdash \ell[n] \sim \tau[n]}\ \textsc{ArrayMatch}$$

$$\frac{}{\mathcal{Q} \vdash () \sim ()}\ \textsc{UnitMatch}$$

$$\begin{array}{ll} \text{bits}(U8) = 8 & \text{bits}(U16) = 16 \\ \text{bits}(U32) = 32 & \text{bits}(U64) = 64 \\ \text{bits}(\_Bool) = 1 & \end{array}$$

Figure 7: Layout–type matching rules in Cedar. Each layout ℓ corresponds to a well-typed region of memory for type τ.

Figure 7 summarizes the core typing rules. This project implements a subset of Dargent's layout-type correspondence that applies to fixed-type size, byte addressed C records and arrays.

3.3.3 Well-formedness

$$\ell\ \mathbf{wf}\,\frac{}{\langle o, s\rangle\ \mathbf{wf}}\textsc{BitRangeWF} \qquad \frac{}{()\ \mathbf{wf}}\textsc{UnitWF} \qquad \frac{}{x\{o\}\ \mathbf{wf}}\textsc{VarWF}$$

$$\frac{\text{for each } i : \ell_i\ \mathbf{wf} \qquad [0.3em]\text{for each } i \neq j : \text{taken}(\ell_i) \cap \text{taken}(\ell_j) = \emptyset}{\texttt{record}\ \{\ f_i : \ell_i\ \}\ \mathbf{wf}}\textsc{RecordWF}$$

$$\frac{\begin{array}{c}\text{for each } i : \ell_i\ \mathbf{wf} \qquad [0.3em]v_i < 2^s \\ [0.3em]\text{taken}(\ell_i) \cap \text{taken}(\langle o, s\rangle) = \emptyset \qquad [0.3em]\text{for each } i \neq j : v_i \neq v_j\end{array}}{\texttt{variant}\ \langle o, s\rangle\ \{A_i(v_i) : \ell_i\}\ \mathbf{wf}}\textsc{VariantWF}$$

where taken(ℓ) $\subseteq \mathbb{N}$ denotes the set of bit positions that ℓ occupies.

Figure 8: Well-formed layouts in Cedar. A layout is well-formed if its fields occupy disjoint bit ranges, variant tags are unique, and all sublayouts are themselves well-formed.

Throughout this paper we refer to the notion of **well-formedness** to describe when a Cedar layout is internally consistent and represents a valid memory configuration. Figure 8 formalizes this notion using structural rules adapted from Dargent's layout calculus. Intuitively, a layout is well-formed if its constituent fields occupy non-overlapping regions of memory, variant tags are unique, and bounded by their declared bit-range, and all sub-layouts are themselves well-formed. The auxiliary function taken(ℓ) computes the set of bit positions that a layout ℓ occupies, ensuring that no two layouts overlap. This property guarantees that the generated C representation corresponds to a contiguous deterministic data structure in memory, forming the foundation for Cedar's soundness.

3.3.4 Design Considerations

Cedar's syntax supports declarative rather than arithmetic. Instead of writing numeric offsets directly, programmers may express structural relationships—"`id` after `name`"—that the compiler resolves into precise addresses. This improves readability and reduces the risk of human error, while still allowing absolute placement (`@`) when required for interoperability with binary formats and hardware registers.

To keep the semantics tractable, Cedar omits features like computed field expressions or conditional layouts. The resulting language is small enough to reason about statically and expressive enough to describe practical C structures.

3.3.5 Supported Directives Summary

Directive	Purpose	Example
`@`	Absolute placement.	`id : u32 @ 64B;`
`after`	Place field immediately after another.	`grade : u8 after id;`
`before`	Place field immediately before another.	`prefix : u8 before name;`
`layout`	Define a new layout block.	`layout Student ...`

Together , the grammar and these directives define Cedar's core language. They form the user-facing layer of the system which is then translated by the compiler into a sequence of well-defined intermediate representations. These representations capture progressively more information about the layout, allowing the checked to validate and later generate code for the specified structures.

3.4 Compiler Architecture

FIgure 9 illustrates Cedar's compiler, as designed by Anderson [2]. This project leaves the design unchanged, and builds upon the two-phase structure, extending the second phase with code generation capabilities as well as completing the first phase's parsing, and lexing components.

Cedar's compiler is organized as a sequence of **pure translation stages** that progressively enrich the information available about each layout. This implementation is lightweight and research-oriented.

Each stage transforms the program into a slightly more concrete form, preserving the structure while adding derived information such as absolute offsets and alignment constraints.

3.4.1 Parsing and Resolution

The grammar is parsed into a structure abstract syntax tree (`L`). A resolution pass then substitutes symbolic references (e.g., `after name`)

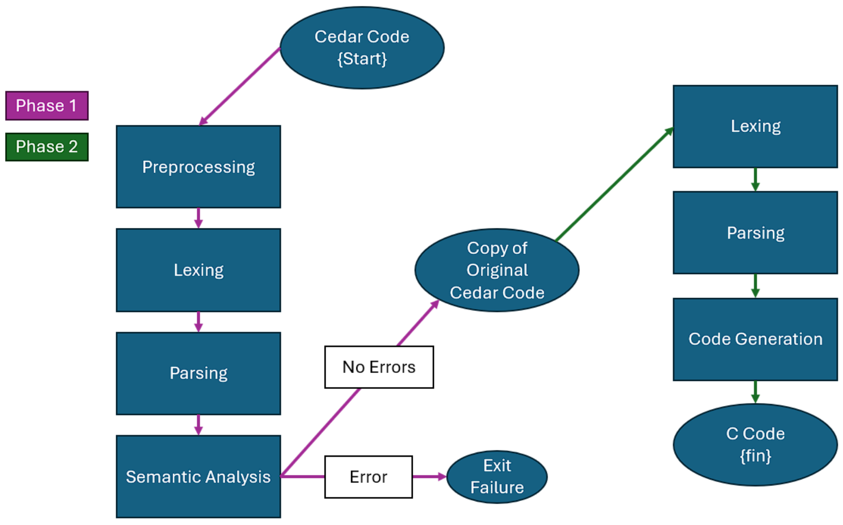

Figure 9: The Cedar compiler processing pipeline, adapted from Anderson's original proposal [2]. Phase 1 performs lexing, parsing, and semantic analysis of the input source code, while Phase 2 performs code generation to produce C output.

with explicit dependencies between fields. Each field returns its declared attributes—type, byte offset, relative placement, and optional endianness. At this point the layout is still declarative: field positions are described relative to one another rather than as numeric offsets.

The parser currently supports: Nested constructs, quoted field names, and forward references between fields. Comments and diagnostics, and multiple layouts per field are not yet implemented. Parse errors are reported generically.

3.4.2 Static Checking

In the next stage (LR), the compiler resolves **relative placements** and **nesting**. Every symbolic reference (`after x`, `before y`) is replaced with a concrete byte offset computed from previously defined fields. Nested `record` blocks are **flattenned** into a single linear sequence of fields using compound names (e.g., `grade_maths`). Endianness and element size information are propagated through this transformation.

The resulting *lowered representation (LR)* is a canonical, order-independent list of fields, each with a fixed, offset, size and endianness tag.

The **checked representation (CR)** introduces explicit byte ranges. Offsets are multiplied by eight to produce bit ranges for each field, which allow the code generator to reason about sub-word accessess and multi-word values. Cedar performs basic consistency checks during this phase:

- fields do not overlap,
- offsets must be non-negative and monotonically increasing, and
- all fields must have defined sizes.

Full alignment checking, total-size verification, and overlap proofs are not yet implemented, but the structure is designed to accommodate them later

3.5 Code Generation

The final stage translated the checked representation into compile-able C code. The output consists of:

1. A header file defining the a flat array of `uint32_t` words; and
2. A C source file containing *accessor* and *mutator* functions for each field.

Nested records are emitted as flattened field names; arrays are unrolled into repeat accessors. Per-field endianness annotations are honoured by inserting byte-swap operations in generated setters and getters.

No runtime support or special compiler modifications are required. The generated code can be compiled by any C compiler, including **GCC** and **CompCert**.

3.5.1 Code Generation Example

Cedar's back end translates a checked layout into portable C that treats the underlying object as a flat array of 32-bit "word lanes" and emits **byte-accurate** accessors/mutators for each field (and, when needed, for each byte "part" of a multi-byte field). The generated code is **standalone C**: no runtime library, no compiler extensions.

3.5.2 In-memory representation

In this section we will follow the example of the `Student` layout:

```
layout Student = record {
  name   : U8[16] @ 0B,
  id     : U32    @ 16B endian LE,
  grades : record {
    maths    : U16 @ 0B  endian BE,
    physics  : U16 @ 2B  endian BE
  } @ 20B
} @ 0B;
```

Listing 1: Cedar layout example

For each layout `T`, Cedar emits a header that defines a packed container:

```
struct T {
    unsigned int data[N];
};
typedef struct T T;
```

Listing 2: Generated C struct for layout Student

`N` is chosen so that all declared byte ranges fit exactly into 32-bit words. In the example below, `Student` needs 6 words (24B) to hold `name: U8[16]` (16B), `id: U32` (4B), and `record` (4B).

```
    struct Student { unsigned int data[6U]; };
```

Listing 3: Generated C struct for layout Student

Addressing model. Fields are addressed as **byte lanes** within `data[w]`, using shifts and masks. A U8 at byte `k` of word `w` is accessed by `(unint32_tdata[w] » (k * 8)) & 0xFF`. Multi-byte scalars are reconstructed from their constituent bytes according to the field's **declared endianness**.

3.6 Flattening of nested records and naming

Nested `record ...  @ off` are **flattened** with separators. For example:

```
    grades: record {math: U16 @0B; physics: U16 @ 2B
  ; } @ 20B
```

Listing 4: Cedar layout example

becomes `grades_math` at `20B + 0B = 20B` and `grades_physics` at `20B + 2B = 22B`. The header declares one getter/setter per flattened field:

```
    uint32_t d_get_grades_maths(const Student *b);
    void d_set_grades_maths(Student *b, uint32_t v);
```

Listing 5: Generated accessors for flattened fields

3.6.1 Scalars and Arrays

- **U8 arrays** generate one accessor per element(index in the symbol): `d_get_name__0`...`d_get_name__15`. Each reads/writes the corresponding byte lane.

  ```
          static uint32_t d_get_name__4__part0(
      const Student *b) {
              return ((uint32_t)(b->data[1U] >> 0U
      ) & 255U) //byte 4 overall -> word1, lane 0
          }
  ```

- U16/U32/U64 scalars are emitted as single getters/setters that **assemble** and **disassemble** the value by combining 8-bit parts with shifts in the declared endianness.

3.6.2 Emdianness Policy

Per-field endianness is honoured by the order of byte assembly/disassembly in the top-level getter/setter:

1. **Little-endian(**`endian LE`**)**: read/write bytes from lowest lane; bytes are combined as `« 0`, `« 8`, `« 16`, ...:

   ```
           // id : U32 @ 16B endian LE  -> all in
       data[4]
           static uint32_t d_get_id_part0(const
       Student *b) { return ((uint32_t)b->data[4U]
       >> 0U)  & 255U; }
           static uint32_t d_get_id_part1(const
       Student *b) { return ((uint32_t)b->data[4U]
       >> 8U)  & 255U; }
           static uint32_t d_get_id_part2(const
       Student *b) { return ((uint32_t)b->data[4U]
       >> 16U) & 255U; }
   ```

```
      static uint32_t d_get_id_part3(const
Student *b) { return ((uint32_t)b->data[4U]
>> 24U) & 255U; }

      uint32_t d_get_id(const Student *b) {
      uint32_t acc_u =
          ((uint64_t)d_get_id_part0(b) << 0U)
          | ((uint64_t)d_get_id_part1(b) << 8U
)
          | ((uint64_t)d_get_id_part2(b) << 16
U)
          | ((uint64_t)d_get_id_part3(b) << 24
U);
      return (uint32_t)acc_u;
      }
```

Listing 6: Generated accessor for U32 little-endian field

2. **Big-endian(**`endian BE`**)**: most-significant byte resides at the lowest address; assembly order is reversed: For `grades_maths: U16 @ 20B endian BE`: (bytes 0-1 of data[5]):

```
      static uint32_t
d_get_grades__maths_part0(const Student *b){
return ((uint32_t)b->data[5U] >> 0U) & 255U;
}   // MSB
      static uint32_t
d_get_grades__maths_part1(const Student *b){
return ((uint32_t)b->data[5U] >> 8U) & 255U;
}   // LSB

      uint32_t d_get_grades__maths(const
Student *b) {
      uint32_t acc_u =
          ((uint64_t)d_get_grades__maths_part0
(b) << 8U)   // MSB << 8
          | ((uint64_t)
d_get_grades__maths_part1(b) << 0U);  // LSB
<< 0
      return (uint32_t)acc_u;
 }
```

Listing 7: Generated accessor for U16 big-endian field

`grades_physics` is similar, using lanes 16 and 24 of `data[5]` with the reversed-byte assembly.

Machine endian `ME` *currently defaults as little-endian.*

3.6.3 Byte-part helpers and masking discipline

Each getter/setter is layered over private `_part<i>` helpers that isolate a single 8-bit chunk. Setters update the target word with a **read-modify-write** using a clear-then-insert mask:

```
 b->data[w] = (b->data[w] & ~(0xFFu << lane)) |
(((0xFFu & v) << lane));
```

This guarantees that writing one byte lane does not disturb adjacent lanes.

3.6.4 Example: offsets and word mapping

```
 \item \texttt{name: U8[16] @ 0B} $\rightarrow$ \
texttt{data[0]..data[3]}, lanes 0, 8, 16, 24 per
 element functions \texttt{d\_get\_name\_\_0 .. d
\_get\_name\_\_15}
 \item \texttt{id: U32 @ 16B endian LE} $\
rightarrow$ \texttt{data[4]}, lanes 0, 8, 16, 24;
 LE assembly (\texttt{<< 0, 8, 16, 24})
 \item \texttt{grades\_maths: U16 @ 20B endian BE
} $\rightarrow$ \texttt{data[5]}, lanes 0, 8; BE
assembly (\texttt{<< 8, 0})
 \item \texttt{grades\_physics: U16 @ 22B endian
BE} $\rightarrow$ \texttt{data[5]}, lanes 16, 24;
 BE assembly (\texttt{<< 8, 0})
```

Listing 8: Cedar layout example

3.6.5 Compilation Targets

The emitted code has been testted with **GCC** (for inspection via *pahole*) and **CompCert** (to verify compatibility with its memory model).

3.6.6 Current codegen limitations

1. The container is always a **flat** `unsigned int data[N]`; no C **struct** field nesting is emitted.

2. No diagnostics for overlapping byte field ranges at this stage, the code is assumed to be well-formed.
3. Pointers/boxed fields, tagged unions, and sub-type bit-fields are out of scope in this compiler.

3.7 Design Rationale

The architecture emphasizes two main principles:

1. **Transparency**. Each representation (L, LR, CR) can be inspected, making the compiler's reasoning traceable
2. **Modularity**. Additional analyses or transformations can be inserted between stages without disrupting others.
3. **Extensibility**. The separation of passes makes it straightforward to extend the language (for example, by adding union layouts or precise bit-field control) without disrupting the core pipeline.

This design allow Cedar to be extended in the future with additional features, and potentially support for automated verification, similar to Dargent.

It provides a practical foundation for Cedar's compiler: small enough to experiment with, both structured in a way that preserves the formal clarity of its inspirations.

3.8 Achievements and Scope

The Cedar compiler demonstrated that declarative data-layout specification can be integrated directly into the C programming environment in a practical way. Its development produced a complete end-to-end toolchain—from parsing and static checking of layout descriptions to the generation of layout-consistent C code—that validates the design concepts outlined in this chapter.

3.8.1 Implemented Features

The current implementation supports:

- **Declarative layout definitions** for `structs`, fixed-size arrays, and tagged unions.
- **Placement directives** (`at`, `before`, `after`) that determine field ordering and spacing.

- **Automatic offset computation** and total-size calculation.
- **C code generation** producing equivalent declarations and accessor functions.

Each of these features we tested on representative examples derived from Dargent's examples and from common systems-programming idioms such as device registers. The compiler successfully detected layout mismatches and generated compilable C code for all supported constructs.

3.8.2 Implementation Coverage

In this table we summarize the functionality currently implemented in the Cedar compiler, based on the grammar and pipeline described in the previous sections. The implementation realizes the *record-offset-endianness* fragment of the Dargent layout language and provides a complete -end-to-end compiler from parsing to C code generation.

Feature	**Status**	**Notes**
Layout declarations(`layout Name = record ...`)	Implemented	Only one layout per file.
Nested records	Implemented	Arbitrary depth; flattened in code generation
Absolute offsets (`@ n B`)	Implemented	Byte-based; no bit-granularity arithmetic
Relative offsets (`after`/`before`)	Implemented	Top-level only.
Per-field endianness (LE, BE, ME)	Implemented	ME defaults to LE
Uniform arrays (`type[n]`)	Implemented	Fixed-size only. No bounds checking
Quoted field names	Implemented	Support fields with names such as `"Field Name"`
Forward references between fields	Implemented	Declarative, order-independent layouts.
Signed integers	Partial	Handles by the parser, but not generated in codegen
Unions/variants	Not implemented	Planned for future work.
Sub-type bit-fields (1<B)	Not implemented	Not supported.
Complex offset expressions (arithmetic)	Not implemented	No evaluation of `a +1B`
Mixed-endian composited	Not implemented	No canonicalized
Comments and diagnostics	Not implemented	Comments are not lexed; generic parse errors only.
Pointer/boxed fields	Not implemented	Treated as opaque machine words.

Table 1: Implementation coverage summary

3.8.3 Practical Outcome

The resulting system is a functioning compiler that demonstrates:

1. Declarative layout syntax can describe C data structures succinctly;
2. Static checking can verify structural inconsistencies that ordinary C compilers accept silently;

3. Generated code remains portable and compatible with existing toolchains.

Together, these results validate Cedar's underlying hypothesis: that separating layout intent from C type declarations can improve programmers' confidence in the correctness of low-level data representations.

The next chapter presents concrete example and results, illustrating how Cedar's layout translate into verified C code and how its checker behaves on representative cases.

3.9 Summary

This chapter has described the design and implementation of Cedar, a language and compiler for declarative specification of C data layouts. Building on CompCert's precise model of C memory and Dargent' declarative layout philosophy, Cedar defines a practical subset of constructs—`layout`, `@`, `before`, `after` end simple arrays—that enable programmers to express physical data representations explicitly and check them statically.

The chapter introduced the supported C types, the language grammar, and the compiler architecture that transforms layouts through successive translation stages into layout-consistent C code. While the current compiler implements only a subset of the planned language, it demonstrated the feasibility of integrating declarative layout checking directly into C development workflows. The system provides static guarantees about field order, alignment, and completeness without altering C's execution model or requiring external verification frameworks.

The following chapter presents the evaluation of Cedar through concrete examples and generated code. It examples how well the language captures real C idioms, how the checker behaves on representative layouts, and what these results suggest for future extensions of the system.

4 Evaluation and Results

This chapter evaluated Cedar's implementation and tests how accurately its declarative layouts describe C data representations. While earlier chapters established the language design and implementation, here we verify that Cedar's generated C code behaves as intended: layouts occupy the correct byte offsets, respect endianness annotations, and match the structure of equivalent C programs.

The evaluation focuses on *layout correctness and fidelity* rather than performance or formal verification. We therefore use small but representative case studies to validate that Cedar's compiler and code generation phases preserve the semantics of the layout description down to the bit level. Each example was compiled to C, linked with a simple test driver, and compared against an equivalent reference layout written in plain C examined using the `pahole` tool.

The chapter proceeds as follows. We start by outlining the experimental setup and measurement procedure. Next we present two examples: a nested record structure demonstrating relative and absolute offsets, and a faithful representation of the 64-bit ELF header showing compatibility with existing binary formats. We conclude the chapter by summarizing findings and known limitations.

4.1 Experimental Procedure

To evaluate the correctness of Cedar's generated code, we adopted a reproducible five-step procedure that compares each layout's C representation to reference C code. All experiments were performed on a x86_64 Linux system, compiling **GCC 12.2.0** and **CompCert 3.16** to ensure consistency across a mainstream compiler, and a formally verified one. Both produced binaries with identical observable layouts in our examples.

1. **Cedar Layout Definition:** Each example was first expressed in Cedar using a combination of absolute and relative byte offsets.
2. **C Code Generation:** The Cedar compiler produced a `.c` and a `.h` file a flat structure definition, of unsigned integer words and a family of field-specific getter and setter functions. These files were compiled without manual modification.
3. **Link to a Driver:** A small C program was written for each example to instantiate the structure, assign deterministic test values through the generated setters, and retrieve them via getters. The driver also printed that raw bytes of thr structure to allow visual inspection of the offsets and endianness.
4. **Inspect Reference Layouts:** For each corresponding hand-written C baseline, we compiled using `gcc -O0 -g` and inspected

the DWARF debug information using **pahole**, which reports exact member offsets and structure sizes:

```
gcc -Wall -O2 -std=c99 -O0 -g -o <test> <sources>
pahole <object.o>
```

5. **Compare Outputs:** The cedar-generated layouts were inspected to match the reference offsets and field order. Successful round-trip conversions (set to get) confirmed that the accessors manipulated the correct bit ranges.

The following sections report these results.

4.2 Nested Record Layouts

The first example evaluated Cedar's handling of **nested records, relative offsets, and endianness**, three fundamental features of the language. This example is deliberately simple, but it exercises common layout idioms encountered in C programming, such as byte arrays and nested structures. Its purpose is to confirm that Cedar can generate accessor functions that correctly reconstruct the expected memory layout.

Layout Description

The cedar source from this example is shown below:

```
layout Student = record {
  name   : U8[16] @ 0B,
  id     : U32    @ 16B endian LE,
  grades : record {
    maths   : U16 @ 0B  endian BE,
    physics : U16 @ 2B  endian BE
  } @ 20B
} @ 0B;
```

This layout declares a student record containing a 16-byte ACII name, a 32-bit integer identifier stored in little-endian order, and a nested record `grades` comprising of two big-endian 16-bit fields. Absolute offsets `@ nB`, determine each field's byte position within the enclosing record, while endianness annotations specify the byte order for multi-byte fields.

Cedar compiles this layout into a `Student` structure represented as an array of 32-bit words, accompanied by automatically generated getter and setter functions such as `d_get_id` and so forth.

Each accessor isolates the relevant bit-range through shifts and masks, providing field-level access without any manual bitwise manipulation.

Reference C Layout

For comparison, an equivalent C hand-written C structure was defined as follows:

```
struct Grades {
  uint16_t maths;
  uint16_t physics;
};

struct Student {
  uint8_t   name[16];
  uint32_t id;
  struct Grades grades;
};
```

Listing 9: Reference C structure for the Student layout

Both the handwritten and Cedar-generated versions were compiled with GCC and CompCert. The compiled debug information was then examined using the `pahole` tool to determine field offsets and total structure size. Its reference output was:

```
    struct Grades {
        uint16_t maths;   /*     0     2 */
        uint16_t physics; /*     2     2 */

        /* size: 4, cachelines: 1, members: 2 */
        /* last cacheline: 4 bytes */
};
struct Student {
        uint8_t name[16];     /*     0    16 */
        uint32_t id;          /*    16     4 */
        struct Grades grades; /*    20     4 */

        /* size: 24, cachelines: 1, members: 3 */
        /* last cacheline: 24 bytes */
};
```

Listing 10: Pahole output for the Student structure

Testing and Results:

A small driver program exercised the generated accessors like so:

```
Student s = {0};
unsigned char name_bytes[16];
for (int i = 0; i < 16; i++) name_bytes[i] = i;
set_name(&s, name_bytes);
d_set_id(&s, 0x11223344);
d_set_grades__maths(&s, 0xABCD);
d_set_grades__physics(&s, 0x0123);
```

Listing 11: Driver code for testing the Student layout

The raw memory dump was:

```
00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F
44 33 22 11  AB CD 01 23
```

Which matches the endian representations for out inputs for `id`, `maths`, and `physics`. Getter calls returned the same values, confirming correct round-trip semantics. The total structure size (24 bytes) matched `pahole`'s output exactly, with no spurious padding or alignment differences.

4.2.1 Discussion

This experiment demonstrates that Cedar's layout and code generation phases produce a memory representation consistent with conventional C semantics for simple nested records. The language correctly handled both absolute and relative offsets and preserved endianness distinctions within the same record. Because Cedar's layout is declared independently of the C type, programmers can reason about the logical structure separately from its binary representation.

However, this example also illustrates some limitations. All fields are byte-aligned, and no bit-level packing or unions are yet supported. Nonetheless, this example successfully confirms the language's core promise: that a declarative specification can produce reliable layout-accurate code without manual offset calculations or platform-specific pragmas.

4.3 Representation of the 64-bit ELF Header

The Executable and Linkable Format (ELF) is a standard library representation for executables, and shared libraries on Unix-like systems [25]. Its header exemplifies the kind of low-level, performance critical structure that must be laid out with exact byte-level precision. It therefore provides a compelling stress test for Cedar's ability to reproduce externally defined binary formats.

Layout Specification:

An ELF64 header consists of a fixed 64-byte sequence of fields beginning with a 16-byte identifier array and followed by a combination of 16-, 32- and 64-but integers in a well-defined order. The following Cedar layout reproduces the structure:

```
layout Elf64_Ehdr = record {
  e_ident     : U8[16] @ 0B,

  e_type      : U16    @ 16B endian LE,
  e_machine   : U16    @ 18B endian LE,
  e_version   : U32    @ 20B endian LE,

  e_entry     : U64    @ 24B endian LE,
  e_phoff     : U64    @ 32B endian LE,
  e_shoff     : U64    @ 40B endian LE,

  e_flags     : U32    @ 48B endian LE,

  e_ehsize    : U16    @ 52B endian LE,
  e_phentsize : U16    @ 54B endian LE,
  e_phnum     : U16    @ 56B endian LE,
  e_shentsize : U16    @ 58B endian LE,
  e_shnum     : U16    @ 60B endian LE,
  e_shstrndx  : U16    @ 62B endian LE
} @ 0B;
```

The layout follows the specification exactly, producing a total of 64 bytes.

Reference C Layout:

The comparison was adapted from the ELF specification used in practice [25]. It is simplified for clarity, omitting custom type definitions:

```
    typedef struct {
    uint8_t  e_ident[16];
    uint16_t e_type;
    uint16_t e_machine;
    uint32_t e_version;
    uint64_t e_entry;
    uint64_t e_phoff;
    uint64_t e_shoff;
    uint32_t e_flags;
    uint16_t e_ehsize;
    uint16_t e_phentsize;
    uint16_t e_phnum;
    uint16_t e_shentsize;
    uint16_t e_shnum;
    uint16_t e_shstrndx;
} Mirror_Elf64_Ehdr;
```

Listing 12: Reference C structure for the ELF64 header

Compiling this structure with

```
gcc -Wall -O2 -std=c99 -g -c mirror.c
pahole mirror.o
```

produced the following DWARF-derived layout:

```
struct Mirror_Elf64_Ehdr {
    uint8_t         e_ident[16];    /*     0    16 */
    uint16_t        e_type;         /*    16     2 */
    uint16_t        e_machine;      /*    18     2 */
    uint32_t        e_version;      /*    20     4 */
    uint64_t        e_entry;        /*    24     8 */
    uint64_t        e_phoff;        /*    32     8 */
    uint64_t        e_shoff;        /*    40     8 */
    uint32_t        e_flags;        /*    48     4 */
    uint16_t        e_ehsize;       /*    52     2 */
    uint16_t        e_phentsize;    /*    54     2 */
    uint16_t        e_phnum;        /*    56     2 */
    uint16_t        e_shentsize;    /*    58     2 */
    uint16_t        e_shnum;        /*    60     2 */
    uint16_t        e_shstrndx;     /*    62     2 */

    /* size: 64, cachelines: 1, members: 14 */
};
```

Listing 13: Pahole output for the ELF64 header

This output represents the canonical ELF layout and serves as the reference for comparison.

Testing and Results:

The Cedar generated an equivalent structure encoded as a 16-element array of 32-bit words, along with accessor functions for each field. A test driver populated the results with fixed data and printed its raw bytes:

```
7F 45 4C 46 00 00 00 00 00 00 00 00 00 00 00 00
02 01 04 03 44 33 22 11 88 77 66 55 44 33 22 11
00 FF EE DD CC BB AA 99 EF CD AB 89 67 45 23 01
D4 C3 B2 A1 10 0E 14 12 18 16 1C 1A 20 1E 24 22
```

Each field aligned exactly with the byte offsets in the ELF specification. The printed byte sequence verifies the little-endian encoding was preserved for all multi-byte fields. The output for GCC and CompCert was identical.

4.3.1 Discussion

This experiment demonstrates that Cedar can accurately reproduce the memory representations of externally defined data formats.

With further development, Cedar can serve as a reliable bridge between verified toolchains such as CompCert, and existing C ecosystems, as more complex layouts are supported.

Although these experiments do not extend to bit fields or unions, they validate the design premise that declarative layout descriptions can coexist with industrial data structures without losing meaning.

4.4 Discussion of Findings and Limitations

These experiments demonstrate that the code generated by Cedar faithfully reproduces expected memory layouts and is suitable for concrete data representations. The examples tested, *Student* and *Elf64_Ehdr*, generated C code whose memory layout matched that of equivalent hand-written structures exactly, down to byte offsets and endianness ordering. These results validate the correctness of the compiler's translation stages. However, they also highlight several limitations and areas for future work.

Accuracy and Correspondence:

The *Student* example confirmed that Cedar handles nested records, arrays, and mixed endianness correctly. Each generated accessor retrieved the expected byte sequence, and the overall structure size (24 bytes) matched the reference layout as inspected by `pahole` for GCC.

Similarly, the *Elf64_Ehdr* example provided a test of Cedar's precision, reproducing an externally standardized binary format with distinct fields of varying widths. Every field aligned with the corresponding offset in the official ELF specification, and the printed byte sequences showed the correct little-endian encoding. These results indicate that Cedar's lowering and code-generation phases preserve layout semantics across multiple scales, from simple records to real industry protocol headers.

The ability to compile the generated code successfully under both GCC and CompCert further supports Cedar's portability. Because the implementation uses only standard C99 constructs and a flat bit representation, the emitted code remains architecture-independent within th assumed word-size configuration. This compatibility demonstrates that Cedar's static checking aligned with the memory model used by industry and verified compilers, suggesting that a future verified backend could integrate without layout mismatch.

Expressiveness and Scope:

Cedar's current feature set covers a practical representative subset of systems-programming layouts: fixed-sized arrays, nested records, and explicit per-field endianness. These constructs are sufficient to model common patterns in C programming. The separation between layout description and logical types allow programmers to verify local correctness independent of the surrounding program logic. The declarative syntax, using directives like `@ nB`, `before`, and `after`, proved readable and unambiguous in all test cases. In practice, the language reduces the cognitive overhead associated with manually calculating offsets and alignment, which are traditionally major sources of error in C code.

Limitations:

Despite these strengths, Cedar has several limitations remain. Most notably, the current implementation **does not yet perform full well-formedness checking**, allowing overlapping fields to be declared. Although internal representations (LR and CR) compute explicit offsets and bit ranges for each field, the compiler does not **validate that those ranges are disjoint or without bounds**. This potentially produces undefined behavior in the generated C code. The rules for well-formedness have been outlined in this project, taking inspiration from Dargent's rule, and implementing their enforcement is a priority for future work, as this is the first step in guaranteeing memory safety. This is a major gap between the theoretical model and the current implementation.

Beyond this, the language presently excludes several common C constructs such as bit-fields, unions and pointers. Supporting these features safely would demand a richer semantic model and explicit tagging or variant typing to prevent aliasing conflicts. As a result Cedar cannot yet represent packed or overlapping layouts, nor structures whose size depends on runtime parameters. The restriction limits expressiveness compared to C's full repertoire, but could be implemented in future work.

Error reporting is also minimal. The compiler assumes syntactically correct input and does not provide detailed diagnostics when semantic inconsistencies occur. Extending the well-formedness pass to detect and localize overlaps, gaps and misalignments would not only increase soundness but also improve usability.

Trustworthiness and Practicality Cedar therefore stands at a middle ground between an unverified C and fully verified systems like Cogent. The current prototype delivers **transparency without enforcement**: layouts are explicit and easy to inspect, but the tool does not yet guarantee their correctness. In this sense, Cedar's results demonstrate *potential* trustworthiness, rather than actual proof of soundness. The infrastructure for static checking exists, The CR stage encodes all the information necessary, but the checking logic has been delegated to future work. Once complete, it will align the implementation with the formal well-formedness model presented in this paper and enable stronger correctness claims.

4.5 Summary

The evaluation demonstrates that Cedar achieves its primary goal: to allow programmers to describe and generate precise, byte-level layouts for C structures through a small declarative language. Across the representative examples, the *Student* and *Elf64_Ehdr*, Cedar's generated code reproduced the expected in-memory representation exactly, matching the offsets and sizes reported by industry-standard compilers and the inspection tool `pahole`. These outcomes show that declarative layout specification can be made practical withing conventional C toolchains, providing immediate benefits in clarity and consistency without introducing new runtime dependencies.

At the same time, the examples reveal the boundaries of the current implementation. WHile Cedar captures layout intent faithfully, and generates correct code for well-behaved inputs, it does not yet enforce full well-formedness. The compiler computes bit ranges for every field but does not verify that those regions are disjoint or contiguous. As a result, malformed layouts may still pass unchecked. This limitation, along with the absence of bit-fields, unions and pointer types, delineates the scope of the present implementation and highlights where future development could focus.

Despite these constraints, Cedar already demonstrates a meaningful step toward trustworthy data layout specification in low-level programming. By externalizing layout intent and compiling it into portable deterministic C code, it provides a mode of how structural correctness can be expressed explicitly rather than inferred implicitly from type declarations and compiler heuristics. The resulting system combines the transparency of declarative description with the familiarity of C's execution model.

Overall, the evaluation confirms Cedar's design principles of separation of logical and physical structure, compositional layout rules and alignment with existing C semantics, are effective in practice. These results provide the empirical foundation for the discussion that follows, which examines Cedar's broader implications, the lessons learned from its design, and the directions for extending the language and implementation toward verified correctness.

5 Future Work

Although the examples shown show how valuable a tool like Cedar can be, it also highlights several avenues for future development.

5.1 Language and Compiler Features

There are a few features that were considered out of scope for this initial implementation, however they heavily limited the real world examples that could be tested with Cedar. Implementing these features would greatly increase the usefulness of the language and set it up as a tool useful for real world systems programming and research.

1. Complete well-formedness checking: The current implementation does not perform a full static analysis of the layout. Implementing this should be a priority, and the current design leaves space for this to be added in future work. The CR stage calculates

offsets and sizes required for this check to to take place, but it is not yet implemented.

2. Missing Structures: Bit fields, unions and pointers are commonly found structures in C programs, and their support would immediately increase the amount of layouts that could be expressed with the language.
3. Unified Array accessors: A trivial but useful addition would be to ensure the codegen stage generates a unified accessor for a whole array, as well as per-element accessors. This would reduce the amount of glue code required to integrate Cedar intro existing C codebases.

5.2 Towards a Verified System

As with its predecessors, Cedar was conceived as an idea for a verified system for data-layouts. The design of the language reflects this, and undertaking the task of ensuring each of the internal translations stages are formally verified, would be the first step into turning Cedar into a verified tool that can integrate easily with verified compilers such as CompCert in order to provide end-to-end guarantees about code behavior.

Formalizing Internal Representations:

Each translation stage (L, LR and CR) encodes a refinement of information. From a symbolic structure, to resolved offsets and finally concrete bit ranges. These transformations are functional and self-contained, enabling automated reasoning about their correctness with a proof assistant such as Roq/Coq. Formalizing their behavior would enable proofs that layout meaning is preserved across stages: that the semantics of a record array in L correspond exactly to its bit-range representation in CR. Such results would elevate Cedar into a verifiable component with a broader compilation framework.

Integrating with Verified Compilation:

An interesting direction would be to develop a custom C compiler, most likely adapting CompCert as it already provides a verified pipeline, and integrate the Cedar compiler into that pipeline. This would allow for Cedar code to be written in-line along with C code, creating a seamless experience for programmers who need a fully verified toolchain for systems programming.

Tooling visualization:

As Cedar's analysis grows more sophisticated, corresponding improvements in usability will become essential. Interactive visualization of layouts, showing byte ranges, field names and endianness, could transform the compiler into a teaching and debugging aid.

6 Conclusion

This project set out to investigate whether a layout specification language could be developed for C. Its purpose was to enable data-layout correctness directly in language design, without relying on external tooling. In this paper we presented extensions to *Cedar*, a declarative language. We built a compiler with a fully implemented pipeline that allows programmers to describe data layouts down to the byte level.

The research demonstrated that it is possible to achieve these goals. It shows that a compact syntax can avoid issues of alignment, or the need to use compiler-specific instructions and non-portable code. Through a minimal syntax and moving layout reasoning to be a first-class concern we demonstrated that it is possible to reproduce real-world layouts, including complex structures such as the ELF header, with high fidelity, and portable code that works across different compilers

Cedar contributes both the conceptual framework for designing data layouts declaratively, and a functioning compiler that realizes this vision. This offers an alternative perspective on programming: correctness can emerge from declaration and specification, without relying on defensive testing and debugging.

At the same time, the work highlights clear opportunities for a continuation. Completing semantic checking, extending type coverage, and integrating with verified backends such as CompCert would advance Cedar as a tool in an ecosystem of full assurance. These directions promise not only a more powerful tool but also a research vehicle for studying how declarative specifications interact with formal proofs and real-world compilers.

In summary, Cedar represents a step towards more reliable and trustworthy systems programming. It bridges the clarity of declarative design, with the pragmatism and industry relevance of C, showing that many of the benefits can be achieved through language structure alone. The project stands as both a functioning compiler and a methodological argument: that improving how we *describe* data is the first step towards improving how we *trust* it.

References

[1] Sidney Amani et al. "Cogent: Verifying high-assurance file system implementations". In: *ACM SIGARCH Computer Architecture News* 44.2 (2016), pp. 175–188.

[2] George Anderson. *Cedar, A Data Layout Description Languange for C*. Nov. 18, 2024.

[3] Godmar Back. "DataScript-A specification and scripting language for binary data". In: *International Conference on Generative Programming and Component Engineering*. Springer. 2002, pp. 66–77.

[4] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. "Programming languages for distributed computing systems". In: *ACM Comput. Surv.* 21.3 (Sept. 1989), pp. 261–322. ISSN: 0360-0300. DOI: 10.1145/72551.72552. URL: https://doi.org/10.1145/72551.72552.

[5] Abhiram Balasubramanian et al. "System Programming in Rust: Beyond Safety". In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. HotOS '17. Whistler, BC, Canada: Association for Computing Machinery, 2017, pp. 156–161. ISBN: 9781450350686. DOI: 10.1145/3102980.3103006. URL: https://doi.org/10.1145/3102980.3103006.

[6] George Balatsouras and Yannis Smaragdakis. "Structure-sensitive points-to analysis for C and C++". In: *International Static Analysis Symposium*. Springer. 2016, pp. 84–104.

[7] Michael Barr. *Programming embedded systems in C and C++*. "OReilly Media, Inc.", 1999.

[8] Guilin Chen, Mahmut Kandemir, and Mustafa Karakoy. "A constraint network based approach to memory layout optimization". In: *Design, Automation and Test in Europe*. IEEE. 2005, pp. 1156–1161.

[9] Zilin Chen et al. “Dargent: A silver bullet for verified data layout refinement”. In: *Proceedings of the ACM on Programming Languages* 7.POPL (2023), pp. 1369–1395.

[10] David A Cock. “Bitfields and Tagged Unions in C: Verification through Automatic Generation.” In: *VERIFY* 8 (2008), pp. 44–55.

[11] L Joshua Crotts. “An Investigation of Compiler-Induced Vulnerabilities and Insecure Optimizations”. In: ().

[12] Mohan Cui et al. “Is unsafe an Achilles’ Heel? A Comprehensive Study of Safety Requirements in Unsafe Rust Programming”. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering.* 2024, pp. 1–13.

[13] Pascal Cuoq, Loı¨c Runarvot, and Alexander Cherepanov. “Detecting strict aliasing violations in the wild”. In: *International conference on verification, model checking, and abstract interpretation.* Springer. 2017, pp. 14–33.

[14] Ralph Duncan, Peder Jungck, and Dwight Mulcahy. “Portable Bit Fields in packetC”. In: *packetC Programming.* Springer, 2011, pp. 363–369.

[15] Nuno Faria, Rui Silva, and João L. Sobral. “Impact of Data Structure Layout on Performance”. In: *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing.* 2013, pp. 116–120. DOI: `10.1109/PDP.2013.24`.

[16] Nathaniel Wesley Filardo et al. “Cornucopia: Temporal safety for CHERI heaps”. In: *2020 IEEE Symposium on Security and Privacy (SP).* IEEE. 2020, pp. 608–625.

[17] Kathleen Fisher and Robert Gruber. “PADS: a domain-specific language for processing ad hoc data”. In: *ACM Sigplan Notices* 40.6 (2005), pp. 295–304.

[18] Michael J Hohnka et al. “Evaluation of compiler-induced vulnerabilities”. In: *Journal of Aerospace Information Systems* 16.10 (2019), pp. 409–426.

[19] Mahmut Kandemir et al. “Enhancing spatial locality via data layout optimizations”. In: *European Conference on Parallel Processing*. Springer. 1998, pp. 422–434.

[20] Steve Klabnik and Carol Nichols. *The Rust programming language*. No Starch Press, 2023.

[21] Chaitanya Koparkar. “Efficient Data Representation Using FlatBuffers”. In: *XRDS: Crossroads, The ACM Magazine for Students* 29.2 (2023), pp. 50–51.

[22] Xavier Leroy et al. “CompCert-a formally verified optimizing compiler”. In: *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. 2016.

[23] Xavier Leroy et al. “The CompCert memory model, version 2”. PhD thesis. Inria, 2012.

[24] Xin Li et al. “A realistic evaluation of memory hardware errors and software system susceptibility”. In: *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. 2010.

[25] Linux Foundation. *ELF Header — Executable and Linkable Format (ELF) Specification, Version 4.0*. System V Application Binary Interface, Edition 4.0. 2020. URL: `https://refspecs.linuxfoundation.org/elf/gabi4+/ch4.eheader.html` (visited on 10/06/2025).

[26] Deepak Majeti et al. “Compiler-driven data layout transformation for heterogeneous platforms”. In: *European Conference on Parallel Processing*. Springer. 2013, pp. 188–197.

[27] Marjan Mernik and Viljem Žumer. “Incremental programming language development”. In: *Computer Languages, Systems & Structures* 31.1 (2005), pp. 1–16.

[28] Liam O’Connor et al. “Bringing effortless refinement of data layouts to Cogent”. In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2018, pp. 134–149.

[29] Paul C van Oorschot. “Memory errors and memory safety: C as a case study”. In: *IEEE Security & Privacy* 21.2 (2023), pp. 70–76.

[30] Graf von Perponcher-Sedlnitzki and Philipp Christian. “Integrating the future into the past: Approach to seamlessly integrate newly-developed Rust-components into an existing C++-system”. PhD thesis. Technische Hochschule Ingolstadt, 2024.

[31] Benjamin Perry and Martin Swany. “Automatic realignment of data structures to improve mpi performance”. In: *2010 Ninth International Conference on Networks*. IEEE. 2010, pp. 42–47.

[32] Deepti Raghavan et al. “Breakfast of champions: towards zero-copy serialization with NIC scatter-gather”. In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. 2021, pp. 199–205.

[33] Dennis M Ritchie. “The development of the C language”. In: *ACM Sigplan Notices* 28.3 (1993), pp. 201–208.

[34] Lilly Sell. “Security Evaluation and Vulnerability Assessment of FlatBuffers”. In: ().

[35] Konstantin Serebryany et al. “AddressSanitizers: A fast address sanity checker”. In: *2012 USENIX annual technical conference (USENIX ATC 12)*. 2012, pp. 309–318.

[36] Julian Seward and Nicholas Nethercote. “Using Valgrind to Detect Undefined Value Errors with Bit-Precision.” In: *USENIX Annual Technical Conference, General Track*. 2005, pp. 17–30.

[37] Laurent Simon, David Chisnall, and Ross Anderson. “What you get is what you C: Controlling side effects in mainstream C compilers”. In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2018, pp. 1–15.

[38] Vilas Sridharan et al. “Memory Errors in Modern Systems: The Good, The Bad, and The Ugly”. In: *SIGARCH Comput. Archit. News* 43.1 (Mar. 2015), pp. 297–310. ISSN: 0163-5964. DOI: 10.1145/2786763.2694348. URL: https://doi.org/10.1145/2786763.2694348.

[39] Michael M Swift et al. "Nooks: An architecture for reliable device drivers". In: *Proceedings of the 10th workshop on ACM SIGOPS European workshop*. 2002, pp. 102–107.

[40] Lucian Radu Teodorescu. *Safety, Revisited.* URL: `https://www.accu.org/journals/overload/32/179/teodorescu/`.

[41] Andrew Tolmach, Chris Chhak, and Sean Anderson. "Defining and preserving more C behaviors: Verified compilation using a concrete memory model". In: *15th International Conference on Interactive Theorem Proving (ITP 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. 2024, pp. 36–1.

[42] Harvey Tuch. "Formal memory models for verifying C systems code". PhD thesis. UNSW Sydney, 2008.

[43] Juan Cruz Viotti and Mital Kinderkhedia. "A survey of json-compatible binary serialization specifications". In: *arXiv preprint arXiv:2201.02089* (2022).

[44] Wm A Wulf and Sally A McKee. "Hitting the memory wall: Implications of the obvious". In: *ACM SIGARCH computer architecture news* 23.1 (1995), pp. 20–24.

[45] Wei Zhao et al. "CapSan-UB: Bug Capabilities Detector Based on Undefined Behavior". In: *Proceedings of the 2023 13th International Conference on Communication and Network Security*. 2023, pp. 6–11.

Appendix

The full source code for the zCedar prototype is available as a downloadable archive at the University of Melbourne OneDrive:

`https://unimelbcloud-my.sharepoint.com/:u:/g/personal/kchinchilla_student_unimelb_edu_au/EW3nqKIyx0tAqhim_3NEGxgBUU792ExNyvd3U7eMNiXjDA?e=K9PHc5`]

The archive contains the full Haskell implementation of Cedar, including parsing, lowering type-checking, and code-generation modules, as well as test layouts and build scripts. The code corresponds exactly to the version referenced in this paper.