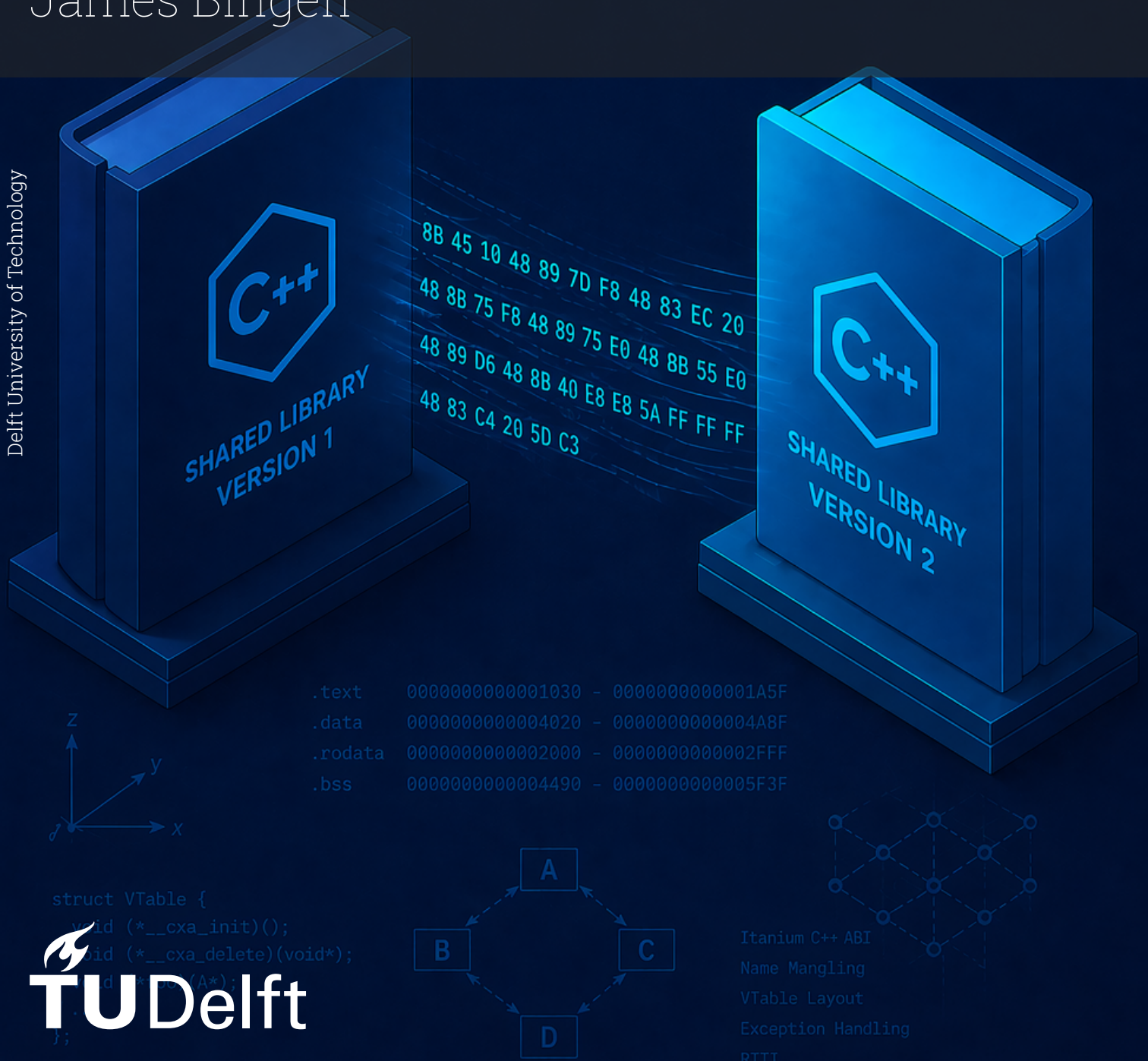
Binary Interface Compatibility for Library Integration

in C++

James Bingen



Delft University of Technology

TU Delft

Binary Interface Compatibility for Library Integration

in C++

by

James Bingen

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday August 13, 2026 at 01:00 PM.

Thesis committee:	Dr. Soham Chakraborty,	TU Delft, supervisor
	Prof. dr. ir. Georgi Gaydadjiev,	TU Delft
	Maximiliano Pin,	ASML, daily supervisor
	Nikolaos Boras,	ASML, daily supervisor
Project duration:	September 8, 2025 – June 1, 2026	
Student number:	4993772	

Cover: Generated using OpenAI DALL·E 3

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

This thesis sits at the intersection of two interests that I have pursued throughout my studies: low-level systems and formal methods. I have long been drawn to complex engineered systems, such as aviation systems, compilers, and operating systems, where correctness rests on every layer underneath agreeing with every layer above. ABI compatibility is exactly that problem in miniature: a binary contract between two pieces of compiled code, easy to break by accident and difficult to reason about by hand. The opportunity to formalize that contract in a proof assistant, and to do so as part of an industrial internship, was an ideal fit for both interests.

The work presented here is my MSc thesis in Computer & Embedded-Systems Engineering at Delft University of Technology, carried out between September 2025 and June 2026 in collaboration with ASML, where it took the form of a research internship.

I express my sincere gratitude to my supervisors at ASML, Nikolaos Boras and Maximiliano Pin, for their guidance throughout my internship. I am especially thankful for the freedom they gave me in shaping the research direction, and for the many discussions that deepened my understanding of C++ at a level that few projects ever require.

I am also grateful to my university supervisor, Soham Chakraborty, for his support and insightful feedback throughout this work. His guidance on the formalization and on techniques for extracting ABI-related information was instrumental to the success of this thesis.

I would also like to thank my bachelor's thesis supervisor, Hans-Dieter Hiep, for first introducing me to formal methods and for the patience with which he taught me a way of thinking about correctness that has shaped how I approach problems years later. Without that early exposure, I would not have pursued this direction.

To my grandparents and my father, I am grateful for the continuous support and encouragement throughout my studies.

Finally, I would like to thank my sister and my sister-in-law, as well as my friends Thijs Wartena, Tingting Wang, Akram Hamdi, Shan Huang, Yannan Shi, George Ragios, and Yaman Kwefati, for their encouragement, support, and motivation throughout the research process.

James Bingen
Delft, August 2026

Summary

Shared libraries evolve, and when a new version changes its binary interface in a way that an already-compiled consumer depends on, the program can fail at load time or silently misbehave at runtime. These application binary interface (ABI) breaks are easy to introduce and hard to diagnose, and in a large software stack a single missed break can force costly recompilation across many components.

This thesis presents `veriabi`, an ABI compatibility checker for C++ shared libraries whose decision procedure is formally verified. Compatibility is specified as an explicit predicate over three layers of the Itanium C++ ABI, covering exported symbols, function and object signatures, and type layout. For each rule a decision procedure is defined in the Rocq proof assistant and proved sound and complete against the specification. The verified core is extracted to OCaml and paired with an unverified C++ front-end that reads ELF and DWARF from two compiled libraries. An optional consumer-aware extension takes the consumer's compiled LLVM IR and narrows the verdict to the breaks that consumer can observe, with the filtering proved sound with respect to the strict result.

On a controlled corpus with known ground truth, `veriabi` catches every in-scope break (18 of 18), while the established tools `abidiff` and `abi_compliance_checker` each miss two to three mainstream changes, such as a base-class reorder or a strong-to-weak binding change. On a 554 MB production library from ASML it completes in about four minutes against more than two hours for the closest comparable tool, and produces a report two orders of magnitude smaller. The work shows that ABI compatibility checking can be given an explicit specification, a decision procedure, and a machine-checked proof that the two agree, at a cost low enough to run on real industrial libraries.

Contents

1 Introduction

Software systems at companies like ASML rely on large numbers of interdependent C++ libraries. These libraries are continuously updated, and new versions must seamlessly integrate with their existing products. However, ensuring Application Binary Interface (ABI) compatibility across versions is not trivial. Unlike Application Programming Interfaces (APIs), which are explicitly defined in header files, ABIs describe how data structures, symbols, and function calls are represented at the binary level. Even small changes such as modifying a class layout or altering a virtual function table can silently break consumers at runtime without any warnings.

At ASML, such ABI breaks have a direct and costly impact. When an ABI break occurs, consumers may need to recompile or modify code. Recompilation can take, in the case of the 'twinscan' product, more than eight hours to complete. However, any modifications to the library can take almost a month to be processed, thus delaying the release schedule of their libraries. To avoid these disruptions, library maintainers need access to a reliable method to detect and prevent ABI-breaking changes before release.

1.1. Research Motivation

ABI compatibility is not unique to ASML. Any project that ships compiled C++ libraries to consumers compiled separately from the library itself faces the same problem. Linux distributions maintain ABI stability across package upgrades for system libraries such as `libstdc++` and `glibc` [10]. Plugin hosts such as VST audio plugins, browser extensions, and operating system kernels all rely on stable ABIs between components built at different times with different compilers. In each of these cases, a silent ABI incompatibility may result in crashes or data corruption occurring far from the location of the originating change.

The problem is sharper in C++ than in C. The C++ Itanium ABI [28] is the common platform ABI on Linux, macOS, and most Unix systems, and it specifies dozens of contracts that must agree between library and consumer: struct layout, alignment and padding rules, virtual table contents and ordering, runtime type information, exception unwinding tables, vague-linkage semantics, and calling conventions for edge cases such as non-trivial destructors and virtual inheritance. A single inserted virtual function shifts every vtable slot below it. A reordered base class shifts every member in every subclass. A changed enumerator invalidates `switch` statements compiled against the old enum.

This thesis was carried out as an internship project at ASML, where there is currently no dedicated ABI checker in use: neither an in-house tool nor a third-party one. Compatibility is enforced through reviews and interface boundaries, and failures typically surface as downstream build or runtime errors. The available open-source alternatives are `abi-compliance-checker` [23], which has not seen an upstream update for several years and no longer keeps pace with newer C++ features and compiler revisions, and LIBABIGAIL with its command-line tool `abidiff` [27], which extracts type and symbol information from DWARF debug data and compares two versions to report differences in human-readable form. Even the most actively maintained tools in this area share a common limitation: the notion of compatibility is encoded as hand-written procedural logic without an independent formal specification. Consequently,

when a reported incompatibility is disputed, there is often no external ground truth beyond the implementation of the checker itself. Similarly, when an incompatibility is not detected, it may remain unnoticed until it manifests as a downstream build issue, runtime failure, or data corruption.

It is worth asking why such tools are underrepresented today. Part of the answer lies in a broader shift in how software is deployed. Cloud services and containerized applications ship with their dependencies, bypassing shared-library compatibility by statically fixing each dependency version per deployment [19, 13]. Newer systems programming languages such as Go and Rust tend towards static linking by default [7, 31], while the C++ ecosystem itself is gradually moving towards modules and other mechanisms that promote tighter source-level coupling [26]. In consumer software, shared-library ABI stability is simply less visible than it once was. In industrial stacks like ASML's, however, where shared libraries remain the dominant method of deployment and must survive across builds spanning months or years, the problem has not gone away.

This thesis takes the position that ABI compatibility checking is a decision procedure for a well-defined predicate and, therefore, has a specification that can be written down, inspected, and proven against.

1.2. Contributions

This thesis contributes:

1. A mechanised specification of ABI compatibility for a subset of the C++ Itanium ABI, covering symbol-level, signature-level and layout-level compatibility.
2. A decision procedure proven sound and complete against the specification.
3. A C++ front-end that reads ELF and DWARF from two shared libraries, builds a typed representation of their ABIs, and hands it to the verified decision procedure (extracted from Rocq to OCaml) through an OCaml-C bridge.
4. An evaluation on a test suite and on two ASML libraries.
5. A publicly released open-source tool, `veriabi`, released by ASML under the Mozilla Public License 2.0 [4].

1.3. Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 frames the problem and states the research questions. Chapter 3 provides the necessary background, including the API/ABI distinction, the binary formats from which ABI information is extracted (ELF and DWARF), the intermediate representation used in the consumer-aware extension (LLVM IR), and the Rocq proof assistant. Chapter 4 situates this work within the context of existing ABI compatibility tools and prior academic research.

The core of the thesis is presented in the next three chapters. Chapter 5 defines the overall workflow: how ABI information is extracted from compiled libraries, how compatibility rules are identified and formalized in Rocq, how each rule is paired with a computable decision procedure, and how this procedure is certified via a per-rule soundness-and-completeness biconditional. It also describes how the resulting verdict is refined through consumer-aware analysis.

Chapter 6 then instantiates this workflow for each rule in scope, providing its mathematical specification, Rocq encoding, decision procedure, proof of the soundness-and-completeness biconditional, and a concrete example of an incompatible change.

Chapter 7 details the implementation pipeline, from DWARF extraction through the C++/OCaml bridge to the extracted decision procedure, and discusses the main engineering challenges encountered.

Chapter 8 extends the checker with consumer-aware filtering, which restricts reported incompatibilities to those affecting a specific consumer binary. Chapter 9 evaluates the tool empirically on a small test suite and two ASML libraries. Finally, Chapter 10 concludes the thesis and outlines directions for future work.

2

Problem Statement

Modern software systems, such as those developed at ASML, rely heavily on C++ libraries that are developed, versioned, and deployed independently. These libraries are often integrated either as source code or as precompiled binaries. While binary integration enables decoupled development, it introduces the risk of Application Binary Interface (ABI) incompatibilities, which may lead to runtime failures, undefined behaviour, or subtle data corruption in dependent applications.

Ensuring ABI compatibility in C++ is challenging due to complex language features, compiler-specific ABI conventions, and the low-level nature of binary interfaces. Even small source-level modifications, such as changes to function signatures or data layout, may result in incompatibilities that are difficult to detect using conventional testing or review processes.

Existing tools, such as `abidiff` from LIBABIGAIL [27], analyse compiled binaries using DWARF debug information and report structural differences between library versions. While platform ABIs such as the Itanium C++ ABI specify the low-level binary interface contract between library and consumer, existing tools do not embed a machine-checkable specification of *compatibility* against which the tool's own logic can be independently verified. Compatibility rules are implemented as procedural code, and tool correctness is established primarily through implementation-level reasoning and testing rather than through a formal connection to a separate specification.

This motivates the following research question:

> *Can we design and implement an ABI compatibility checking tool for C++ libraries that operates on compiled binaries and whose decision procedure is formally verified with respect to an explicit specification of compatibility?*

To address this question, this thesis investigates how ABI compatibility rules can be formally specified, how such specifications can be connected to a computable decision procedure over DWARF-derived representations, and how formal verification can be used to establish correctness guarantees for the resulting checker. Specifically, the following subquestions are considered:

1. Which changes in C++ libraries can cause ABI incompatibilities, and how can they be formally specified in a machine-checkable form?
2. How can we design and implement a checker that automatically detects these incompatibilities between two library versions, extracted directly from compiled binaries?
3. Can formal methods be applied to prove the decision procedure sound and complete with respect to this specification?
4. How can the tool take advantage of knowledge about actual consumer usage to distinguish between guaranteed breaks and incompatibilities that are harmless in practice for a given consumer?
5. Can the tool be applied to real-world C++ libraries used at ASML, and what are the practical limitations of this approach when faced with industrial-scale binary sizes?

2.1. Scope

This thesis focuses on binary compatibility of exported symbols in C++ shared libraries on Linux systems following the Itanium C++ ABI [28], the common platform ABI on Linux, macOS, and most Unix systems. The following aspects are covered:

- **Symbol-level compatibility.** Exported functions and global objects, their defined or undefined state, kind (function, object, or RTTI typeinfo), and binding strength (strong or weak).
- **Function and object signatures.** Return types, parameter types, parameter counts, variadic flags, and the types of exported global variables.
- **Type layout compatibility.** Structs, unions, and enums, including their size, alignment, members, bitfield layouts, base classes (including virtual inheritance), and virtual dispatch tables.
- **Consumer-aware filtering.** Restricting the set of reported incompatibilities to those that are observable through a given consumer's LLVM IR, in addition to the strict library-versus-library comparison.

The following are explicitly out of scope:

- Non-Itanium ABIs, in particular the Windows MSVC ABI.
- Full template compatibility reasoning beyond what surfaces in DWARF debug information as concrete type instantiations.
- ABI changes introduced by differing compiler versions on the same source code (we compare two already-compiled binaries, not source files).
- Exception unwinding tables (`.eh_frame`) and thread-local storage models, which have their own ABI contracts orthogonal to the ones we address.
- SONAME and library-level versioning, which belong to the packaging layer rather than the type-level compatibility layer.

2.2. Approach in Brief

The approach taken in this thesis separates the checker into a front-end that extracts ABI information from ELF and DWARF metadata of the two compiled libraries under comparison, and a verified core that decides compatibility on this extracted representation. An optional consumer-aware refinement narrows the verdict to incompatibilities observable through a given consumer binary. The methodology and the implementation of each component are developed in Chapters 5 and 7.

3

Background

This chapter introduces the technical background knowledge that is assumed throughout the remainder of this thesis. It covers the distinction between API and ABI compatibility and the types of changes that affect each, the binary formats from which the ABI checker extracts information (ELF and DWARF), the intermediate representation used in the consumer-aware extension (LLVM IR), and the proof assistant used to formalize the model and decision procedures (Rocq).

Readers already familiar with any of these topics may skip the corresponding sections.

3.1. API vs ABI

A library exposes two contracts to its consumers. The first is the *Application Programming Interface* (API): the source-level interface against which the consumer is written, consisting of declarations such as function signatures, type definitions, macros, and templates in the library headers. The second is the *Application Binary Interface* (ABI): the binary-level contract that dictates how compiled code interacts with the library at runtime, including details such as symbol names, calling conventions, type layouts, and object representations [9, 20].

API and ABI are related but distinct. Two library versions are *API-compatible* if source code that compiled against the old version also compiles against the new one and introduces no semantic changes. They are *ABI-compatible* if a binary that was compiled and linked against the old version continues to load and execute correctly against the new one without recompilation [9, 20].

API compatibility is a property of source interfaces and is checked whenever a consumer is rebuilt. In contrast, ABI compatibility concerns artifacts already compiled and the runtime environment. Since the consumer is not recompiled, the compiler does not re-check the contract. A library may therefore preserve API compatibility while breaking ABI compatibility, in which case the problem only appears when an existing consumer binary is executed against the new library version. Conversely, the API may change while the ABI remains unchanged, so that existing binaries continue to run but new source builds fail.

The distinction can be subtle in practice. Consider the change shown in Figure 3.1, where the `red` and `blue` fields of a `Color` struct are swapped. At the source level this is an API break: code written against `red` now refers to the field that previously stored `blue`, and vice versa. However, the ABI records only the binary layout of the object, not the field names. The layout is therefore unchanged between versions. Three `uint8_t` values appear at offsets 0, 1, and 2, and a consumer compiled against v1 continues to load and execute against v2 without complaint from the dynamic linker. Since the ABI does not carry names, this swap is indistinguishable at the binary layer from a pure rename or from no change at all, and a checker confined to binary contracts must, correctly, report all three identically. The change is therefore an API incompatibility but not an ABI incompatibility.

v1 header

```
struct Color {
  uint8_t red;
  uint8_t green;
  uint8_t blue;
};
```

v2 header

```
struct Color {
  uint8_t blue;
  uint8_t green;
  uint8_t red;
};
```

Figure 3.1: A change that is API-incompatible but ABI-compatible.

The difference is particularly clear in C++. The Itanium C++ ABI [28] is the widely used platform ABI on Linux, macOS, and most Unix systems. It defines a large set of binary-level details that the C++ language itself leaves implementation-defined, such as class layout, virtual table organization, symbol mangling, runtime type information, virtual inheritance, and the handling of inline functions and template instantiations. Each of these affects the binary contract between library and consumer, and relatively small source-level changes can be enough to break it. Examples include inserting new class members, reordering virtual methods, changing inheritance structure, or modifying enum representations.

3.2. Layers of the ABI

The binary contract between a library and its consumers can be analyzed at three layers, each contributing a distinct set of constraints that must agree across versions for compatibility to hold. The same layering organizes the kinds of break this thesis addresses. Any change that violates a constraint at one of these layers is an ABI break.

At the *symbol layer*, the ABI defines what a library exposes to its consumers. Each exported symbol has a name, a kind (function, object, or RTTI typeinfo), and a binding strength (strong or weak). Breaks in this layer include the removal of a symbol the consumer references, a change in its kind such as a function becoming a variable, or a change in its binding strength that alters which definition the dynamic linker selects when multiple libraries define the same name.

At the *signature layer*, the ABI defines how functions are called and how exported objects are represented. This includes the return type, the number and types of parameters, whether the function is variadic, and the type of any exported global objects. Breaks here include changing the return type, adding or removing parameters, or changing the variadic property. On most platforms these changes also affect the calling convention, so a consumer compiled against the old version continues to load but reads or writes values in the wrong registers at runtime.

At the *layout layer*, the ABI defines how data structures are laid out in memory. This includes the size and alignment of structs, unions, and enums, the offset and type of each member, the placement of base classes including virtual inheritance, and the order and contents of virtual function tables. Breaks at this layer include changes in struct size or alignment, changes in member offsets or types, shifts in base class layout, and changes in the order or size of virtual tables.

Symbol layer. Exported entities, kind (function / object / RTTI), binding strength.
Example: a removed export, or a strong symbol weakened to weak.

Signature layer. Return type, parameter list, variadicity, object type.
Example: an inserted parameter, or a changed return type.

Layout layer. Struct/union/enum size, member offsets, base classes, vtables.
Example: a shifted member offset, or a reordered vtable slot.

Figure 3.2: The three layers of ABI.

What unifies the three layers is that every break is undetectable from the consumer's compile-time view of the interface. The consumer is compiled once against the old headers and then executes against a new binary that no longer satisfies the original interface assumptions. The resulting runtime behaviour depends on the nature of the mismatch. A removed symbol may lead to a load-time `symbol lookup error` from the dynamic linker. A changed calling convention can cause silent register misinterpretation. A shifted member offset can lead to arbitrary memory writes. In some cases, there is no immediate failure, and the inconsistency only becomes visible later when corrupted data is accessed. Figure 3.2 summarizes the three layers and a representative break of each. Chapter 6 returns to these layers and formalizes a subset of these breaks as explicit rules.

3.3. ELF and DWARF

A C++ shared library on Linux is distributed as an *ELF* (Executable and Linkable Format) object [32]. ELF organizes a compiled binary into sections that carry code, data, dynamic-linking information, and metadata. The dynamic symbol table section (`.dynsym`) in ELF is central to ABI analysis and lists every symbol the library exports for use by other binaries. Each entry records a symbol name, an address, a size, a type (function, object, or no-type), and a binding strength (global, weak, or local). ELF alone is enough to recover the *symbol-level* surface of the library but does not record any of the type information needed to compare signatures or layouts. For example, an exported function symbol carries an address and a size in bytes, but nothing about its return type, parameter list, or calling convention.

Type information comes from *DWARF* [1], the standard debug-information format used by GCC and Clang on Linux. DWARF is emitted into the binary at compile time when the compiler is invoked with debug flags such as `-g`. It is stored in a set of dedicated sections, including `.debug_info`, `.debug_abbrev`, and `.debug_str`. The information is organized as a tree of *debug information entries* (DIEs). Each function in the source code corresponds to a DIE that records attributes such as its source name, mangled linkage name, return type, and parameter list. Each struct is represented by a DIE with child entries for its members and base classes, where each member is annotated with its byte offset. DIEs are linked through section offsets: a function or struct DIE refers to the DIEs of the types it uses, allowing full type structures to be reconstructed by following these cross-references.

Figure 3.3 shows the DIE tree emitted by Clang for the C++ snippet from Listing 3.4. The compile unit at offset `0x0c` contains three siblings: a `DW_TAG_subprogram` for `total_legs`, a `DW_TAG_pointer_type`, and a `DW_TAG_structure_type` for `Animal`. The subprogram has a nested `DW_TAG_formal_parameter` whose `DW_AT_type` attribute points, via the pointer-type DIE, at the structure DIE. This is the chain used to recover the type of `a`. The subprogram records both the source name and the Itanium-mangled `DW_AT_linkage_name`, which is the symbol that appears in the dynamic symbol table. The `Animal` DIE here carries only `DW_AT_declaration` because its definition lives in a different translation unit.

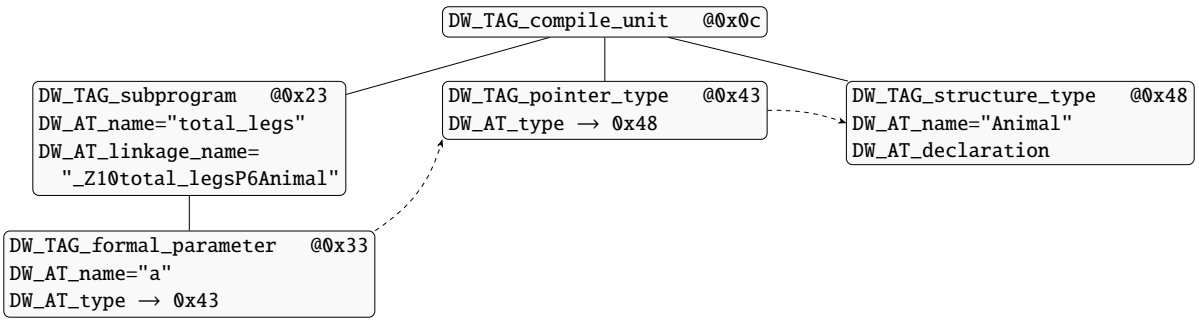


Figure 3.3: DWARF DIE tree for the C++ snippet of Listing 3.4, as printed by `objdump -dwarf=info`. Solid edges show parent–child nesting; dashed arrows show `DW_AT_type` cross-references.

An ABI checker for shipped libraries therefore relies on both ELF and DWARF. ELF determines which entities are exported, and DWARF determines their typed shape. The dependence on debug information is a real constraint: production builds that strip DWARF leave no types to compare, and partial builds (`-g1`) omit attributes such as member offsets that the layout checker depends on. Chapter 7 describes how the tool handles these constraints in practice.

3.4. LLVM IR

The consumer-aware extension of Chapter 8 requires a third input in addition to the two library versions: a representation of the consumer's compiled code. The representation used in this work is *LLVM IR* [15, 29], the typed static-single-assignment intermediate representation used by the LLVM compiler infrastructure. LLVM IR is generated by `clang` between parsing and machine-code generation, and can be emitted either as a textual `.ll` file or as binary LLVM bitcode (`.bc`). Both forms represent the same program.

An LLVM IR module is the top-level unit emitted per translation unit. A module contains global variables, function declarations and definitions, named type definitions, and metadata. Global variables and functions represent the entities defined or referenced by the module. Named type definitions record the layout of aggregate types such as structs. Metadata nodes, written using the `!` syntax, attach auxiliary information to instructions and declarations without affecting program semantics.

Function bodies are represented in *static single-assignment* (SSA) form. Each named value is assigned exactly once, and control-flow merges are represented using `phi` instructions. SSA form simplifies analysis because every value has a unique definition and explicit data dependencies.

Several properties of LLVM IR make it suitable for consumer-aware ABI analysis. First, LLVM IR is strongly typed. Function parameters, memory operations, loads, stores, and aggregate accesses all carry explicit type information. This makes it possible to recover how a consumer interacts with library types without re-running source-level type analysis.

Second, common source-level constructs compile into highly regular IR patterns. A C++ virtual call, for example, typically compiles into a sequence consisting of:

1. a load of the vtable pointer,
2. a `getelementptr` selecting a vtable slot,
3. a load of the function pointer from that slot,
4. and an indirect `call`.

Because this structure appears consistently across consumers, virtual dispatch sites can be identified reliably by pattern matching over the SSA graph. Figure 3.4 illustrates the correspondence between a simple C++ virtual call and its generated LLVM IR. The IR shown has been lightly simplified for readability. Temporaries and metadata annotations have been omitted, but the dispatch structure itself is unchanged. The consumer-aware analysis of Chapter 8 relies on some of that omitted metadata, in particular the TBAA tag on the vtable-pointer load, which `clang` emits at `-O1` and above.

Third, LLVM IR may contain debug-information metadata such as `!DIType` and `!DILocalVariable`. These annotations map SSA values back to source-level names and types. Without this information, a vtable access appears only as a sequence of pointer operations. However, with debug metadata, the analysis can associate the access with a specific class type and source-level entity.

```
struct Animal {
  virtual int legs() const;
};

int total_legs(Animal* a) {
  return a->legs();
}
```

```
define i32 @_Z10total_legsP6Animal(ptr %a) {
  %vtable = load ptr, ptr %a
  %vfn    = getelementptr inbounds ptr, ptr %vtable, i64 0
  %fn     = load ptr, ptr %vfn
  %call   = call i32 %fn(ptr %a)
  ret i32 %call
}
```

Figure 3.4: A C++ virtual call (left) and the LLVM IR it compiles to (right).

Together, these properties allow the analysis to recover which parts of a library ABI are actually used by a consumer. From a single IR module, the checker can determine which exported symbols are referenced, which struct members are accessed, which virtual-function slots are used, and which library types are instantiated or allocated by the consumer. The consumer-aware filter uses this information to

distinguish between incompatibilities that are observable for a given consumer and incompatibilities that remain unused in practice.

The approach does impose constraints. The consumer must be compiled using a toolchain capable of emitting LLVM IR, which in this work means `clang`. This trade-off, along with the practical limitations of IR-based analysis, is discussed further in Chapter 8.

3.5. Rocq and Verified Programming

The compatibility specification and decision procedure of Chapter 6 are written in *Rocq* [30], an interactive proof assistant in which programs and the propositions about those programs are written in the same dependently-typed functional language. The label "Rocq" is the recently adopted name for what was historically called Coq. The system, its tactic language, and its standard library are unchanged. Rocq has been used to verify a C compiler against an operational semantics of C [16] and a number of smaller but mathematically substantial systems. It is mature enough that the proof obligations of this thesis fit comfortably inside what its tactic language can deliver.

A Rocq development typically follows the same three-step structure. The author first writes an executable function (in our case, a checker that returns the list of detected breaks), then a propositional specification of what the function should compute (in our case, a relation that describes when two ABIs are compatible), and finally proves the function correct against the specification (in our case, a biconditional theorem stating that the checker returns the empty list exactly when the relation holds). The proof itself is constructed interactively, by applying tactics that reduce the goal to simpler subgoals until each leaf is closed by a primitive rule. The proof script is checked by Rocq's kernel, a small core trusted to be correct. Any mistake in the proof is caught by the kernel before the theorem is accepted.

A small example illustrates the structure:

```
Definition is_zero (n : nat) : bool :=
  match n with
  | O   => true
  | S _ => false
  end.

Theorem is_zero_correct :
  forall n, is_zero n = true <-> n = 0.
Proof.
  intro n. destruct n; simpl; split; intro H;
    first [reflexivity | discriminate].
Qed.
```

The function `is_zero` returns a Boolean value indicating whether a natural number equals zero. The theorem `is_zero_correct` states that `is_zero n` evaluates to `true` if and only if `n` equals zero. The biconditional (⇔) is essential here because it requires both directions to be proven. The proof therefore establishes both soundness (the function never returns `true` for a non-zero value) and completeness (the function never returns `false` for zero). The proof script achieves this by case analysis on `n` and by discharging each resulting case using reflexivity or a contradiction.

The compatibility checkers developed in Chapter 6 follow the same overall structure. Each checker function is paired with a theorem of the form `check_X` $\texttt{env}_1$ $\texttt{env}_2$ $\cdots = [] \Leftrightarrow$ `X_spec` $\texttt{env}_1$ $\texttt{env}_2$ $\ldots$, asserting that the checker reports no incompatibilities exactly when the corresponding compatibility relation holds between the two ABI representations. The executable functions operate over structured ABI representations rather than individual values, and the associated proofs are correspondingly more involved, but the underlying proof pattern remains the same.

Once the executable functions have been proven correct, Rocq's extraction mechanism [17] translates them into OCaml source code that can be compiled and executed as ordinary OCaml programs. During extraction, proofs are erased because they are only required during proof checking and carry no runtime meaning. Rocq's inductive datatypes are translated into equivalent OCaml representations.

The extracted OCaml is then linked with the C++ front-end through a thin bridge layer, described in Chapter 7, allowing the verified decision procedure to run as part of the final toolchain. Under the assumptions of Rocq's trusted kernel and extraction mechanism, the resulting proofs establish that the extracted decision procedures satisfy their corresponding specifications. The verification argument remains localized after extraction. Defects in the C++ front-end appear either as malformed ABI representations that are rejected by the verified core or as incorrect inputs whose effects can be attributed to the unverified part of the system. Any error in the verified core would have been detected by Rocq's kernel during proof checking before the theorem was accepted.

4

Related Work

This chapter looks at four areas of prior work related to this thesis. The first treats ABI fragility at the language level by designing compilers that give programmers explicit control over the binary contract. The second consists of post-hoc analysis tools that compare two compiled versions of a library and report differences, without providing formal guarantees about the analysis itself. The third looks at how other language ecosystems study and check the same compatibility problem. The fourth is the broader body of formally verified systems software, in particular verified compilers, whose methodology this thesis adapts to ABI compatibility checking.

The chapter concludes by positioning this work within these four areas.

4.1. Language-Level Approaches to ABI Compatibility

Atkinson, Flatt, and Lindstrom propose ZL, a C++-compatible systems language designed to address the problems associated with fragile and incompatible ABIs [3]. In ZL, high-level constructs such as classes are defined through macros over a C-like core language. This gives programmers direct control over ABI-relevant properties such as class layout, virtual table organization, and calling conventions. As a result, developers can implement idioms like fixed-size classes or the pointer-to-implementation (pimpl) pattern, and can even define entirely new ABIs tuned for either performance or long-term compatibility. A central advantage of ZL is that different ABIs can coexist within the same program, that existing compiler ABIs from GCC and Clang can be combined, and that binary compatibility can be preserved even when class definitions evolve.

The ZL approach relies on a new compiler and programming model, which makes it difficult to adopt where large C++ codebases and toolchains are already established. ZL also focuses on giving developers the tools to build ABI-stable software from the ground up, rather than addressing stability in libraries that already exist and continue to evolve through ordinary versioning. The work of this thesis takes the opposite stance: it accepts the standard Itanium ABI as a given and asks how to check, after the fact, whether two compiled versions of a library remain compatible under that contract.

4.2. ABI Compliance Checker

Ponomarenko and Rubanov present a method for automatically identifying binary compatibility problems, implemented in the tool `abi-compliance-checker` [22]. Their approach distinguishes two definitions of compatibility. The first is backward binary-level compatibility, which refers to the ability of an application to continue running against a newer version of a library without recompilation. The second is source-level backward compatibility, which guarantees that applications can be recompiled against the new version of a library without source-code changes. In practice, this means that while a rebuild may be necessary, the source-level interface remains stable.

A central contribution of their work is the combination of binary-level analysis with API-level analysis. Unlike earlier tools that compare only compiled binaries, `abi-compliance-checker` also inspects the

library headers. This lets it capture not only low-level layout and symbol changes but also higher-level API differences that affect ABI compatibility. The tool generates a detailed HTML report listing detected incompatibilities, such as missing symbols, changed function signatures, or altered class layouts.

`abi-compliance-checker` is a post-hoc analyzer: given two versions of a library, it reports whether they are binary-compatible. Its notion of compatibility is encoded as procedural code without an independent specification, and, as discussed in Chapter 1, the tool has not seen an upstream update for several years.

4.3. Libabigail and `abidiff`

Libabigail is a C++ library providing tools to construct, manipulate, serialize, and deserialize *ABI artifacts*: structured representations of the types, variables, functions, and declarations of a library or application [27]. A complete set of such constructs forms an *ABI corpus*. The library is shipped with command-line tools such as `abidiff` and `abicompat`, which compare two ABI corpora such as two versions of a library and report the differences.

`abidiff` categorizes each detected change into one of three outcomes. A change marked **harmless** indicates an ABI difference that the tool considers safe by default. A change marked **harmful**, signalled by the `ABIDIFF_ABI_CHANGE` exit-code bit, indicates a difference that may or may not break consumers, and therefore requires human review. A change marked **incompatible**, signalled by the `ABIDIFF_ABI_INCOMPATIBLE_CHANGE` bit, indicates a change that is considered definitively breaking. Libabigail currently classifies only two changes as incompatible in this strict sense: the removal of an exported symbol referenced by other binaries, and a modification of a virtual function table slot index.

The conservative middle category, changes that may or may not break a given consumer, highlights a structural limitation of differential tools that analyze only a library. Consider a library that defines a struct `Foo` with two integer members, together with a function that returns a pointer to one of them:

```
struct Foo { int a; int c; };
Foo* make_foo();
```

Suppose two consumers exist. Consumer A accesses both members; Consumer B accesses only `a`:

```
// Consumer A
Foo* f = make_foo();
std::cout << f->a << " " << f->c << std::endl;

// Consumer B
Foo* g = make_foo();
std::cout << g->a << std::endl;
```

If the library inserts a new field between `a` and `c`, so that the struct becomes `{a, b, c}`, the layout of `Foo` changes and `c` now sits at a different offset. The pointer arithmetic baked into the compiled consumer binaries still assumes the old layout. Consumer A reads from the wrong offset and exhibits undefined behavior: a real ABI break for it. Consumer B accesses only `a`, which remains at the same offset, and is unaffected. `abidiff` would mark the change as harmful for both, since it has no way to know which members are actually used. The tool cannot distinguish a break that affects all consumers from one that affects only some, and so it conservatively flags every reachable layout change as a candidate. The consumer-aware extension of Chapter 8 makes a first step towards addressing this gap.

4.4. Compatibility Checking in Other Ecosystems

ABI and API stability is not specific to C++. The languages that face it most have produced both empirical studies of how compatibility breaks in practice and practical tools to detect it. This work lies outside the C++ toolchain this thesis targets, but it shows that the underlying problem is general and that automated compatibility checking is an established need.

Empirical studies show that breaking changes are common even where a versioning discipline is meant to prevent them. Raemaekers, van Deursen, and Visser study the Maven repository and find that a large share of releases, on the order of a third, introduce at least one breaking change, and that many do

so without the major-version increment that semantic versioning prescribes [24]. Dietrich, Jezek, and Brada study Java programs and classify how a library upgrade breaks its clients, distinguishing changes that surface as compile-time, link-time, or run-time errors, and observe that the risk of such breakage makes developers reluctant to upgrade at all [6]. The situation mirrors C++. A source-level version number is a weak guarantee, and the binary contract has to be checked directly.

A related group of tools decides, for a specific update, whether it introduces an incompatibility, rather than only comparing two versions in the abstract. Foo et al. present a lightweight static analysis that compares library versions at the method level and reports whether an upgrade would break the projects that depend on it. It is light enough to fit into a continuous-integration pipeline and is evaluated across the Maven, PyPI, and RubyGems ecosystems [8]. Checking an update against the code that actually depends on it is similar in spirit to the consumer-aware analysis of Chapter 8, though Foo et al. work at the API level of managed languages rather than on compiled C++ and its ABI.

The established tools in these ecosystems, such as `japicmp` [11] and Revapi [25] for Java, `cargo-semver-checks` [5] for Rust, and `apidiff` [2] for Go, are mature and widely used. Like `abidiff` and `abi_compliance_checker`, however, they encode compatibility as procedural logic without an independent, machine-checked specification. This thesis differs from all of them in that respect.

4.5. Verified Systems Software

Another area of related work asks not how to detect bugs in systems software, but how to rule them out by proof. A well-known example is CompCert, a C compiler whose every optimization pass is proved correct against an operational semantics of C [16]. CompCert's correctness theorem states that the compiled assembly preserves the observable behavior of the source program for every input the source semantics admits. The proof is mechanized in Rocq, and the verified passes have, in independent random-testing experiments, exhibited no miscompilation bugs against compilers such as GCC and Clang. CompCert's significance for this thesis is methodological rather than technical: it demonstrates that a non-trivial systems tool can be specified, decided, and proved correct end-to-end in a proof assistant, and that the proof can be extracted into running code without a separate re-implementation step. The same method is applied to ABI compatibility checking, which is the focus of this thesis.

Other verified systems work, including the seL4 microkernel [12] and the CakeML compiler [14], demonstrates that this methodology scales beyond CompCert. None of these efforts target ABI compatibility directly, but together they show that the formal-verification approach is mature enough to deliver real systems software rather than only paper proofs.

Closer to the representations this thesis consumes, Vellvm provides a formal semantics of the LLVM intermediate representation in Rocq and uses it to verify program transformations [33]. The consumer-aware analysis of Chapter 8 reads LLVM IR but treats it syntactically, matching instruction patterns rather than reasoning about a formal semantics. A verified account of the IR-level properties used by this analysis, such as the representation of virtual calls after lowering, could build on a formal LLVM semantics combined with verified compiler and ABI assumptions.

4.6. Positioning

The areas surveyed point to three distinct strategies for the ABI compatibility problem. ZL addresses fragility at the language level, by giving programmers a compiler that makes the binary contract explicit and controllable. `abi-compliance-checker`, `abidiff`, and the per-ecosystem tools of Section 4.4 address it at the artifact level, by comparing two compiled versions and reporting differences. Verified systems software takes a different approach by defining behavior in a proof assistant and proving that an implementation satisfies that definition.

This thesis follows the third strategy and applies it to the second problem. It stays within the standard Itanium ABI and the existing C++ toolchain, like `abi-compliance-checker` and `abidiff`, and produces post-hoc compatibility verdicts on already-compiled libraries. Unlike those tools, the decision procedure is paired with an explicit specification and a mechanized proof that the two agree. The result is an ABI compatibility checker whose verdicts are computed from a formal definition of compatibility, within the scope set out in Chapter 2.

5 Methodology

This chapter describes the workflow used throughout the thesis. The workflow has three phases. The first extracts ABI-relevant information from compiled shared libraries and converts it into a structured representation suitable for verified comparison. The second formalizes ABI compatibility as a Rocq specification paired with a computable checker, together with a correctness proof, and applies this pattern to all rules in scope. The third phase refines the resulting verdict using the consumer's compiled code as an additional input and filters out breaks that the consumer cannot observe.

The workflow separates extraction, verification, and filtering. The first and third phases are implemented on the unverified side, where ELF, DWARF, and LLVM IR are processed in conventional C++ code. The second phase lives on the verified side, where compatibility rules are machine-checked. The interface between both sides is the structured ABI representation produced by the first phase. This separation makes it clear what is covered by the verified part and keeps the unverified components interchangeable. A different extractor or consumer analysis can be added without changing any proof. Figure 5.1 shows the three phases and where the trust boundary falls. Extraction is unverified C++, verification and filtering run in the Rocq-extracted core, and the consumer's LLVM IR is an optional third input to the filter whose reader is itself unverified.

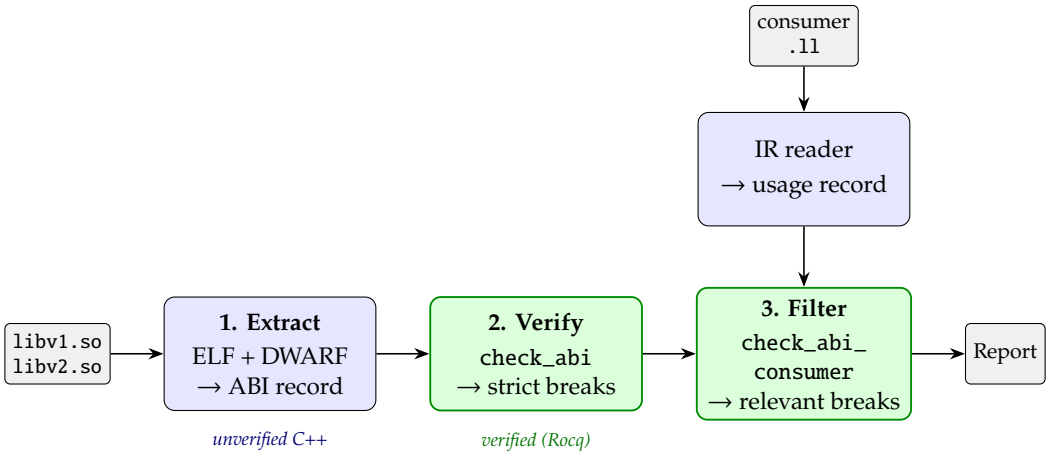


Figure 5.1: The methodology workflow.

The remainder of this chapter describes each phase at a high level.

5.1. ABI Extraction

The input to the workflow is a pair of compiled C++ shared libraries, denoted V_1 and V_2. The output of the extraction phase is a typed intermediate representation of each library's ABI surface, which is

processed by the verified core.

The extractor is custom and not based on existing tooling. Existing tools usually produce ABI summaries for inspection or internal use. The verified core instead needs a representation that matches the formal model in Chapter 6, where every field is explicitly present and nothing is left implicit. A custom extractor makes it easier to control this mapping and to limit the supported subset of C++.

Each library is processed in two passes. The first pass reads the ELF dynamic symbol table and records exported symbols, including their mangled names, kind (function, object, or typeinfo), definition state, and binding strength. The second pass walks the DWARF debug information and reconstructs typed signatures for functions and typed layouts for structs, unions, enums, and classes. Type identity in the representation is based on the source-level type name, since Itanium mangling encodes these names directly in symbols. The result of both passes is combined into a single record, which is passed through a C/OCaml bridge into Rocq.

The extractor is outside the verified scope. A full formal verification of this layer would require modeling ELF, DWARF, and the compiler toolchain in detail, which is out of the scope of this work. As a result, a bug in this layer may misrepresent the binary and lead to an incorrect verdict from the verified core.

5.2. Formalisation of Compatibility Rules

The second phase replaces an informal notion of ABI compatibility with a set of formal rules written in Rocq. The workflow consists of four steps.

Identification. For each ABI component, such as exported symbols, function signatures, object signatures, and type layouts, we identify the invariants required for compatibility. These invariants are derived from the Itanium ABI and from runtime requirements of the dynamic linker and C++ runtime.

Specification. Each invariant is expressed as a proposition in Rocq. For two libraries represented as ABI structures, the proposition has the form

$$R(\mathcal{S}_1, \mathcal{S}_2), \tag{5.1}$$

stating that corresponding ABI elements satisfy the required specification. These propositions define compatibility rather than compute it.

Decision procedure. For each specification, we implement a computable function *check_X* that returns a list of detected violations. An empty list represents compatibility under that rule. Each violation is represented by a structured constructor such as `MissingSymbol` or `StructSizeMismatch`, which carries enough information for reporting.

Correctness proof. Each checker is connected to its specification through a biconditional:

$$\mathit{check\_X}(\mathcal{S}_1, \mathcal{S}_2) = \emptyset \iff R(\mathcal{S}_1, \mathcal{S}_2). \tag{5.2}$$

The forward direction establishes soundness. The reverse establishes completeness. Together they rule out both false positives and false negatives at the rule level.

5.3. Consumer-Aware Refinement

The first two phases compute a strict comparison between two library versions. This is sufficient for library releases, where any potential break is relevant. In integration settings, however, many reported issues do not matter because the consumer never uses the affected parts of the library.

The third phase handles this case. It takes the consumer's compiled LLVM IR as an additional input. The strict checker is run first, producing a list of ABI breaks. This list is then filtered based on what the consumer actually uses, which is extracted from the LLVM IR. The filter is defined in Rocq and proved sound with respect to the strict verdict. It removes exactly the breaks that the relevance predicates mark as unused by the consumer, and it never introduces a break the strict checker did not report.

The consumer extraction layer is unverified and implemented in C++ on top of LLVM IR. As with the ABI extractor, this layer is outside the verified scope. A bug here can produce an incorrect usage record, which would in turn affect the filtered verdict.

6

Formal ABI Model

In this chapter, we define the formal model of ABI compatibility that the rest of this thesis builds on. The model is structured in three distinct layers, matching the main elements exposed by an ABI: the *symbol table* of a shared object, the *signatures* of its exported functions and global variables, and the *layout* of the types that appear at the interface.

Each rule in this chapter is presented in a consistent way. We start with an informal description that motivates the rule from the binary level, then state the formal specification as a mathematical predicate together with its Rocq counterpart. After that, we present a decision procedure in the form of a Rocq definition. A proof of correctness sketches the proof strategy and names the support lemmas, which are used to show the decision procedure complies with the formal specification. Finally, we give a concrete C++ example that demonstrates how the rule can be violated, with supporting evidence extracted from standard binary inspection tools. The complete Rocq proof scripts for every correctness theorem can be found in the `rocq/` directory of the attached thesis artifact. In the main text, we refer to Rocq definitions by name without expanding them.

The model presented here is deliberately *strict*. It treats any change between two library versions that could potentially break a consumer as an incompatibility, even if that break would not be observable in a particular build or runtime setup. Refinements that exploit the knowledge of a specific consumer to narrow the report to breaks that affect it are deferred to the consumer-aware analysis of Chapter 8.

6.1. Notation and Conventions

Throughout this chapter, we distinguish library versions using subscripts. For example, $\mathcal{S}_1$ and $\mathcal{S}_2$ denote the symbol tables of the two versions being compared; $\mathcal{F}_1$ and $\mathcal{F}_2$ their function lists; $\mathcal{O}_1$ and $\mathcal{O}_2$ their object lists; and $\mathcal{T}_1$ and $\mathcal{T}_2$ their type environments.

We use the following notation for tuples such as symbol records, $\sigma = (n, k, d, w)$, with $\sigma.n$ denoting projection onto the field n. Lists are written as $[a_1, \ldots, a_k]$, and membership as $a \in L$. Decidable equality in a domain X is written as $=_X$, but we treat it as a regular equality unless there is ambiguity.

Each compatibility predicate is paired with a decision procedure in Rocq. The two are related by a soundness–completeness theorem of the shape

$$\mathrm{check}_R(\mathcal{X}_1, \mathcal{X}_2) = \emptyset \iff \mathrm{Spec}_R(\mathcal{X}_1, \mathcal{X}_2),$$

where check_R returns the (possibly empty) list of breaks reported under rule R. Throughout the chapter we cite these theorems by their definition names in Rocq.

6.2. Symbol-Level Compatibility

The first layer of compatibility concerns the dynamic symbol table of the shared object. Every dynamically linked ELF object exposes a set of global symbols through its `.dynsym` section, which are

needed for dynamic linking [21]. Each entry corresponds to either a function (`STT_FUNC`) or global object (`STT_OBJECT`), and is annotated with a binding (`STB_GLOBAL` or `STB_WEAK`) and a section index that distinguishes a definition from a reference.

We model a symbol as a four-tuple

$$\sigma := (n,\ k,\ d,\ w) \tag{6.1}$$

with components

- $n \in \mathit{String}$ — the symbol's linkage name as stored in the string table,
- $k \in \{\mathit{Function}, \mathit{Object}, \mathit{TypeInfo}, \mathit{NoType}\}$ — the symbol's kind (the additional *TypeInfo* tag identifies the `RTTI typeinfo` object of a polymorphic class, mangled `_ZTI`⟨`class`⟩ [28]),
- $d \in \{\top, \bot\}$ — whether the symbol is defined in this library ($\top$, section index not `SHN_UNDEF`) or referenced ($\bot$),
- $w \in \{\mathit{Strong}, \mathit{Weak}\}$ — the ELF binding, controlling override semantics during dynamic resolution.

A symbol table is a finite list of such tuples, $\mathcal{S} \in \mathit{list}(\mathit{Symbol})$. The corresponding Rocq definitions are

```
Inductive SymbolType   := Object | Function | NoType | TypeInfo.
Inductive SymbolStrength := Strong | Weak.

Record Symbol := {
  name       : string;
  kind       : SymbolType;
  is_defined : bool;
  strength   : SymbolStrength;
}.
```

where the four record fields correspond directly to the tuple components n, k, d, w, and the two inductives enumerate the value sets of k and w. The remaining chapter refers to the projections by their Rocq names.

Load-order assumption. The rules in this section reason about runtime symbol resolution under the assumption that the consumer's load order is preserved across versions: the set and ordering of dependencies (`DT_NEEDED` entries, `LD_PRELOAD`, search paths) resolved before the library under analysis is unchanged. Reasoning about breaks caused by changes elsewhere in the process namespace would require modeling the entire dependency graph and lies outside our scope.

6.2.1. R1: Symbol Preservation

When a consumer is compiled and linked against version 1 of a shared library, references to library symbols are recorded in the consumer as undefined dynamic symbols in the `.dynsym` table. The actual symbol definitions remain in the shared library. At process startup, the dynamic linker loads the required shared objects and resolves the consumer's undefined symbols by matching them against definitions exported through the libraries' `.dynsym` tables. This process establishes the runtime bindings between consumer references and library implementations. [18]

If the lookup fails for any required name, the program aborts at load time with a `symbol lookup` error. The minimal cross-version invariant is therefore that every name exported by v1 must continue to be exported by v2. If this is not the case, a consumer compiled against v1 cannot start when v2 is dropped in.

We strengthen the preservation rule by additionally requiring symbol names to be unique in v2. Under normal circumstances this constraint is redundant, since the ELF specification requires entries in .dynsym to have unique names [32]. The additional condition is included to guard against malformed inputs, for example when symbol extraction fails or when a build unintentionally introduces duplicate definitions by combining multiple copies of the same source. In such cases, dynamic resolution may silently select one definition while ignoring the other according to symbol table order. The preservation

requirement therefore states that every symbol name present in v1 must correspond to exactly one symbol in v2. Formally,

$$\text{Pres}(\mathcal{S}_1, \mathcal{S}_2) \;:=\; \forall\, \sigma_1,\ \sigma_1 \in \mathcal{S}_1 \;\rightarrow\; \exists!\, \sigma_2,\ \sigma_2 \in \mathcal{S}_2 \;\wedge\; \sigma_1.n = \sigma_2.n. \tag{6.2}$$

The unique-existence quantifier $\exists!$ captures both aspects simultaneously: a matching symbol must exist in v2, and no second symbol with the same name may be present. The corresponding Rocq definition is:

```
Definition symbol_preservation_spec (S1 S2 : list Symbol) : Prop :=
  forall s1, In s1 S1 ->
    exists! s2, In s2 S2 /\ name s1 = name s2.
```

Decision procedure. For each $\sigma_1 \in \mathcal{S}_1$, the checker performs a lookup of $\sigma_1.n$ in $\mathcal{S}_2$ and classifies the result as *NotFound*, *Duplicate*, or *Found*. A missing match yields a `MissingSymbol` break, except for `TypeInfo` symbols, where absence is reported as `RttiChanged`, since a missing type information symbol typically indicates a renamed or removed polymorphic class rather than a missing function. A duplicate match yields `DuplicateSymbol`, while a unique match produces no break.

This uniform three-way classification is reused across symbol-, signature-, and type-level checks. We therefore factor it into a generic preservation procedure `check_preservation_by`, shown at a high level in Algorithm 1 and defined in Rocq below.

Algorithm 1 Generic preservation check (check_preservation_by)

Require: key projection *key*, break constructors *mkMissing*, *mkDuplicate*, lists S_1, S_2
Ensure: list of breaks
breaks ← [ ]
for all $x \in S_1$ **do**
  if *key*(x) has no match in S_2 **then**
    breaks ← *breaks* ++ [*mkMissing*(x)]
  else if *key*(x) has more than one match in S_2 **then**
    breaks ← *breaks* ++ [*mkDuplicate*(x)]
  end if ▷ a unique match yields no break
end for
return *breaks*

```
Fixpoint check_preservation_by {A K Break}
        (key : A -> K)
        (mk_missing mk_duplicate : A -> Break)
        (eq_key  : K -> K -> bool)
        (eq_full : A -> A -> bool)
        (S1 S2 : list A) : list Break :=
  match S1 with
  | [] => []
  | x :: xs =>
      let rest := check_preservation_by ... xs S2 in
      match find_unique_by key eq_key eq_full (key x) S2 with
      | UFNotFound  => mk_missing x :: rest
      | UFDuplicate => mk_duplicate x :: rest
      | UFFound _   => rest
      end
  end.
```

The procedure is parameterised over the element type A, key type K, and break type $Break$, allowing the same procedure to be reused for symbols, function signatures, and type declarations. It iterates over $\mathcal{S}_1$, queries $\mathcal{S}_2$ using `find_unique_by` (defined below), and emits a break only when the lookup is not uniquely successful.

Specialising the key to symbol names yields `check_symbol_preservation`:

```
Definition check_symbol_preservation
  (S1 S2 : list Symbol) : list SymbolBreak :=
  check_preservation_by
    name
    (fun s =>
      match kind s with
      | TypeInfo => RttiChanged (name s)
      | _        => MissingSymbol (name s)
      end)
    (fun s => DuplicateSymbol (name s))
    String.eqb Symbol_eqb
    S1 S2.
```

Proof of correctness. We prove the correctness of the symbol preservation checker via a generic library result, and then instantiate it for symbols. Formally, the per-rule theorem states

$$\mathit{check\_symbol\_preservation}(\mathcal{S}_1, \mathcal{S}_2) = \emptyset \iff \mathrm{Pres}(\mathcal{S}_1, \mathcal{S}_2), \tag{6.3}$$

that is, the checker emits no breaks if and only if every symbol in $\mathcal{S}_1$ has a unique name-matching counterpart in $\mathcal{S}_2$.

Rather than proving this directly, we rely on the correctness theorem `check_preservation_by_correct` for the parameterised procedure `check_preservation_by`. Let $\mathcal{X}_1, \mathcal{X}_2$ denote two arbitrary lists of records, and let π be any key projection. The generic theorem states that the procedure returns no breaks if and only if every element of $\mathcal{X}_1$ has a unique π-matching counterpart in $\mathcal{X}_2$.

Its proof proceeds by induction on $\mathcal{X}_1$. For each element, a helper lookup function `find_unique_by` traverses $\mathcal{X}_2$ and classifies the result as no match, multiple matches, or a unique match. A structured set of auxiliary lemmas characterises the meaning of each outcome in terms of existence and uniqueness in $\mathcal{X}_2$.

The main theorem then follows by case analysis on the lookup result: the absence of a match contradicts existence, multiple matches contradict uniqueness, and a unique match provides the required witness. The inductive hypothesis discharges the remainder of $\mathcal{X}_1$.

The symbol-preservation checker is obtained by instantiating the generic procedure with the symbol name as key, full-record equality as comparison, and the appropriate break constructors. Its correctness then follows from the generic theorem. The proof is completed using standard decidable-equality properties for strings and `Symbol` records:

```
Theorem check_symbol_preservation_correct :
  forall S1 S2,
    check_symbol_preservation S1 S2 = [] <->
    symbol_preservation_spec S1 S2.
Proof.
  intros S1 S2.
  apply (check_preservation_by_correct ...).
  apply String.eqb_eq. apply String.eqb_neq. apply String.eqb_refl.
  apply Symbol_eqb_eq. apply Symbol_eqb_neq. apply Symbol_eqb_refl.
Qed.
```

The full proof script is in `rocq/library.v` and `rocq/Symbols.v` in the attached artifact.

Break example. We illustrate the missing-symbol half of the rule by removing an exported function across versions.

```
// libv1/foo.cc
extern "C" void foo() {}
```

```
// libv2/foo.cc
// (foo removed in v2)
```

```
// app/main.cc
extern "C" void foo();
int main() { foo(); }
```

We use `extern "C"` so that the linkage name on both sides is simply `foo`, sidestepping name mangling for this first rule. The consumer is compiled against `libv1.so`. Substituting `libv2.so` on the runtime search path produces a load-time failure before `main` runs:

```
$ ./app
./app: symbol lookup error: ./app: undefined symbol: foo
```

The cause is visible directly in the libraries' dynamic symbol tables. In v1, `foo` appears as a defined global function:

```
$ readelf -W --dyn-syms libv1.so | grep ' foo$'
   12: 0000000000001119  11 FUNC    GLOBAL DEFAULT   12 foo
```

In v2, the same query returns nothing:

```
$ readelf -W --dyn-syms libv2.so | grep ' foo$'
$
```

This textual difference is precisely the formal fact that $\mathrm{Pres}(\mathcal{S}_1, \mathcal{S}_2)$ rules out: the name `foo` is in $\mathcal{S}_1$ and has no counterpart in $\mathcal{S}_2$. Running the checker on the same pair of libraries surfaces the disagreement as

```
$ veriabi libv1.so libv2.so
  FAIL  Symbols
        MissingSymbol (1):
          - foo
```

which corresponds to the `MissingSymbol` break constructor produced by the *NotFound* branch of `check_symbol_preservation`.

6.2.2. R2: Symbol Kind Consistency

A consumer compiled against v1 generates code based on the kind of each external symbol it uses. A function symbol is invoked through a call instruction, while an object symbol is loaded by address. Reusing the same name in v2 with a different kind silently violates this contract: a v1 consumer's call site jumps into static data, or its load reads bytes from executable code. The contract holds across versions iff every v1 symbol's counterpart in v2 has the same kind. Formally,

$$\mathrm{KindCon}(\mathcal{S}_1, \mathcal{S}_2) \;:=\; \forall\, \sigma_1, \sigma_2,\; \sigma_1 \in \mathcal{S}_1 \;\to\; \sigma_2 \in \mathcal{S}_2 \;\to\; \sigma_1.n = \sigma_2.n \;\to\; \sigma_1.k = \sigma_2.k. \tag{6.4}$$

The corresponding Rocq definition is:

```
Definition symbol_kind_consistency_spec (S1 S2 : list Symbol) : Prop :=
  forall s1 s2, In s1 S1 -> In s2 S2 ->
    name s1 = name s2 -> kind s1 = kind s2.
```

Decision procedure. This rule is the first instance of a recurring pattern: a *consistency check* keyed by one projection (the name) and projecting on a second (the kind). We define a generic checker `check_consistency_by` that captures the pattern, shown at a high level in Algorithm 2 and defined in Rocq below.

Algorithm 2 Generic consistency check (check_consistency_by)

Require: key and value projections *key*, *val*, break constructor *mkBreak*, lists S_1, S_2
Ensure: list of breaks

```
breaks ← [ ]
for all s1 ∈ S1 do
    for all s2 ∈ S2 with key(s2) = key(s1) do
        if val(s1) ≠ val(s2) then
            breaks ← breaks ++ [ mkBreak(s1, s2) ]
        end if
    end for
end for
return breaks
```

```
Fixpoint check_consistency_by {A K V Break}
  (proj_key : A -> K) (eqb_key : K -> K -> bool)
  (proj_val : A -> V) (eqb_val : V -> V -> bool)
  (mk_break : A -> A -> Break)
  (S1 S2 : list A) : list Break :=
match S1 with
| [] => []
| s1 :: st =>
    let rest    := check_consistency_by ... st S2 in
    let matches := find_by proj_key eqb_key (proj_key s1) S2 in
    check_matches_by proj_val eqb_val mk_break s1 matches ++ rest
end.
```

For each $\sigma_1 \in \mathcal{S}_1$ the checker uses two helpers (both shown in the proof below): `find_by` collects the bucket of $\sigma_2 \in \mathcal{S}_2$ that share the key with σ_1, and `check_matches_by` emits a break for every bucket member that disagrees with σ_1 on the value projection. The function is parametrised over the element, key, value, and break types, so the same recursion can be reused for any rule of this shape. Instantiating it with the symbol name as the key and the symbol kind as the value yields `check_symbol_kind_consistency`:

```
Definition check_symbol_kind_consistency
  (S1 S2 : list Symbol) : list SymbolBreak :=
  check_consistency_by
    name String.eqb
    kind SymbolType_eqb
    (fun s1 s2 => KindMismatch (name s1) (kind s1) (kind s2))
    S1 S2.
```

The same factoring is reused for R3 and several signature-level rules without further commentary.

Proof of correctness. We prove the correctness of the symbol-kind-consistency checker via the same generic library result that handles every check of this shape. Formally, the per-rule theorem is

$$\mathit{check\_symbol\_kind\_consistency}(\mathcal{S}_1, \mathcal{S}_2) = \emptyset \iff \mathrm{KindCon}(\mathcal{S}_1, \mathcal{S}_2), \tag{6.5}$$

that is, the checker emits no breaks if and only if every same named v1/v2 symbol pair agrees on the kind projection.

The generic correctness theorem `check_consistency_by_correct` states that, for any two lists $\mathcal{X}_1, \mathcal{X}_2$, any key projection π_k, and any value projection π_v, the procedure returns no breaks if and only if every pair $(x_1, x_2) \in \mathcal{X}_1 \times \mathcal{X}_2$ that agrees on the key also agrees on the value.

Its proof proceeds by induction on $\mathcal{X}_1$. For each element, a helper `find_by` collects the bucket of $\mathcal{X}_2$ elements that share the key, and a second helper `check_matches_by` emits a break for every bucket

member that disagrees on the value. Two auxiliary lemmas bridge between the spec's universal quantifier over $\mathcal{X}_2$ and the checker's iteration over the bucket: one characterises membership in the bucket, the other characterises an empty break list as value agreement on every bucket member.

The main theorem then follows by case analysis on each name matched pair: a name match with value agreement contributes no break, while a name match with value disagreement contributes exactly one break of the corresponding constructor. The inductive hypothesis discharges the remainder of $\mathcal{X}_1$.

The symbol-kind-consistency checker is obtained by instantiating the generic procedure with the symbol name as key and the symbol kind as value. Its correctness then follows from the generic theorem. The proof is completed using standard decidable-equality properties for strings and `SymbolType`:

```
Theorem check_symbol_kind_consistency_correct :
  forall S1 S2,
    check_symbol_kind_consistency S1 S2 = [] <->
    symbol_kind_consistency_spec S1 S2.
Proof.
  intros S1 S2.
  apply check_consistency_by_correct with
    (proj_key := name)         (proj_val := kind)
    (eqb_key := String.eqb)    (eqb_val := SymbolType_eqb) ...
  apply String.eqb_eq.
  apply SymbolType_eqb_eq. apply SymbolType_eqb_neq.
Qed.
```

The full proof script is in `rocq/library.v` and `rocq/Symbols.v` in the attached artifact.

Break example. A function symbol in v1 is reused as a global variable in v2. We trace the failure step by step, from the consumer's compiled call site to the moment of the segfault.

```
// libv1/foo.cc
extern "C" int foo() { return 1; }
```

```
// libv2/foo.cc
extern "C" int foo = 2;  // now an object
```

```
// app/main.cc
extern "C" int foo();
int main() { return foo(); }
```

The consumer was compiled against `libv1.so`, where `foo` was a function. Its emitted code contains a literal `call` through the procedure linkage table:

```
$ objdump -d ./app | grep -A3 '<main>:'
0000000000001139 <main>:
    1139:       55                      push   %rbp
    113a:       48 89 e5                mov    %rsp,%rbp
    113d:       e8 ee fe ff ff          call   1030 <foo@plt>
```

At load time the PLT entry is resolved to whatever address the dynamic linker finds for `foo` in the loaded library. With `libv2.so` loaded, `foo` is no longer in executable code: `readelf` classifies it as an object in `.data`,

```
$ readelf -W --dyn-syms libv2.so | grep 'foo'
    5: 0000000000004008     4 OBJECT  GLOBAL DEFAULT   17 foo
```

and the four bytes at offset `0x4008` are the encoded integer `2`:

```
$ objdump -j .data -s libv2.so | grep '4008'
 4008 02000000
```

The chain is now end-to-end visible. The consumer's `call` instruction jumps to a memory region storing `02 00 00 00`, which the CPU attempts to execute as machine code, and the program aborts before any meaningful work happens:

```
$ ./app
Segmentation fault
```

The kind change recorded in v2's symbol table is the root cause. The columns `FUNC` and `OBJECT` in the `readelf` dump reflect the ELF symbol-type field (`STT_FUNC` versus `STT_OBJECT`) [32], which is exactly what $\sigma.k$ records in our model. The checker surfaces the disagreement as

```
$ veriabi libv1.so libv2.so
  FAIL  Symbols
        KindMismatch (1):
          - foo (expected: Function, got: Object)
```

matching the `KindMismatch` break constructor produced by `check_symbol_kind_consistency`.

6.2.3. R3: Symbol Strength Consistency

The binding strength field $\sigma.w$ of a symbol determines how it participates in dynamic linking when multiple libraries export the same name. Under the *classical linking rule*, a strong definition overrides a weak one across libraries: a strong symbol is the canonical definition, while a weak symbol is a placeholder that yields whenever a strong definition is found in any other loaded library. Reusing a name across versions with a different strength can therefore silently redirect the consumer's reference to a different implementation whenever a third library participates. The runtime impact varies across linker implementations: in modern glibc (version 2.2 and later [18]) symbol resolution follows a *first-found* principle in which strength plays no role, while older or non-glibc linkers may still apply the classical rule. In line with the strict scope set out in the chapter introduction, we treat any strength change as a violation: every v1 symbol's counterpart in v2 must have the same strength. Formally,

$$\mathrm{StrCon}(\mathcal{S}_1, \mathcal{S}_2) \;:=\; \forall\, \sigma_1, \sigma_2,\ \sigma_1 \in \mathcal{S}_1 \;\to\; \sigma_2 \in \mathcal{S}_2 \;\to\; \sigma_1.n = \sigma_2.n \;\to\; \sigma_1.w = \sigma_2.w. \tag{6.6}$$

The corresponding Rocq definition is:

```
Definition symbol_strength_consistency_spec
  (S1 S2 : list Symbol) : Prop :=
  forall s1 s2, In s1 S1 -> In s2 S2 ->
    name s1 = name s2 -> strength s1 = strength s2.
```

Decision procedure. A direct instantiation of `check_consistency_by` with $\pi_k = n$ (project on the name) and $\pi_v = w$ (project on the strength) gives `check_symbol_strength_consistency`. The break constructor wraps the v1 and v2 strength values into a `StrengthMismatch` break tagged with the symbol name:

```
Definition check_symbol_strength_consistency
  (S1 S2 : list Symbol) : list SymbolBreak :=
  check_consistency_by
    name String.eqb
    strength SymbolStrength_eqb
    (fun s1 s2 => StrengthMismatch (name s1) (strength s1) (strength s2))
    S1 S2.
```

The structure is identical to R2 with `kind` / `SymbolType_eqb` / `KindMismatch` swapped for `strength` / `SymbolStrength_eqb` / `StrengthMismatch`.

Proof of correctness. This rule shares its proof structure with R2. The per-rule theorem is

$$\mathit{check\_symbol\_strength\_consistency}(\mathcal{S}_1, \mathcal{S}_2) = \emptyset \iff \mathrm{StrCon}(\mathcal{S}_1, \mathcal{S}_2), \tag{6.7}$$

and follows from `check_consistency_by_correct` with the symbol-strength equality properties in place of the symbol-kind ones:

```
Theorem check_symbol_strength_consistency_correct :
  forall S1 S2,
    check_symbol_strength_consistency S1 S2 = [] <->
    symbol_strength_consistency_spec S1 S2.
Proof.
  intros S1 S2.
  apply check_consistency_by_correct with
    (proj_key := name)          (proj_val := strength)
    (eqb_key := String.eqb)     (eqb_val := SymbolStrength_eqb) ...
  apply String.eqb_eq.
  apply SymbolStrength_eqb_eq. apply SymbolStrength_eqb_neq.
Qed.
```

The full proof is in `rocq/Symbols.v` in the attached artifact.

Break example. A symbol that was strong in v1 is declared weak in v2. The break occurs when a third-party library is loaded alongside the v2 library and also defines the same name as a strong symbol. Under the classical rule that a strong definition overrides a weak one across libraries, the consumer's reference is silently redirected to the third party's definition. On modern glibc the classical rule is disabled by default (resolution is first-found, regardless of strength), but the loader still implements it. By setting the environment variable `LD_DYNAMIC_WEAK=1` [18], the classical behavior gets re-enabled for the running process. We use that switch to reproduce the break on a current system.

The setup uses three libraries defining `foo` and a consumer that prints whichever `foo` resolves at runtime:

```
// libv1/foo.cc
#include <cstdio>
extern "C" void foo() { puts("v1"); }
```

```
// libv2/foo.cc
#include <cstdio>
extern "C" __attribute__((weak)) void foo() { puts("v2"); }
```

```
// libalt/foo.cc
#include <cstdio>
extern "C" void foo() { puts("alt"); }
```

```
// app/main.cc
extern "C" void foo();
int main() { foo(); }
```

`libalt.so` is the third-party library that also exports `foo` as a strong symbol. In the following scenario, the consumer is compiled and linked against `libv1.so` and `libalt.so`, and the v1 to v2 upgrade is simulated by replacing the file at `libv1.so` with v2's content. The link order leaves a clear trace in the binary's `DT_NEEDED` list, which determines the runtime load order:

```
$ readelf -d app | grep NEEDED
 0x0000000000000001 (NEEDED)             Shared library: [./libv1.so]
 0x0000000000000001 (NEEDED)             Shared library: [./libalt.so]
 0x0000000000000001 (NEEDED)             Shared library: [libstdc++.so.6]
 0x0000000000000001 (NEEDED)             Shared library: [libm.so.6]
 0x0000000000000001 (NEEDED)             Shared library: [libgcc_s.so.1]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
```

Because `libv1.so` precedes `libalt.so`, the consumer's load order is fixed accordingly, regardless of the implementation currently provided by `libv1.so`. The strength flip introduced by v2 is visible in `nm`:

```
$ nm -D libv1.so | grep foo
0000000000001109 T foo
$ nm -D libv2.so | grep foo
0000000000001109 W foo
```

The `T` marks a strong symbol in the text section and the `W` marks a weak one [32]. This is exactly the $\sigma.w$ component of our model. The break manifests once `libv1.so` is overwritten with v2's content. With the file replaced, the consumer's `DT_NEEDED` list is unchanged, but the symbol at the `libv1.so` slot is now weak:

```
$ LD_LIBRARY_PATH=. ./app
v2
$ LD_DYNAMIC_WEAK=1 LD_LIBRARY_PATH=. ./app
alt
```

Under default first-found resolution, the consumer binds to the now-weak `foo` at the `libv1.so` slot and prints `v2`. Under classical resolution, the loader walks past the weak `foo` and binds to `libalt`'s strong `foo` instead, printing `alt`.

The strength change in v2 is what makes this divergence possible. If v2 had kept `foo` strong, both definitions would still coexist in the same load order, but the classical rule would have no weak symbol to override and both modes would resolve to whichever object appeared first. By introducing the weak attribute on `foo`, v2 opens the door for `libalt`'s strong definition to take over when `LD_DYNAMIC_WEAK` is set.

The checker flags the change either way:

```
$ veriabi libv1.so libv2.so
  FAIL  Symbols
        StrengthMismatch (1):
          - foo (expected: Strong, got: Weak)
```

matching the `StrengthMismatch` break constructor produced by `check_symbol_strength_consistency`.

6.2.4. R4: Symbol Definedness Monotonicity

The fourth rule enforces that a symbol that was defined in v1 remains defined in v2. Unlike the previous rules, this is a one-way constraint. The reverse direction, in which a symbol that was undefined in v1 gains a definition in v2, is treated as non-breaking. We therefore require monotonicity rather than full preservation.

The relaxation rests on the assumption that v2's new definition coincides with whatever provider the consumer's load order would resolve to before the upgrade. This is the typical case in practice: either no other loaded object was exporting the name (in which case the v1 reference would have failed at load time anyway, so any v2 definition can only help), or v2 has absorbed code from a sister library that previously supplied the symbol with the same semantics. The assumption fails when a v1-imported name resolves to a third-party provider and v2 silently overrides it with different semantics. In such cases the strict layer is too coarse to detect the break, and the consumer-aware analysis of Chapter 8 is the natural setting. Formally,

$$\mathrm{DefMono}(\mathcal{S}_1, \mathcal{S}_2) \;:=\; \forall\, \sigma_1, \sigma_2,\; \sigma_1 \in \mathcal{S}_1 \;\to\; \sigma_2 \in \mathcal{S}_2 \;\to\; \sigma_1.n = \sigma_2.n \;\to\; \sigma_1.d = \top \;\to\; \sigma_2.d = \top. \quad (6.8)$$

The corresponding Rocq definition is:

```
Definition symbol_monotonicity_spec (S1 S2 : list Symbol) : Prop :=
  forall s1 s2, In s1 S1 -> In s2 S2 ->
    name s1 = name s2 -> is_defined s1 = true -> is_defined s2 = true.
```

Decision procedure. The asymmetry of DefMono does not fit the value-equality shape of `check_consistency_by`, so the checker is implemented directly, as shown in Algorithm 3. For each $s_1 \in \mathcal{S}_1$ and each $s_2 \in \mathcal{S}_2$ that shares the name with s_1, it emits a `DefinitionMismatch` break if s_1 is defined and s_2 is not.

Algorithm 3 Definedness monotonicity check (check_symbol_monotonicity)

Require: symbol lists $\mathcal{S}_1, \mathcal{S}_2$
Ensure: list of breaks
1: *breaks* $\leftarrow$ [ ]
2: **for all** $s_1 \in \mathcal{S}_1$ **do**
3: **for all** $s_2 \in \mathcal{S}_2$ with *name*(s_2) = *name*(s_1) **do**
4: **if** s_1 is defined **and** s_2 is undefined **then**
5: *breaks* $\leftarrow$ *breaks* $+\!\!+$ [DefinitionMismatch(*name*(s_1))]
6: **end if**
7: **end for**
8: **end for**
9: **return** *breaks*

The Rocq definition is:

```
Definition check_symbol_monotonicity
  (S1 S2 : list Symbol) : list SymbolBreak :=
  flat_map
    (fun s1 =>
       flat_map
         (fun s2 =>
            if String.eqb (name s1) (name s2)
               && is_defined s1
               && negb (is_defined s2)
            then [DefinitionMismatch (name s1) true false]
            else [])
         S2)
    S1.
```

Proof of correctness. Unlike R1–R3, this rule does not reuse the generic consistency framework. The property is asymmetric: it constrains v2 only when the corresponding v1 symbol is defined, whereas `check_consistency_by` assumes symmetric value agreement. Formally, the per-rule theorem states

$$\mathit{check\_symbol\_monotonicity}(\mathcal{S}_1, \mathcal{S}_2) = \emptyset \iff \mathrm{DefMono}(\mathcal{S}_1, \mathcal{S}_2), \tag{6.9}$$

that is, the checker emits no breaks if and only if no v1-defined symbol has an undefined same named counterpart in v2. The corresponding Rocq theorem is:

```
Theorem check_symbol_monotonicity_correct :
  forall S1 S2,
    check_symbol_monotonicity S1 S2 = [] <->
    symbol_monotonicity_spec S1 S2.
```

The checker enumerates every pair $(\sigma_1, \sigma_2) \in \mathcal{S}_1 \times \mathcal{S}_2$ and emits a break exactly when the symbols share the same name, σ_1 is defined, and σ_2 is undefined. Auxiliary lemmas about the underlying list traversal establish that these conditions precisely characterise break emission. The main theorem then follows in both directions: an empty checker output excludes every violating pair and therefore yields DefMono, while DefMono rules out the break-emission condition for every pair, so the checker emits nothing. The full proof is in `rocq/Symbols.v` in the attached artifact.

Break example. A previously defined symbol becomes a forward reference only. The consumer compiled against v1 fails to resolve the symbol at load time, just as if it had been removed.

```
// libv1/foo.cc
extern "C" void foo() {}
```

```
// libv2/foo.cc
extern "C" void foo();              // declaration only
extern "C" void use_foo() { foo(); } // forces foo into v2's dynsym as UND
```

The definedness change is visible in the section-index column of `.dynsym`. In v1, `foo` resides in section 12 (the `.text` section):

```
$ readelf -W --dyn-syms libv1.so | grep 'foo'
   12: 0000000000001119    1 FUNC    GLOBAL DEFAULT   12 foo
```

In v2 it resides in `UND`, the special index for undefined symbols, and its symbol type is recorded as `NOTYPE` because the linker does not propagate function/object type information for undefined references:

```
$ readelf -W --dyn-syms libv2.so | grep 'foo'
    4: 0000000000000000   0 NOTYPE  GLOBAL DEFAULT  UND foo
```

The `UND` marker corresponds to a section index of `SHN_UNDEF` [32], which negates $\sigma.d = \top$ in our model. Substituting v2 at runtime yields the same load-time failure as removal:

```
$ ./app
./app: symbol lookup error: ./app: undefined symbol: foo
```

The checker reports two breaks:

```
$ veriabi libv1.so libv2.so
  FAIL  Symbols
        KindMismatch (1):
          - foo (expected: Function, got: NoType)
        DefinitionMismatch (1):
          - foo (expected: true, got: false)
```

Both are real symbol-table breaks. The dynsym genuinely records v1's defined `foo` as `Function` and v2's undefined `foo` as `NoType`, so the model correctly reports a kind change in addition to the definedness change. The `DefinitionMismatch` is the one produced by `check_symbol_monotonicity` and is the violation this rule directly captures. The `KindMismatch` is produced by R2 and reflects the corresponding kind change in the dynsym. A single source-level edit (removing the definition) therefore decomposes into two orthogonal symbol-table violations, each caught by its own rule, illustrating how the layered checks compose.

6.2.5. Combined Symbol-Level Compatibility

The four invariants combine into a single specification:

$$\begin{aligned} \text{SymbolsSpec}(\mathcal{S}_1, \mathcal{S}_2) \;:=\; & \text{Pres}(\mathcal{S}_1, \mathcal{S}_2) \\ & \wedge\, \text{KindCon}(\mathcal{S}_1, \mathcal{S}_2) \\ & \wedge\, \text{StrCon}(\mathcal{S}_1, \mathcal{S}_2) \\ & \wedge\, \text{DefMono}(\mathcal{S}_1, \mathcal{S}_2) \end{aligned} \tag{6.10}$$

The corresponding Rocq definition conjoins the four sub-specs:

```
Definition SymbolsSpec (S1 S2 : list Symbol) : Prop :=
  symbol_kind_consistency_spec S1 S2 /\
  symbol_preservation_spec S1 S2 /\
  symbol_strength_consistency_spec S1 S2 /\
  symbol_monotonicity_spec S1 S2.
```

Decision procedure. The combined checker concatenates the four individual checkers:

```
Definition check_symbols (S1 S2 : list Symbol) : list SymbolBreak :=
  check_symbol_kind_consistency S1 S2 ++
  check_symbol_preservation S1 S2 ++
  check_symbol_strength_consistency S1 S2 ++
  check_symbol_monotonicity S1 S2.
```

The list-based `check_symbols` scans $\mathcal{S}_2$ for every element of $\mathcal{S}_1$ via each per-rule checker, giving $O(n^2)$ time in the total number of symbols. An equivalent $O(n \log n)$ implementation `check_symbols_fast` first builds an `FMap` from names to symbol lists and then dispatches each $\sigma_1 \in \mathcal{S}_1$ directly to the bucket of v2 symbols that share its name, performing all four per-rule checks on the bucket alone. It is proved correct against the same specification. On production libraries with thousands of exported symbols the list-based variant is too slow in practice, and `veriabi` invokes `check_symbols_fast` by default.

Proof of correctness. Formally, the combined theorem is

$$\mathit{check\_symbols}(\mathcal{S}_1, \mathcal{S}_2) = \emptyset \iff \mathrm{SymbolsSpec}(\mathcal{S}_1, \mathcal{S}_2), \tag{6.11}$$

that is, the combined checker emits no breaks if and only if all four sub-specs (preservation, kind consistency, strength consistency, definedness monotonicity) hold simultaneously. In Rocq:

```
Theorem check_symbols_correct :
  forall S1 S2,
    check_symbols S1 S2 = [] <-> SymbolsSpec S1 S2.
```

The proof decomposes the concatenated output via `app_eq_nil` into four empty sub-outputs, one per rule. Each sub-output is converted into its sub-spec by the corresponding per-rule correctness theorem. The four sub-specs reassemble into `SymbolsSpec`. The backward direction reverses the process: each conjunct of `SymbolsSpec` yields an empty sub-checker, and the four empty pieces concatenate to the empty list.

The full proof is in `rocq/Symbols.v` in the attached artifact.

6.3. Signature-Level Compatibility

The second layer of compatibility concerns the signatures of exported functions and global objects, extracted from the binaries. The symbol table determines *what* the library exports, and the signatures determine *how* those exports are invoked or read. Whether a signature change is visible at the symbol level depends on name mangling: under the Itanium C++ ABI [28], function parameter types are encoded into the mangled name and a parameter-list change therefore is caught at the symbol compatibility level by R1. However, return types, `extern "C"` declarations, and global object types are not encoded in the symbol name. The signature-level rules cover precisely these gaps, where the symbol layer cannot see a change that nonetheless breaks the calling convention or the typed memory access on the consumer side.

Type values used by both functions and objects are not raw type names but a lightweight signature type $SigType$ that encodes the pointer-indirection structure relevant to the calling convention,

$$\tau \ ::= \ \mathrm{Base}(s) \ \mid \ \mathrm{Ptr}(\tau), \tag{6.12}$$

with constructors

- Base(s) — a leaf type, where $s \in \mathit{String}$ is the full qualified name of the underlying type (its identifier together with its namespace and class qualifiers, such as `std::string` or `Outer::Inner`). It may be a primitive (e.g., `int`, `double`) or the user-given name of an aggregate type, enum, or typedef declared in the library (e.g., `Foo`). The same string serves as the lookup key into the type environments $\mathcal{T}_1, \mathcal{T}_2$ of Section 6.4, where the full layout of the named type is recorded.
- Ptr(τ) — one level of pointer indirection around an inner signature type τ. Both C++ pointers and references (lvalue `T&` and rvalue `T&&`) map to Ptr(τ), since references are passed and returned in the same registers and stack slots as pointers at the binary level [28].

We model an exported function as

$$\varphi \;:=\; (n,\ \tau_r,\ [\tau_1, \ldots, \tau_k],\ v), \tag{6.13}$$

with components

- $n \in \mathit{String}$ — the function's linkage name as stored in the dynamic symbol table,
- $\tau_r \in \mathit{SigType}$ — the return type,
- $[\tau_1, \ldots, \tau_k] \in \mathit{list}(\mathit{SigType})$ — the ordered list of parameter types,
- $v \in \{\top, \bot\}$ — whether the function is variadic ($\top$) or fixed-arity ($\bot$).

A global object is modelled symmetrically as

$$\omega \;:=\; (n,\ \tau), \tag{6.14}$$

with components

- $n \in \mathit{String}$ — the object's linkage name,
- $\tau \in \mathit{SigType}$ — the object's declared type.

The corresponding Rocq definitions are:

```
Inductive SigType :=
  | STBase    (name  : string)
  | STPointer (inner : SigType).

Record Function := {
  fname       : string;
  return_sig  : SigType;
  param_sigs  : list SigType;
  is_variadic : bool;
}.

Record Object := {
  oname    : string;
  type_sig : SigType;
}.
```

Cross-name signature compatibility. A naive compatibility predicate would require $\tau_1 = \tau_2$ as raw strings. This is too strict in practice. An `extern "C"` function declared as `void f(struct Foo*)` is not affected by a rename of `Foo`, as long as the layout is unchanged. The reason behind this is that the consumer only ever passes pointers and never inspects the layout itself. The same holds for a mangled C++ function `Foo* get()` whose return type is renamed, because the Itanium mangling of `get` does not encode its return type and the symbol layer therefore sees no change. We therefore lift the naive predicate to an *environment-aware* compatibility relation $\mathrm{compat}_\tau(\mathcal{T}_1, \mathcal{T}_2, f, \tau_1, \tau_2)$, parameterised by two type environments $\mathcal{T}_1, \mathcal{T}_2$ (introduced in Section 6.4, which map qualified type names to their structural definitions) and a recursion fuel $f \in \mathbb{N}$. Formally:

$$\mathrm{compat}_\tau(\mathcal{T}_1, \mathcal{T}_2, f, \tau_1, \tau_2) \;:=\; \begin{cases} \mathrm{compat}_m(\mathcal{T}_1, \mathcal{T}_2, f, s_1, s_2) & \tau_1 = \mathrm{Base}(s_1),\ \tau_2 = \mathrm{Base}(s_2) \\ \mathrm{name}(\tau_1') = \mathrm{name}(\tau_2') \vee & \tau_1 = \mathrm{Ptr}(\tau_1'),\ \tau_2 = \mathrm{Ptr}(\tau_2') \\ \quad \mathrm{compat}_\tau(\mathcal{T}_1, \mathcal{T}_2, f, \tau_1', \tau_2') & \\ \bot & \text{otherwise} \end{cases} \tag{6.15}$$

where $\mathrm{name}(\tau)$ is the printed name of a signature type, defined inductively as

$$\mathrm{name}(\tau) \;:=\; \begin{cases} s & \tau = \mathrm{Base}(s) \\ \mathrm{name}(\tau') \mathbin{+\!\!+} \texttt{"*"} & \tau = \mathrm{Ptr}(\tau'). \end{cases} \tag{6.16}$$

Three design choices in this definition need some clarification.

Role of the environments $\mathcal{T}_1, \mathcal{T}_2$. These are the type environments of v1 and v2 introduced in Section 6.4: each maps a qualified type name (such as `std::string` or `Outer::Inner`) to that type's structural definition (members, base classes, vtable, size, alignment). The base case of compat_τ resolves the two named types s_1, s_2 through $\text{lookup}(\mathcal{T}_1, s_1)$ and $\text{lookup}(\mathcal{T}_2, s_2)$ and defers to the layout-level relation compat_m to compare the resulting layouts. The environments must therefore be threaded through every recursive call so the base case can perform the lookups.

Role of the fuel f. The fuel bounds the recursion depth so that the relation terminates on cyclic type graphs. A struct may reference itself indirectly through a pointer chain (for instance, a linked-list node whose `next` field is a pointer to the same struct), and an unbounded recursion would loop forever. The fuel is consumed on every pointer indirection. The correctness theorems are stated with $f = |\mathcal{T}_1|$, the length of the v1 type environment, which is an upper bound on the number of distinct types reachable from any signature and therefore guarantees that no compatible pair is ever rejected for lack of fuel. At runtime, however, the fuel is a tunable parameter of `veriabi`: production libraries rarely have deeply nested pointer chains, so a much smaller value (for instance $f = 10$ for our largest evaluation library) is enough to accept all genuine compatibilities while substantially reducing checker runtime. The trade-off is discussed in the evaluation chapter.

The disjunction in the pointer case. At the signature layer, the only question for a pointer parameter is whether the consumer's pointer-passing convention remains valid. Pointers occupy the same register or stack slot regardless of pointee layout, so the actual layout check belongs to the type layer (R3, Section 6.4). The two branches of the disjunction handle two distinct rename scenarios. The name-equality branch accepts when pointee names are identical (`Foo*` versus `Foo*`). Any layout change to `Foo` between v1 and v2 is caught at the type layer, where R3 pairs `Foo` with itself by name and compares the two layouts. The signature layer must not double-report this case, so it accepts on name match alone. The structural branch handles renames: `Foo*` versus `FooV2*` where v2 has renamed `Foo` to `FooV2`. The type layer cannot pair these by name and would treat them as unrelated, so the structural branch explicitly calls $\text{compat}_m(s_1, s_2)$ on the two differently-named pointees to check whether they share an underlying layout. Together, the two branches accept exactly those pointer pairs where the consumer's pointer-passing code stays valid: identical names (deferring layout to the type layer) or different names with structurally compatible layouts. The corresponding Rocq definition mirrors the case analysis:

```
Fixpoint sig_type_compat (env1 env2 : list TypeDef) (fuel : nat)
    (st1 st2 : SigType) : Prop :=
  match st1, st2 with
  | STBase t1,    STBase t2    => member_type_compat env1 env2 fuel t1 t2
  | STPointer s1, STPointer s2 =>
      sig_type_name s1 = sig_type_name s2 \/
      sig_type_compat env1 env2 fuel s1 s2
  | _,            _            => False
  end.
```

The executable checker `check_sig_compat` mirrors the structure of `sig_type_compat`, as shown in Algorithm 4. The base case delegates to the layout-level compatibility of the two named types, while the pointer case accepts either identical printed names or recursively compatible pointees.

Algorithm 4 Generic signature-type compatibility (check_sig_compat)

function check_sig_compat($\mathcal{T}_1, \mathcal{T}_2, f, st_1, st_2$)
  if $st_1 = \text{Base}(t_1)$ **and** $st_2 = \text{Base}(t_2)$ **then**
    return $\text{compat}_m(\mathcal{T}_1, \mathcal{T}_2, f, t_1, t_2)$ ▷ defer to the layout layer
  else if $st_1 = \text{Ptr}(p_1)$ **and** $st_2 = \text{Ptr}(p_2)$ **then**
    if $name(p_1) = name(p_2)$ **then return true** ▷ pointer passing unchanged
    end if
    return check_sig_compat($\mathcal{T}_1, \mathcal{T}_2, f, p_1, p_2$) ▷ renamed pointee
  else return false ▷ base versus pointer
  end if
end function

The Rocq definition is:

```
Fixpoint check_sig_compat (env1 env2 : list TypeDef) (fuel : nat)
    (st1 st2 : SigType) : bool :=
  match st1, st2 with
  | STBase t1,    STBase t2    =>
      match check_member_type_env env1 env2 fuel "" "" t1 t2 with
      | []     => true
      | _ :: _ => false
      end
  | STPointer s1, STPointer s2 =>
      if String.eqb (sig_type_name s1) (sig_type_name s2) then true
      else check_sig_compat env1 env2 fuel s1 s2
  | _,            _            => false
  end.
```

The base case calls the layout-level checker `check_member_type_env` with empty struct and member names (the strings are used only for naming breaks in error reports) and accepts the pair if and only if no break is emitted. The pointer case implements the disjunction of the propositional relation. It returns `true` immediately on a name match and otherwise recurses on the pointee types. Mismatched constructors return `false`. The support lemma `check_sig_compat_correct` states that the boolean checker and the propositional relation are equivalent:

$$\begin{aligned} &\mathit{check\_sig\_compat}(\mathcal{T}_1, \mathcal{T}_2, f, \tau_1, \tau_2) = \top \\ &\iff \text{compat}_\tau(\mathcal{T}_1, \mathcal{T}_2, f, \tau_1, \tau_2). \end{aligned} \tag{6.17}$$

The corresponding Rocq theorem is:

```
Theorem check_sig_compat_correct :
  forall env1 env2 fuel st1 st2,
    check_sig_compat env1 env2 fuel st1 st2 = true <->
    sig_type_compat env1 env2 fuel st1 st2.
```

The correctness of `check_sig_compat` is proved by structural induction on the pair of signature types (τ_1, τ_2).

In the base case both types are of the form `STBase`, and the result follows directly from the layout-level correctness theorem `check_member_type_env_correct` (Section 6.4), which relates the boolean layout comparison to the propositional `member_type_compat` relation.

In the inductive case for pointers, the proof proceeds by analyzing the two possible outcomes of the name comparison. If the names coincide, the left disjunct of `sig_type_compat` applies directly and the result follows. Otherwise, the algorithm recurses on the pointee types, and the inductive hypothesis yields the desired equivalence.

All mixed constructor cases (one base and one pointer) are impossible under the specification and therefore reduce to $\bot$ on both sides of the equivalence.

The full proof is in `rocq/Signatures.v` in the attached artifact. The per-rule proofs of R1, R2, and R4 invoke this lemma whenever they need to recover the propositional compatibility from a boolean checker result

6.3.1. R1: Return-Type Compatibility

The first signature-level rule enforces that the return type of a function exported under the same name in both versions remains compatible. A change in return type, even one that is implicit-convertible at the source level, can change the calling convention: on x86_64 SysV [28], integer returns smaller than 64 bits use a different register width than 64-bit returns, and floating-point returns use a separate register entirely. The contract holds across versions if and only if every v1 function's same named counterpart in v2 has a compatible return type. Formally:

$$\begin{aligned}\text{RetCompat}(\mathcal{T}_1, \mathcal{T}_2, f, \mathcal{F}_1, \mathcal{F}_2) \;:=\; \forall\, \varphi_1 \in \mathcal{F}_1,\ \varphi_2 \in \mathcal{F}_2,\\ \varphi_1.n = \varphi_2.n \;\rightarrow\; \text{compat}_\tau(\mathcal{T}_1, \mathcal{T}_2, f, \varphi_1.\tau_r,\ \varphi_2.\tau_r).\end{aligned} \tag{6.18}$$

The corresponding Rocq definition is:

```
Definition function_rt_compat_spec (env1 env2 : list TypeDef) (fuel : nat)
    (F1 F2 : list Function) : Prop :=
  forall f1 f2, In f1 F1 -> In f2 F2 ->
    fname f1 = fname f2 ->
    sig_type_compat env1 env2 fuel (return_sig f1) (return_sig f2).
```

Decision procedure. For each pair (φ_1, φ_2) sharing a name, the checker compares the return types under `check_sig_compat`. When the comparison fails, the break constructor distinguishes whether the printed return-type names are equal (`ReturnLayoutChanged`, the name was kept but the underlying layout drifted) or differ (`ReturnTypeMismatch`, the return type was renamed). Algorithm 5 shows the check.

Algorithm 5 Return-type check (check_function_rt_consistency)

Require: type environments $\mathcal{T}_1, \mathcal{T}_2$, fuel f, function lists $\mathcal{F}_1, \mathcal{F}_2$
Ensure: list of function breaks

breaks ← [ ]
for all $(f_1, f_2) \in \mathcal{F}_1 \times \mathcal{F}_2$ with $name(f_1) = name(f_2)$ **do**
  if not check_sig_compat($\mathcal{T}_1, \mathcal{T}_2, f, ret(f_1), ret(f_2)$) **then**
    if the printed names of $ret(f_1)$ and $ret(f_2)$ are equal **then**
      append ReturnLayoutChanged($name(f_1)$) ▷ name kept, layout drifted
    else
      append ReturnTypeMismatch($name(f_1)$) ▷ return type renamed
    end if
  end if
end for
return *breaks*

In Rocq:

```
Definition check_function_rt_consistency (env1 env2 : list TypeDef) (fuel
   : nat)
    (F1 F2 : list Function) : list FunctionBreak :=
  flat_map (fun f1 =>
    flat_map (fun f2 =>
      if String.eqb (fname f1) (fname f2) then
```

```
        if check_sig_compat env1 env2 fuel (return_sig f1) (return_sig f2)
            then []
        else let n1 := sig_type_name (return_sig f1) in
             let n2 := sig_type_name (return_sig f2) in
             if String.eqb n1 n2
             then [ReturnLayoutChanged (fname f1) n1]
             else [ReturnTypeMismatch  (fname f1) n1 n2]
      else []
    ) F2
  ) F1.
```

Proof of correctness. Formally, the per-rule theorem states

$$\begin{aligned} &\mathit{check\_function\_rt\_consistency}(\mathcal{T}_1, \mathcal{T}_2, f, \mathcal{F}_1, \mathcal{F}_2) = \emptyset \\ &\quad\Longleftrightarrow\ \mathrm{RetCompat}(\mathcal{T}_1, \mathcal{T}_2, f, \mathcal{F}_1, \mathcal{F}_2), \end{aligned} \tag{6.19}$$

that is, the checker emits no breaks if and only if every same named v1/v2 function pair has compatible return types under compat_τ. The corresponding Rocq theorem is:

```
Theorem check_function_rt_consistency_correct :
  forall env1 env2 fuel F1 F2,
    check_function_rt_consistency env1 env2 fuel F1 F2 = [] <->
    function_rt_compat_spec env1 env2 fuel F1 F2.
```

The proof reduces the nested `flat_map` to a per-pair statement via `flat_map_empty`, then case splits on the name comparison for each pair (φ_1, φ_2). Pairs with different names are skipped by construction. For same named pairs, the body is empty exactly when `check_sig_compat` accepts the return types. `check_sig_compat_correct` converts between the boolean result and the propositional compat_τ in both directions. The full proof is in `rocq/Signatures.v` in the attached artifact.

Break example. A return-type change where the value the function returns is conceptually unchanged but the calling convention shifts to an entirely different register class:

```
// v1/temp.cc
int read_temperature() { return 25; }
```

```
// v2/temp.cc, same value changed to double for precision
double read_temperature() { return 25.0; }
```

```
// app/main.cc
#include <iostream>
int read_temperature();
int main() {
  int t = read_temperature();
  std::cout << "temperature = " << t << "\n";
  return t;
}
```

The Itanium mangling rule encodes a free function's name and its parameter types into the symbol but deliberately omits the return type, so both libraries export the same mangled symbol `_Z16read_temperaturev` despite the change from `int` to `double`:

```
$ nm --dynamic v1/libtemp.so | grep read_temperature
00000000000010f9 T _Z16read_temperaturev
$ nm --dynamic v2/libtemp.so | grep read_temperature
00000000000010f9 T _Z16read_temperaturev
```

The symbol layer therefore sees no break and the consumer continues to bind. The two implementations, however, use disjoint return-value registers. v1 places the `int` in `%eax`, the integer return register:

```
$ objdump -d --demangle v1/libtemp.so | grep -A4 'read_temperature()>:'
00000000000010f9 <read_temperature()>:
    10f9:       55                      push   %rbp
    10fa:       48 89 e5                mov    %rsp,%rbp
    10fd:       b8 19 00 00 00          mov    $0x19,%eax
    1102:       5d                      pop    %rbp
```

v2 places the `double` in `%xmm0`, the vector return register, and never touches `%eax`:

```
$ objdump -d --demangle v2/libtemp.so | grep -A4 'read_temperature()>:'
00000000000010f9 <read_temperature()>:
    10f9:       55                      push   %rbp
    10fa:       48 89 e5                mov    %rsp,%rbp
    10fd:       f2 0f 10 05 fb 0e 00    movsd  0xefb(%rip),%xmm0
    1104:       00
```

The consumer was compiled against v1's `int`-returning declaration, so its emitted call site reads `%eax` right after the call and stores it into the local `t`:

```
$ objdump -d --demangle ./app/app | grep -A8 '<main>:'
0000000000001169 <main>:
    1169:       55                      push   %rbp
    116a:       48 89 e5                mov    %rsp,%rbp
    116d:       48 83 ec 10             sub    $0x10,%rsp
    1171:       e8 ba fe ff ff          call   1030 <read_temperature()@plt>
    1176:       89 45 fc                mov    %eax,-0x4(%rbp)
    1179:       48 8d 05 84 0e 00 00    lea    0xe84(%rip),%rax        # 2004 <
        _IO_stdin_used+0x4>
    1180:       48 89 c6                mov    %rax,%rsi
    1183:       48 8d 05 b6 2e 00 00    lea    0x2eb6(%rip),%rax        # 4040 <
        std::cout@GLIBCXX_3.4>
```

Loading v1 leaves `%eax = 25` after the call and the consumer prints the correct value. Loading v2 leaves `%eax` untouched, so the consumer picks up whatever stale data the runtime had left there:

```
$ LD_LIBRARY_PATH=v1 ./app/app; echo "exit=$?"
temperature = 25
exit=25
$ LD_LIBRARY_PATH=v2 ./app/app; echo "exit=$?"
temperature = -1798090423
exit=73
```

The garbage integer `-1798090423` bears no relation to the value v2 actually computed. It is leftover state in `%eax` that v2's `read_temperature` never overwrote, since the `double` result was placed in `%xmm0`. The masked exit code `73` is the unsigned low byte of the same garbage (`0x49`). The checker surfaces the disagreement as

```
$ veriabi v1/libtemp.so v2/libtemp.so
  FAIL  Function signatures
        ReturnTypeMismatch (1):
          - read_temperature (expected: int, got: double)
```

matching the `ReturnTypeMismatch` constructor emitted by `check_function_rt_consistency`.

6.3.2. R2: Parameter-List Compatibility

The second signature-level rule enforces that the parameter list of a function preserves both its length and the type of each entry. A length mismatch is a *count* break. A type mismatch at index i is a *type*

break at that index. Either kind produces a register and stack layout that disagrees between caller and callee. Formally,

$$\begin{aligned}\text{ParamCompat}(\mathcal{T}_1, \mathcal{T}_2, f, \mathcal{F}_1, \mathcal{F}_2) \;:=\; &\forall\, \varphi_1 \in \mathcal{F}_1,\ \varphi_2 \in \mathcal{F}_2,\ \varphi_1.n = \varphi_2.n \;\rightarrow \\ &|\varphi_1.\bar{\tau}| = |\varphi_2.\bar{\tau}| \;\wedge\; \forall\, i, \\ &\quad \text{compat}_\tau(\mathcal{T}_1, \mathcal{T}_2, f, \varphi_1.\bar{\tau}[i],\ \varphi_2.\bar{\tau}[i]).\end{aligned} \tag{6.20}$$

The two conjuncts capture, respectively, the length-equality and the per-position compatibility of the two parameter lists. The corresponding Rocq definition uses `List.Forall2` from the standard library, which lifts a binary predicate over two equally long lists and packages both conjuncts in a single quantifier:

```
Definition function_param_sigs_compat_spec (env1 env2 : list TypeDef) (
    fuel : nat)
    (F1 F2 : list Function) : Prop :=
  forall f1 f2, In f1 F1 -> In f2 F2 ->
    fname f1 = fname f2 ->
    List.Forall2 (sig_type_compat env1 env2 fuel) (param_sigs f1) (
        param_sigs f2).
```

Decision procedure. The checker compares the two parameter lists element by element through the helper `check_param_types`. On length disagreement it emits a `ParamCountMismatch` reporting the v1 and v2 lengths. On type disagreement at position i it emits either a `ParamTypeMismatch` (when the printed names differ) or a `ParamLayoutChanged` (when the names match but the layout has drifted under the env-aware comparison). The top-level checker iterates over each name matched function pair and applies this helper, as Algorithm 6 shows.

Algorithm 6 Parameter-list check (check_function_param_types_consistency)

Require: type environments $\mathcal{T}_1, \mathcal{T}_2$, fuel f, function lists $\mathcal{F}_1, \mathcal{F}_2$
Ensure: list of function breaks

```
breaks ← [ ]
for all (f1, f2) ∈ F1 × F2 with name(f1) = name(f2) do
    breaks ← breaks ++ CheckParamTypes(name(f1), T1, T2, f, params(f1), params(f2))
end for
return breaks
```

In Rocq:

```
Definition check_function_param_types_consistency
    (env1 env2 : list TypeDef) (fuel : nat)
    (F1 F2 : list Function) : list FunctionBreak :=
  flat_map (fun f1 =>
    flat_map (fun f2 =>
      if String.eqb (fname f1) (fname f2)
      then check_param_types (fname f1) env1 env2 fuel
             (param_sigs f1) (param_sigs f2) 0
      else []
    ) F2
  ) F1.
```

Proof of correctness. Formally, the per-rule theorem states

$$\begin{aligned}&\mathit{check\_function\_param\_types\_consistency}(\mathcal{T}_1, \mathcal{T}_2, f, \mathcal{F}_1, \mathcal{F}_2) = \emptyset \\ &\quad\Longleftrightarrow\ \text{ParamCompat}(\mathcal{T}_1, \mathcal{T}_2, f, \mathcal{F}_1, \mathcal{F}_2),\end{aligned} \tag{6.21}$$

that is, the checker emits no breaks if and only if every same named v1/v2 function pair has parameter lists of equal length with pairwise-compatible types under compat_τ. The corresponding Rocq theorem is:

```
Theorem check_function_param_types_consistency_correct :
  forall env1 env2 fuel F1 F2,
    check_function_param_types_consistency env1 env2 fuel F1 F2 = [] <->
    function_param_sigs_compat_spec env1 env2 fuel F1 F2.
```

The proof factors through an inner helper lemma `check_param_types_correct`, which establishes that the element-by-element comparison of the two parameter lists corresponds exactly to the `Forall2 sig_type_compat` relation.

The helper lemma is proved by induction on the v1 parameter list with a case split on the v2 list. When both lists are empty, the walk emits no break and `Forall2` holds trivially. When the lists differ in length, the walk emits a `ParamCountMismatch` and `Forall2` fails by inversion. When both heads exist, the head pair is checked via `check_sig_compat_correct` and the two tails are discharged by the inductive hypothesis.

The outer theorem then follows by the same shape as R1. `flat_map_empty` reduces the nested `flat_map` to a per-pair statement, and case analysis on each name matched function pair invokes the helper lemma to convert between the per-pair body and the `Forall2` conjunct of ParamCompat.

The full proof is in `rocq/Signatures.v` in the attached artifact.

Break example. A library function gains a parameter between versions, with a default value chosen specifically to preserve the v1 behaviour at the source level. Most C++ developers assume this is binary-compatible. Every existing source-level caller continues to compile and, on paper, produces the same result. The intuition is wrong. Default arguments are applied at the consumer's call site at compile time, never stored in the callee's binary. A v1-compiled consumer that was not recompiled against v2's header therefore never sets the second-argument register, and v2 reads whatever stale value happens to be there.

```
// v1/foo.cc
extern "C" int foo(int x) { return x + 1; }
```

```
// v2/foo.cc, add a delta parameter, default = 1 so old callers see the same
    result
extern "C" int foo(int x, int delta = 1) { return x + delta; }
```

```
// app/main.cc, compiled against v1's single-argument declaration
#include <iostream>
extern "C" int foo(int);
int main() {
  int r = foo(10);
  std::cout << "foo(10) = " << r << std::endl;
  return r;
}
```

Both libraries export the same `extern "C"` symbol `foo`, so the symbol layer accepts and the consumer continues to bind:

```
$ nm --dynamic v1/libfoo.so | grep ' foo$'
00000000000010f9 T foo
$ nm --dynamic v2/libfoo.so | grep ' foo$'
00000000000010f9 T foo
```

The v1 implementation reads only `%edi` (the first integer-argument register) and adds the literal constant `1`:

```
$ objdump -d v1/libfoo.so | grep -A6 '<foo>:'
00000000000010f9 <foo>:
    10f9:       55                      push   %rbp
```

```
    10fa:       48 89 e5                mov    %rsp,%rbp
    10fd:       89 7d fc                mov    %edi,-0x4(%rbp)
    1100:       8b 45 fc                mov    -0x4(%rbp),%eax
    1103:       83 c0 01                add    $0x1,%eax
    1106:       5d                      pop    %rbp
```

The v2 implementation reads both `%edi` and `%esi` and adds the two registers together. The `= 1` default in the source is nowhere to be seen in the compiled callee (which would otherwise show an `add $0x1,...` like v1's):

```
$ objdump -d v2/libfoo.so | grep -A8 '<foo>:'
00000000000010f9 <foo>:
    10f9:       55                      push   %rbp
    10fa:       48 89 e5                mov    %rsp,%rbp
    10fd:       89 7d fc                mov    %edi,-0x4(%rbp)
    1100:       89 75 f8                mov    %esi,-0x8(%rbp)
    1103:       8b 55 fc                mov    -0x4(%rbp),%edx
    1106:       8b 45 f8                mov    -0x8(%rbp),%eax
    1109:       01 d0                   add    %edx,%eax
    110b:       5d                      pop    %rbp
```

The consumer, compiled against v1's one-argument declaration, sets only `%edi = 10` (`0xa`) and calls `foo`. `%esi` is never written:

```
$ objdump -d ./app/app | grep -A6 '<main>:'
0000000000001169 <main>:
    1169:       55                      push   %rbp
    116a:       48 89 e5                mov    %rsp,%rbp
    116d:       48 83 ec 10             sub    $0x10,%rsp
    1171:       bf 0a 00 00 00          mov    $0xa,%edi
    1176:       e8 b5 fe ff ff          call   1030 <foo@plt>
    117b:       89 45 fc                mov    %eax,-0x4(%rbp)
```

Loading v1 produces the expected result. Loading v2 returns `10 + (garbage in %esi)`:

```
$ LD_LIBRARY_PATH=v1 ./app/app; echo "exit=$?"
foo(10) = 11
exit=11
$ LD_LIBRARY_PATH=v2 ./app/app; echo "exit=$?"
foo(10) = -1112944686
exit=210
```

The value `-1112944686` is precisely `10` plus whatever 32-bit word `%esi` held at the consumer's call site, and the exit code `210` is the unsigned low byte of the same value (`0xD2`). Recompiling `main.cc` against v2's two-argument declaration would let the compiler inject the default at the call site, setting `%esi = 1` and restoring the v1 behaviour, but no such recompile is required at the binary interface, and that is exactly the trap. The checker reports

```
$ veriabi v1/libfoo.so v2/libfoo.so
  FAIL  Function signatures
        ParamCountMismatch (1):
          - foo (expected: 1 params, got: 2 params)
```

matching the `ParamCountMismatch` constructor emitted by the helper when the two parameter lists differ in length.

6.3.3. R3: Variadic-Flag Consistency

The third signature-level rule enforces that the variadic flag of a function does not change between versions. Variadic functions are invoked under a different calling convention than fixed-arity functions

on most platforms. On x86_64 SysV the caller of a variadic function must set the `AL` register to the number of vector arguments passed in registers, and the callee reads it [28]. Toggling the variadic flag silently changes this contract. A consumer compiled against a fixed-arity declaration does not set `AL` or pass variadic arguments, but a v2 implementation that became variadic expects both. Formally,

$$\text{VariadicCon}(\mathcal{F}_1, \mathcal{F}_2) := \forall\, \varphi_1, \varphi_2,\ \varphi_1 \in \mathcal{F}_1 \ \to\ \varphi_2 \in \mathcal{F}_2 \ \to\ \varphi_1.n = \varphi_2.n \ \to\ \varphi_1.v = \varphi_2.v. \tag{6.22}$$

The corresponding Rocq definition is:

```
Definition function_variadic_consistency_spec (F1 F2 : list Function) :
    Prop :=
  forall f1 f2, In f1 F1 -> In f2 F2 ->
    fname f1 = fname f2 -> is_variadic f1 = is_variadic f2.
```

Decision procedure. This rule reuses the consistency-check pattern from R2 of the symbol layer. A direct instantiation of `check_consistency_by` with $\pi_k = \mathit{fname}$ and $\pi_v = \mathit{is\_variadic}$ gives `check_function_variadic_consistency`, with a break constructor that wraps the two boolean variadic flags into a `VariadicMismatch` tagged by the function name:

```
Definition check_function_variadic_consistency
  (F1 F2 : list Function) : list FunctionBreak :=
  check_consistency_by
    fname String.eqb
    is_variadic Bool.eqb
    (fun f1 f2 => VariadicMismatch (fname f1) (is_variadic f1) (
       is_variadic f2))
    F1 F2.
```

Proof of correctness. This rule reuses the consistency-check pattern established for the symbol layer (R2–R3). Formally, the per-rule theorem states

$$\mathit{check\_function\_variadic\_consistency}(\mathcal{F}_1, \mathcal{F}_2) = \emptyset \iff \text{VariadicCon}(\mathcal{F}_1, \mathcal{F}_2), \tag{6.23}$$

that is, the checker emits no breaks if and only if every same named v1/v2 function pair agrees on the variadic flag. Its correctness then follows from `check_consistency_by_correct` with the function name as key and the variadic flag as value; the proof is completed using standard decidable-equality properties for strings and booleans:

```
Theorem check_function_variadic_consistency_correct :
  forall F1 F2,
    check_function_variadic_consistency F1 F2 = [] <->
    function_variadic_consistency_spec F1 F2.
Proof.
  intros.
  apply check_consistency_by_correct.
  intros. apply String.eqb_eq.
  intros. apply Bool.eqb_true_iff.
  intros. apply Bool.eqb_false_iff.
Qed.
```

The full proof is in `rocq/Signatures.v` in the attached artifact.

Break example. A function loses its variadic flag between versions while keeping the same name and the first fixed parameter:

```
// libv1/foo.cc
#include <cstdarg>
extern "C" int foo(int n, ...) {
```

```
  va_list ap; va_start(ap, n);
  int s = 0; for (int i = 0; i < n; i++) s += va_arg(ap, int);
  va_end(ap); return s;
}
```

```
// libv2/foo.cc
extern "C" int foo(int n) { return n; }
```

```
// app/main.cc
extern "C" int foo(int, ...);
int main() { return foo(2, 10, 20); }
```

For `extern "C"` functions, the variadic flag is not encoded in the symbol name, and the parameter-count layer (R2) sees one fixed parameter on both sides, so neither earlier rule catches the change. For mangled C++ functions the Itanium ABI does encode the variadic flag in the mangled name, so R1 would detect the removal. The consumer's call site follows the variadic calling convention. On x86_64 SysV it zeroes `%eax` (whose low byte `%al` carries the count of vector arguments passed in registers) before `call`, in addition to placing the three integer arguments in `%edi`, `%esi`, `%edx`.

```
$ objdump -d ./app/app | grep -A7 '<main>:'
0000000000001139 <main>:
    1139:       55                      push   %rbp
    113a:       48 89 e5                mov    %rsp,%rbp
    113d:       ba 14 00 00 00          mov    $0x14,%edx
    1142:       be 0a 00 00 00          mov    $0xa,%esi
    1147:       bf 02 00 00 00          mov    $0x2,%edi
    114c:       b8 00 00 00 00          mov    $0x0,%eax
    1151:       e8 da fe ff ff          call   1030 <foo@plt>
```

v2's non-variadic implementation is correspondingly minimal. It copies `%edi` into a local stack slot, loads it back into `%eax` as the return value, and returns:

```
$ objdump -d v2/libfoo.so | grep -A6 '<foo>:'
00000000000010f9 <foo>:
    10f9:       55                      push   %rbp
    10fa:       48 89 e5                mov    %rsp,%rbp
    10fd:       89 7d fc                mov    %edi,-0x4(%rbp)
    1100:       8b 45 fc                mov    -0x4(%rbp),%eax
    1103:       5d                      pop    %rbp
    1104:       c3                      ret
```

`%esi` and `%edx` are never read, so the extra arguments the consumer believed it was passing are simply discarded. The two runs disagree:

```
$ LD_LIBRARY_PATH=v1 ./app/app; echo "exit=$?"
exit=30
$ LD_LIBRARY_PATH=v2 ./app/app; echo "exit=$?"
exit=2
```

The 10 and 20 are silently dropped on v2. There is no crash, just a different result. v2 reads only `%edi` and ignores the extra integer registers entirely. With floating-point variadic arguments the same change would be worse, since variadic and non-variadic conventions place floats in different register classes and every float would land in the wrong one. The checker surfaces the change as

```
$ veriabi v1/libfoo.so v2/libfoo.so
  FAIL  Function signatures
        VariadicMismatch (1):
          - foo (expected: true, got: false)
```

matching the `VariadicMismatch` constructor produced by `check_function_variadic_consistency`.

6.3.4. R4: Object Type Compatibility

The fourth signature-level rule applies to global objects rather than functions. The consumer's load or store at the address of `g` reads or writes `sizeof(type of g)` bytes from a fixed location. If v2 redeclares `g` with a different type, the consumer reads the wrong width or writes past the v2 storage. Object symbol names are mangled, but the type is not encoded in the mangling, so this break is invisible at the symbol level. Formally,

$$\begin{aligned}\text{ObjCompat}(\mathcal{T}_1, \mathcal{T}_2, f, \mathcal{O}_1, \mathcal{O}_2) \;:=\; &\forall\, \omega_1 \in \mathcal{O}_1,\ \omega_2 \in \mathcal{O}_2,\\ &\omega_1.n = \omega_2.n \;\rightarrow\; \text{compat}_\tau(\mathcal{T}_1, \mathcal{T}_2, f, \omega_1.\tau,\ \omega_2.\tau).\end{aligned} \tag{6.24}$$

The corresponding Rocq definition is:

```
Definition object_type_compat_spec (env1 env2 : list TypeDef) (fuel : nat)
    (O1 O2 : list Object) : Prop :=
  forall o1 o2, In o1 O1 -> In o2 O2 ->
    oname o1 = oname o2 ->
    sig_type_compat env1 env2 fuel (type_sig o1) (type_sig o2).
```

Decision procedure. The structure is the same as for R1's return-type checker, with `return_sig` replaced by `type_sig`, `fname` by `oname`, and the two break constructors specialised to objects, as Algorithm 7 shows.

Algorithm 7 Object-type check (check_object_type_consistency)

Require: type environments $\mathcal{T}_1, \mathcal{T}_2$, fuel f, object lists $\mathcal{O}_1, \mathcal{O}_2$
Ensure: list of object breaks

breaks ← [ ]
for all $(o_1, o_2) \in \mathcal{O}_1 \times \mathcal{O}_2$ with $name(o_1) = name(o_2)$ **do**
  if not check_sig_compat($\mathcal{T}_1, \mathcal{T}_2, f, type(o_1), type(o_2)$) **then**
    if the printed names of $type(o_1)$ and $type(o_2)$ are equal **then**
      append TypeLayoutChanged($name(o_1)$) ▷ name kept, layout drifted
    else
      append TypeMismatch($name(o_1)$) ▷ object type renamed
    end if
  end if
end for
return *breaks*

In Rocq:

```
Definition check_object_type_consistency (env1 env2 : list TypeDef) (fuel
   : nat)
    (O1 O2 : list Object) : list ObjectBreak :=
  flat_map (fun o1 =>
    flat_map (fun o2 =>
      if String.eqb (oname o1) (oname o2) then
        if check_sig_compat env1 env2 fuel (type_sig o1) (type_sig o2)
           then []
        else let n1 := sig_type_name (type_sig o1) in
             let n2 := sig_type_name (type_sig o2) in
             if String.eqb n1 n2
             then [TypeLayoutChanged (oname o1) n1]
             else [TypeMismatch      (oname o1) n1 n2]
      else []
    ) O2
  ) O1.
```

Proof of correctness. Formally, the per-rule theorem states

$$\begin{aligned} &\mathit{check\_object\_type\_consistency}(\mathcal{T}_1, \mathcal{T}_2, f, O_1, O_2) = \emptyset \\ &\quad \Longleftrightarrow \ \mathrm{ObjCompat}(\mathcal{T}_1, \mathcal{T}_2, f, O_1, O_2), \end{aligned} \tag{6.25}$$

that is, the checker emits no breaks if and only if every same named v1/v2 object pair has compatible types under compat_τ. The corresponding Rocq theorem is:

```
Theorem check_object_type_consistency_correct :
  forall env1 env2 fuel O1 O2,
    check_object_type_consistency env1 env2 fuel O1 O2 = [] <->
    object_type_compat_spec env1 env2 fuel O1 O2.
```

The proof has the same structure as R1, with object projections (`oname`, `type_sig`) in place of function projections. The nested `flat_map` is reduced to a per-pair statement via `flat_map_empty`, and `check_sig_compat_correct` converts between boolean acceptance and propositional compat_τ for each same named pair. The full proof is in `rocq/Signatures.v` in the attached artifact.

Break example. A global variable changes type between versions, with the v2 value chosen specifically to preserve the v1 nominal value. The intuition that a "value-preserving" type upgrade, here from `int` to `double` for precision, is binary-compatible is wrong for the same reason that the default-argument intuition is wrong for R2. The consumer's compiled code commits to v1's storage layout, and v2's different layout silently misroutes the load.

```
// v1/g.cc
int g = 42;
```

```
// v2/g.cc, changed to double for precision, same nominal value
double g = 42.0;
```

```
// app/main.cc , compiled against v1's int declaration
#include <iostream>
extern int g;
int main() {
  std::cout << "g = " << g << std::endl;
  return g;
}
```

The Itanium ABI does not mangle variables declared in the global namespace (mangling only applies to namespaced or class-scoped variables, where the qualified name must be encoded). Both libraries therefore export the same plain symbol `g`:

```
$ nm --dynamic v1/libg.so | grep ' g$'
0000000000004008 D g
$ nm --dynamic v2/libg.so | grep ' g$'
0000000000004008 D g
```

The dynamic linker resolves the consumer's reference and proceeds, although `ld.so` does notice the size disparity via the ELF symbol-size field and prints a non-fatal warning when v2 is loaded. The warning is advisory only. The actual storage layouts of the two versions are visible directly in the `.data` section dumps. v1 holds the four little-endian bytes `2a 00 00 00` for the integer `42`:

```
$ objdump -s -j .data v1/libg.so
Contents of section .data:
 4000 00400000 00000000 2a000000           .@......*...
```

v2 holds the eight little-endian bytes `00 00 00 00 00 00 45 40` for the IEEE 754 representation of `42.0`:

```
$ objdump -s -j .data v2/libg.so
Contents of section .data:
 4000 00400000 00000000 00000000 00004540  .@.....E@
```

The decisive observation is the byte layout. In IEEE 754 double precision, `42.0` is encoded as `0x4045 000000000000`: the sign bit, the eleven-bit exponent, and the high bits of the mantissa live in the upper bytes (the leading `4045`), while the trailing 52 bits of the mantissa, all zero for a small integer like 42, occupy the lower bytes. On little-endian x86_64 the high-order bytes therefore appear last on disk. The consumer's compiled code, however, reads the *first* four bytes at the address of `g`, which are all zero. Running both configurations confirms the silent zero, with `ld.so`'s warning visible on the v2 run:

```
$ LD_LIBRARY_PATH=v1 ./app/app; echo "exit=$?"
g = 42
exit=42
$ LD_LIBRARY_PATH=v2 ./app/app; echo "exit=$?"
./app/app: Symbol ‘g’ has different size in shared object, consider re-linking
g = 0
exit=0
```

The v2 run prints `g = 0` because the consumer's four-byte little-endian load picks up the four leading zeros of the double's bit pattern, not because v2's stored value is in any sense "zero". The checker surfaces the disagreement as

```
$ veriabi v1/libg.so v2/libg.so
  FAIL  Object signatures
        TypeMismatch (1):
          - g (expected: int, got: double)
```

matching the `TypeMismatch` constructor emitted by `check_object_type_consistency`.

6.3.5. Combined Signature-Level Compatibility

The function-level and object-level invariants combine into

$$\begin{aligned}&\mathrm{SigSpec}(\mathcal{T}_1, \mathcal{T}_2, f, \mathcal{F}_1, \mathcal{F}_2, \mathcal{O}_1, \mathcal{O}_2) \;:=\\ &\quad \mathrm{RetCompat} \wedge \mathrm{ParamCompat} \wedge \mathrm{VariadicCon} \wedge \mathrm{ObjCompat}.\end{aligned} \tag{6.26}$$

In Rocq the conjunction is split across two definitions, one per artifact kind, since the checker pipeline processes functions and objects through separate lists with different break types:

```
Definition FunctionsSpec (env1 env2 : list TypeDef) (fuel : nat)
    (F1 F2 : list Function) : Prop :=
  function_rt_compat_spec env1 env2 fuel F1 F2 /\
  function_param_sigs_compat_spec env1 env2 fuel F1 F2 /\
  function_variadic_consistency_spec F1 F2.

Definition ObjectsSpec (env1 env2 : list TypeDef) (fuel : nat)
    (O1 O2 : list Object) : Prop :=
  object_type_compat_spec env1 env2 fuel O1 O2.
```

Decision procedure. The function combined checker concatenates the three function-level checkers, and the object combined checker delegates to the single object checker:

```
Definition check_function_signatures (env1 env2 : list TypeDef) (fuel :
    nat)
    (F1 F2 : list Function) : list FunctionBreak :=
  check_function_rt_consistency          env1 env2 fuel F1 F2 ++
  check_function_param_types_consistency env1 env2 fuel F1 F2 ++
```

```
  check_function_variadic_consistency    F1 F2.

Definition check_object_signatures (env1 env2 : list TypeDef) (fuel : nat)
    (O1 O2 : list Object) : list ObjectBreak :=
  check_object_type_consistency env1 env2 fuel O1 O2.
```

As with the symbol layer, $O(n \log n)$ FMap-backed variants (`check_function_signatures_fast`, `check_object_signatures_fast`) are proved correct against the same specifications and used by `veriabi` by default.

Proof of correctness. Formally, the two combined theorems state

$$\begin{aligned} &\mathit{check\_function\_signatures}(\mathcal{T}_1, \mathcal{T}_2, f, \mathcal{F}_1, \mathcal{F}_2) = \emptyset \\ &\iff \text{FunctionsSpec}(\mathcal{T}_1, \mathcal{T}_2, f, \mathcal{F}_1, \mathcal{F}_2), \end{aligned} \tag{6.27}$$

$$\begin{aligned} &\mathit{check\_object\_signatures}(\mathcal{T}_1, \mathcal{T}_2, f, \mathcal{O}_1, \mathcal{O}_2) = \emptyset \\ &\iff \text{ObjectsSpec}(\mathcal{T}_1, \mathcal{T}_2, f, \mathcal{O}_1, \mathcal{O}_2), \end{aligned} \tag{6.28}$$

that is, each combined checker emits no breaks if and only if every sub-spec for its artifact kind holds simultaneously. In Rocq:

```
Theorem check_function_signatures_correct :
  forall env1 env2 fuel F1 F2,
    check_function_signatures env1 env2 fuel F1 F2 = [] <->
    FunctionsSpec env1 env2 fuel F1 F2.

Theorem check_object_signatures_correct :
  forall env1 env2 fuel O1 O2,
    check_object_signatures env1 env2 fuel O1 O2 = [] <->
    ObjectsSpec env1 env2 fuel O1 O2.
```

On the function side, the proof decomposes the concatenated output via `app_eq_nil`. Each of the three empty sub-outputs is converted into its sub-spec by the corresponding per-rule theorem, and the three sub-specs reassemble into `FunctionsSpec`. The backward direction reverses the process. On the object side, `ObjectsSpec` contains only one sub-spec, so the proof is a direct application of `check_object_type_consistency_correct`. The full proofs are in `rocq/Signatures.v` in the attached artifact.

6.4. Layout-Level Compatibility

The third layer concerns the layout of named types that appear across the ABI boundary: structs, classes, unions, enums, base types, and pointer types. A type's layout determines the byte offsets of its members, the contents of its virtual table, and the size and alignment of its in-memory representation. Layout changes are invisible at the symbol and signature levels, but they are exactly the changes that consumers observe directly through pointer arithmetic, vtable indexing, and base-class upcasts [28].

Layout information is extracted from DWARF debug data [1] and represented as an inductive type with five constructors,

$$\begin{aligned} T \;::=\; &\text{TBase}(n, s, a) \mid \text{TStruct}(n, s, a, \bar{m}, \bar{b}, \bar{v}) \mid \\ &\text{TUnion}(n, s, a, \bar{m}) \mid \text{TEnum}(n, s, \bar{e}) \mid \text{TPointer}(n, p), \end{aligned} \tag{6.29}$$

with constructors

- TBase(n, s, a) — a primitive type (`int`, `double`, etc.) named $n \in \mathit{String}$, of size $s \in \mathbb{N}$ bytes and alignment $a \in \mathbb{N}$ bytes.
- TStruct$(n, s, a, \bar{m}, \bar{b}, \bar{v})$ — a struct or class named n of size s and alignment a, with members $\bar{m} \in \mathit{list}(\mathit{Member})$, base classes $\bar{b} \in \mathit{list}(\mathit{BaseClass})$, and virtual function table $\bar{v} \in \mathit{list}(\mathit{String})$ recorded as an ordered list of mangled function names occupying the vtable slots.

- TUnion$(n, s, a, \bar{m})$ — a union named n of size s and alignment a, with members $\bar{m}$ all sharing offset zero.
- TEnum$(n, s, \bar{e})$ — an enum named n of size s, with enumerators $\bar{e} \in \mathit{list}(\mathit{String} \times \mathbb{Z})$ pairing each enumerator's source name with its integer value.
- TPointer(n, p) — a named pointer type, where $p \in \mathit{String}$ is the qualified name of the pointee.

A member $m \in \mathit{Member}$ is a record

$$m \;:=\; (m.n,\; m.\tau_n,\; m.\mathit{off},\; m.\mathit{boff},\; m.\mathit{bsize}), \tag{6.30}$$

with components

- $m.n \in \mathit{String}$ — the member's source name (e.g., `mode`).
- $m.\tau_n \in \mathit{String}$ — the qualified name of the member's type, used as a lookup key into the type environment.
- $m.\mathit{off} \in \mathbb{N}$ — the byte offset of the member within its enclosing type.
- $m.\mathit{boff} \in \mathbb{N}$ — the bit offset (within byte $m.\mathit{off}$) for bitfield members; zero for ordinary members.
- $m.\mathit{bsize} \in \mathbb{N}$ — the bit size for bitfield members; zero for ordinary members.

A base class $b \in \mathit{BaseClass}$ is similarly a record

$$b \;:=\; (b.n,\; b.\mathit{off},\; b.\mathit{virt}), \tag{6.31}$$

with components $b.n \in \mathit{String}$ (the base class's qualified name), $b.\mathit{off} \in \mathbb{N}$ (its offset within the derived class), and $b.\mathit{virt} \in \{\top, \bot\}$ (whether the base is virtual).

A *type environment* $\mathcal{T} \in \mathit{list}(T)$ is a finite list of type definitions, and lookup by name is defined as

$$\mathrm{lookup}(\mathcal{T}, n) \;:=\; \begin{cases} \mathit{Some}(T) & \text{if } T \in \mathcal{T} \;\wedge\; T.n = n \\ \mathit{None} & \text{otherwise} \end{cases} \tag{6.32}$$

returning the first match. The corresponding Rocq definitions are

```
Record Member := {
  mem_name       : string;
  mem_type_name  : string;
  mem_offset     : nat;
  mem_bit_offset : nat;
  mem_bit_size   : nat;
}.

Record BaseClass := {
  base_name       : string;
  base_offset     : nat;
  base_is_virtual : bool;
}.

Inductive TypeDef :=
  | TBase    (name : string) (size align : nat)
  | TStruct  (name : string) (size align : nat)
             (members : list Member) (bases : list BaseClass)
             (vtable : list string)
  | TUnion   (name : string) (size align : nat) (members : list Member)
  | TEnum    (name : string) (size : nat) (enumerators : list (string * Z)
     )
  | TPointer (name : string) (pointee : string).

```

```
Fixpoint lookup_type (env : list TypeDef) (n : string) : option TypeDef :=
  match env with
  | []         => None
  | td :: rest => if String.eqb (typedef_name td) n then Some td
                  else lookup_type rest n
  end.
```

where the inductive constructors map directly to TBase, . . . , TPointer, the record fields correspond to the tuple components (n to `mem_name`/`base_name`, τ_n to `mem_type_name`, *off* to `mem_offset`, etc.), and the recursive `lookup_type` walks the environment list returning the first match. An $O(\log n)$ `FMap`-backed variant `lookup_type_map` (built once via `build_type_map`) is proved equivalent. The remaining chapter refers to projections by their Rocq names.

Cross-name layout compatibility. The main layout-level relation is the *environment-aware member type compatibility* $\text{compat}_m(\mathcal{T}_1, \mathcal{T}_2, f, t_1, t_2)$, defined recursively on a fuel parameter $f \in \mathbb{N}$. It expresses when two qualified type names t_1 in v1 and t_2 in v2 are layout-compatible types, looking each name up in its respective environment and recursing structurally through the resulting definitions. The fuel bounds recursion through pointer chains and nested struct members, and guarantees termination on cyclic type graphs (such as a linked-list node whose `next` field is a pointer to the same struct).

The defining cases, with $T_1 = \text{lookup}(\mathcal{T}_1, t_1)$ and $T_2 = \text{lookup}(\mathcal{T}_2, t_2)$, are:

- **Fuel exhausted ($f = 0$):** require $t_1 = t_2$. This is the conservative base case for recursion bound.
- **Both unresolved ($T_1 = T_2 = \textit{None}$):** compatible. Names that do not resolve to any layout (typically opaque types whose definition is not exported) are assumed compatible at this layer. The signature-level layer ensures the pointer-passing convention is preserved.
- **One or both base:** require $t_1 = t_2$. These are the fundamental types fixed by the platform's data model (`int`, `double`, etc.) and not affected by renaming, so the comparison reduces to name equality.
- **Both struct:** require equal size and alignment, every v1 member to have a v2 counterpart at the same byte and bit offset, slot-matched members to have compatible types (recursive call with $f - 1$) and matching bit sizes, base classes to match by name with the same offsets and virtuality, and vtables to be index-compatible.
- **Both union:** require equal size and alignment, and every v1 member to have a v2 counterpart of the same name and compatible type. Unions match by member *name* rather than by offset because all union members share offset zero, and matching by offset would collapse them.
- **Both enum:** require equal size and equal integer values for every shared enumerator name.
- **Both pointer:** recurse on the pointees with $f - 1$.
- **Mixed kinds:** incompatible.

Algorithm 8 states this decision procedure. The struct, union, and enum cases delegate to the per-kind checks of rules R3, R4, and R5, and the Rocq fixpoint that follows is its verified form.

Algorithm 8 Generic layout compatibility (compat_m)

function $\text{compat}_m(\mathcal{T}_1, \mathcal{T}_2, f, t_1, t_2)$
  if $f = 0$ **then return** $t_1 = t_2$ ▷ fuel exhausted
  end if
  $D_1 \leftarrow \textsc{lookup}(\mathcal{T}_1, t_1)$; $D_2 \leftarrow \textsc{lookup}(\mathcal{T}_2, t_2)$
  if $D_1 = \bot$ **and** $D_2 = \bot$ **then return true** ▷ both opaque
  end if
  if D_1 or D_2 is a base type **then return** $t_1 = t_2$
  end if
  if both struct **then return** $\textsc{StructCompat}(D_1, D_2, f)$
  else if both union **then return** $\textsc{UnionCompat}(D_1, D_2, f)$
  else if both enum **then return** $\textsc{EnumCompat}(D_1, D_2)$
  else if both pointer **then return** $\text{compat}_m(\mathcal{T}_1, \mathcal{T}_2, f - 1, \textsc{pointee}(D_1), \textsc{pointee}(D_2))$
  else return false ▷ mixed kinds
  end if
end function

The corresponding Rocq fixpoint is

```
Fixpoint member_type_compat (env1 env2 : list TypeDef) (fuel : nat)
    (t1 t2 : string) : Prop :=
  match fuel with
  | O    => t1 = t2
  | S f' =>
    match lookup_type env1 t1, lookup_type env2 t2 with
    | None,     None     => True
    | Some td1, Some td2 =>
        if is_base td1 || is_base td2 then t1 = t2
        else match td1, td2 with
        | TStruct _ s1 a1 ms1 bs1 vt1, TStruct _ s2 a2 ms2 bs2 vt2 =>
            s1 = s2 /\ a1 = a2 /\
            (forall m1, In m1 ms1 ->
              exists m2, In m2 ms2 /\
                mem_offset m1 = mem_offset m2 /\
                mem_bit_offset m1 = mem_bit_offset m2) /\
            (forall m1 m2, In m1 ms1 -> In m2 ms2 ->
              mem_offset m1 = mem_offset m2 ->
              mem_bit_offset m1 = mem_bit_offset m2 ->
              member_type_compat env1 env2 f' (mem_type_name m1) (
                 mem_type_name m2) /\
              mem_bit_size m1 = mem_bit_size m2) /\
            bases_compat bs1 bs2 /\ vtable_compat vt1 vt2
        | TUnion _ s1 a1 ms1, TUnion _ s2 a2 ms2 =>
            s1 = s2 /\ a1 = a2 /\
            (forall m1, In m1 ms1 ->
              exists m2, In m2 ms2 /\ mem_name m1 = mem_name m2) /\
            (forall m1 m2, In m1 ms1 -> In m2 ms2 ->
              mem_name m1 = mem_name m2 ->
              member_type_compat env1 env2 f' (mem_type_name m1) (
                 mem_type_name m2))
        | TEnum _ s1 es1, TEnum _ s2 es2 =>
            s1 = s2 /\ enumerators_compat es1 es2
        | TPointer _ pt1, TPointer _ pt2 =>
            member_type_compat env1 env2 f' pt1 pt2
        | _, _ => False
        end
```

```
    | _,         _         => False
    end
  end.
```

The executable checker `check_member_type_env` has the same shape as the propositional fixpoint, returning a list of `TypeBreak` breaks instead of a `Prop`. In each leaf case it constructs the appropriate break constructor (`MemberTypeMismatch`, `StructSizeMismatch`, etc.) on disagreement and returns the empty list on agreement. The support lemma `check_member_type_env_correct` states that the boolean checker and the propositional relation are equivalent. Formally:

$$\begin{aligned} &\mathit{check\_member\_type\_env}(\mathcal{T}_1, \mathcal{T}_2, f, s, m, t_1, t_2) = \emptyset \\ &\quad \Longleftrightarrow\ \mathrm{compat}_m(\mathcal{T}_1, \mathcal{T}_2, f, t_1, t_2), \end{aligned} \tag{6.33}$$

where $s, m \in \mathit{String}$ are the enclosing struct and member names used only to label the breaks (they do not affect the truth value). The corresponding Rocq theorem is:

```
Theorem check_member_type_env_correct :
  forall env1 env2 fuel sname mname t1 t2,
    check_member_type_env env1 env2 fuel sname mname t1 t2 = [] <->
    member_type_compat env1 env2 fuel t1 t2.
```

The proof proceeds by induction on the fuel f and case analysis on the two looked-up type definitions.

In the base case $f = 0$ the checker emits a `MemberTypeMismatch` break exactly when the names differ, which is exactly the condition the propositional definition imposes.

In the inductive case, the proof case splits on the four combinations of lookup$(\mathcal{T}_1, t_1)$ and lookup$(\mathcal{T}_2, t_2)$. If neither type is found, the result is trivially true. If exactly one is found, the result is trivially false. When both lookups succeed and both type definitions are of the same kind, the proof dispatches further on the kind. The struct case factors through the helper lemmas `check_members_correct`, `check_bases_correct`, and `check_vtable_correct`. The remaining three cases factor through dedicated helpers: `check_union_members_correct` for unions, `check_enumerators_correct` for enums, and the inductive hypothesis at f' for pointers. Mixed-kind combinations collapse to $\bot$ via the kind tag.

A subtlety in the proof is that the recursion in `member_type_compat` is on fuel rather than on the inductive structure of the types, so the natural induction principle is on f (with the type pair quantified inside the inductive hypothesis). The full proof is in `rocq/Types.v` in the attached artifact. The per-rule proofs of R3, R4, and R6 invoke this lemma whenever they need to compare two member types under the env-aware relation.

6.4.1. R1: Type-Kind Stability

The first layout-level rule enforces that a type name preserved between versions has the same *kind* of type: a struct stays a struct, an enum stays an enum, and so on. A kind change typically indicates a deeper redesign that no consumer compiled against the old kind will be compatible with, since the consumer's accesses encode the kind in their generated code. The contract holds across versions if and only if every same named v1/v2 type-definition pair agrees on the kind tag. Formally, with kind$(T) \in \{\mathit{base}, \mathit{struct}, \mathit{union}, \mathit{enum}, \mathit{pointer}\}$ the tag of the inductive constructor:

$$\mathrm{TKindCon}(\mathcal{T}_1, \mathcal{T}_2) \ :=\ \forall\, T_1, T_2,\ T_1 \in \mathcal{T}_1 \ \rightarrow\ T_2 \in \mathcal{T}_2 \ \rightarrow\ T_1.n = T_2.n \ \rightarrow\ \mathrm{kind}(T_1) = \mathrm{kind}(T_2). \tag{6.34}$$

The corresponding Rocq definition is:

```
Definition type_kind_spec (T1 T2 : list TypeDef) : Prop :=
  forall t1 t2, In t1 T1 -> In t2 T2 ->
    typedef_name t1 = typedef_name t2 ->
    typedef_kind t1 = typedef_kind t2.
```

where `typedef_kind` extracts the constructor tag as a string (`"base"`, `"struct"`, etc.).

Decision procedure. The kind check is folded into the dispatch function `check_type_pair`. When two same named type definitions have different kind tags (for example a `TStruct` and a `TEnum`), the checker emits a `TypeKindChanged` break. There is no separate list-level kind-consistency checker. The kind check fires as part of the broader `check_types` procedure (Combined section below). The break constructor is

```
| TypeKindChanged (name : string).
```

Proof of correctness. Correctness of the kind clause is proved as one branch of the combined theorem `check_types_correct` (Combined section below). The proof case splits on the constructors of each same named type pair. Matching constructors satisfy the kind equality directly. Mixed constructors are ruled out by the `TypeKindChanged` branch of `check_type_pair`. The full proof is in `rocq/Types.v` in the attached artifact.

Break example. A type that v1 declared as a polymorphic class is changed to an enum in v2, while the library continues to export a function returning a pointer to it.

```
// v1/lib.cc, Status is a polymorphic class
class Status {
public:
  virtual int code() const;
};
int Status::code() const { return 42; }
extern "C" Status* make_status() { static Status s; return &s; }
```

```
// v2/lib.cc, Status is now an enum
enum Status { Ok = 0, Err = 1 };
extern "C" Status* make_status() { static Status s = Ok; return &s; }
```

```
// app/main.cc, compiled against v1
#include <cstdio>
class Status {
public:
  virtual int code() const;    // declaration only, defined in libv1
};
extern "C" Status* make_status();
int main() {
  Status* s = make_status();
  std::printf("code = %d\n", s->code());
  return 0;
}
```

Loaded against v1 the consumer dispatches `s->code()` through the vtable and prints `42`. Loaded against v2 the same call reads the 4-byte enum value and the bytes that happen to follow it as if they were a vtable pointer. The resulting indirect call dereferences whatever address is encoded there (in this case a null pointer) and the consumer exits with `SIGSEGV`:

```
$ LD_LIBRARY_PATH=v1 ./app/app; echo "exit=$?"
code = 42
exit=0
$ LD_LIBRARY_PATH=v2 ./app/app; echo "exit=$?"
Segmentation fault
exit=139
```

The checker reports three breaks for the same source-level change:

```
$ veriabi v1/lib.so v2/lib.so
  FAIL  Symbols
        MissingSymbol (1):
```

```
          - _ZNK6Status4codeEv  ->  Status::code() const
        RttiChanged (1):
          - _ZTI6Status  ->  typeinfo for Status
  FAIL  Type layouts
        TypeKindChanged (1):
          - Status
```

All three are real breaks and are a result of the same change. The `TypeKindChanged` is the one this rule directly captures: v1's type registry holds `Status` as a `TStruct` and v2's as a `TEnum`. Replacing the polymorphic class deletes `Status::code()` from the dynamic symbol table (caught by R1 of the symbol layer as `MissingSymbol`), and removes the RTTI typeinfo `_ZTI6Status` that was emitted alongside v1's vtable (caught by the same R1 as `RttiChanged`). A single source-level edit therefore results into three breaks across two layers, each caught by its own rule.

6.4.2. R2: Base-Type Size and Alignment

The second layout-level rule enforces that two base types of the same name agree on their size and alignment. The platform's data model normally fixes the size and alignment of built-in types, but the same effect occurs whenever a typedef alias for an integer type is widened between versions (such as a `size_type` typedef that was `unsigned int` in v1 and became `unsigned long` in v2), or whenever the library exports its own opaque base types whose representation it controls. Formally:

$$\begin{aligned}\mathrm{BaseSpec}(\mathcal{T}_1, \mathcal{T}_2) \;:=\; &\forall\, T_1, T_2,\; T_1 \in \mathcal{T}_1 \;\to\; T_2 \in \mathcal{T}_2 \;\to \\ &T_1 = \mathrm{TBase}(n_1, s_1, a_1) \;\to\; T_2 = \mathrm{TBase}(n_2, s_2, a_2) \;\to\; n_1 = n_2 \;\to\; s_1 = s_2 \,\wedge\, a_1 = a_2.\end{aligned} \tag{6.35}$$

The corresponding Rocq definition is:

```
Definition base_type_spec (T1 T2 : list TypeDef) : Prop :=
  forall t1 t2, In t1 T1 -> In t2 T2 ->
    forall n1 s1 a1 n2 s2 a2,
      t1 = TBase n1 s1 a1 ->
      t2 = TBase n2 s2 a2 ->
      n1 = n2 ->
      s1 = s2 /\ a1 = a2.
```

Decision procedure. The checker `check_base_type` compares size and alignment of two same named base types directly and emits a `BaseSizeMismatch` or `BaseAlignMismatch` break on disagreement, as Algorithm 9 shows.

Algorithm 9 Base-type check (check_base_type)

Require: type name, sizes s_1, s_2, alignments a_1, a_2
Ensure: list of type breaks
1: *breaks* ← [ ]
2: **if** $s_1 \neq s_2$ **then**
3: append BaseSizeMismatch(*name*, s_1, s_2)
4: **end if**
5: **if** $a_1 \neq a_2$ **then**
6: append BaseAlignMismatch(*name*, a_1, a_2)
7: **end if**
8: **return** *breaks*

In Rocq:

```
Definition check_base_type (name : string) (s1 s2 a1 a2 : nat)
  : list TypeBreak :=
  (if Nat.eqb s1 s2 then [] else [BaseSizeMismatch  name s1 s2]) ++
  (if Nat.eqb a1 a2 then [] else [BaseAlignMismatch name a1 a2]).
```

This is the simplest checker in the layout layer. There are no recursive members, base classes, or vtables to walk, only two natural-number comparisons.

Proof of correctness. The theorem states

$$check\_base\_type(n, s_1, s_2, a_1, a_2) = \emptyset \iff s_1 = s_2 \wedge a_1 = a_2. \tag{6.36}$$

The corresponding Rocq theorem is:

```
Theorem check_base_type_correct :
  forall name s1 s2 a1 a2,
    check_base_type name s1 s2 a1 a2 = [] <-> (s1 = s2 /\ a1 = a2).
Proof.
  intros. unfold check_base_type. split.
  - intros H. apply app_eq_nil in H. destruct H as [H1 H2].
    split; destruct (Nat.eqb _ _) eqn:E;
    [apply Nat.eqb_eq | discriminate | apply Nat.eqb_eq | discriminate];
    assumption.
  - intros [Hs Ha]. subst. rewrite !Nat.eqb_refl. reflexivity.
Qed.
```

The proof decomposes the concatenation of the two sub-checkers via `app_eq_nil`. Each piece is then discharged by the decidable equality of natural numbers (`Nat.eqb_eq`).

The combined theorem `check_types_correct` (Combined section below) packages the list-level correctness of BaseSpec as one conjunct. Its proof applies `check_base_type_correct` through the per-pair dispatch `check_type_pair_correct` on every same named base-type pair in $\mathcal{T}_1 \times \mathcal{T}_2$.

Break example. The motivating example is a typedef alias whose underlying width is widened between versions. The library stores and returns values through the alias. A consumer compiled against v1 holds them in a local of the v1 alias type.

```
// v1/lib.cc
typedef unsigned int handle_t;
namespace { handle_t stored = 0; }
extern "C" void     set_handle(handle_t h) { stored = h; }
extern "C" handle_t get_handle()           { return stored; }
```

```
// v2/lib.cc,  handle_t widened to 64 bits
typedef unsigned long handle_t;
namespace { handle_t stored = 0; }
extern "C" void     set_handle(handle_t h) { stored = h; }
extern "C" handle_t get_handle()           { return stored; }
```

```
// app/main.cc,  uses v1's narrow handle_t
#include <iostream>
typedef unsigned int handle_t;
extern "C" void     set_handle(handle_t h);
extern "C" handle_t get_handle();
int main() {
  set_handle(0xHUBAHUBAu);
  std::cout << "got 0x" << std::hex << get_handle() << std::endl;
  return 0;
}
```

Both runs print `0xhubahuba` because the test value fits in 32 bits. A value above 2^{32} would be silently truncated by the consumer's 32-bit local.

The change is not caught at the layout level. In DWARF, the typedef `handle_t` appears identically in both versions. Only the underlying `DW_TAG_base_type` differs (4-byte `unsigned int` in v1, 8-byte `long unsigned int` in v2).

```
$ objdump --dwarf=info v1/lib.so
...
 <1><32>: Abbrev Number: 2 (DW_TAG_typedef)
    <33>   DW_AT_name        : (indirect string, ...): handle_t
    <3a>   DW_AT_type        : <0x3e>
 <1><3e>: Abbrev Number: 3 (DW_TAG_base_type)
    <3f>   DW_AT_byte_size   : 4
    <41>   DW_AT_name        : (indirect string, ...): unsigned int
...
$ objdump --dwarf=info v2/lib.so
...
 <1><32>: Abbrev Number: 2 (DW_TAG_typedef)
    <33>   DW_AT_name        : (indirect string, ...): handle_t
    <3a>   DW_AT_type        : <0x3e>
 <1><3e>: Abbrev Number: 3 (DW_TAG_base_type)
    <3f>   DW_AT_byte_size   : 8
    <41>   DW_AT_name        : (indirect string, ...): long unsigned int
...
```

The front-end resolves the typedef and registers the base type under its canonical name, so the type registry contains two distinctly named `TBase` entries that R2 does not pair. The change is instead caught by the signature-level rules, where `set_handle` and `get_handle` expose the underlying type directly in their signatures.

```
$ veriabi v1/lib.so v2/lib.so
  FAIL  Function signatures
        ParamTypeMismatch (1):
          - set_handle param 0 (expected: unsigned int, got: long unsigned int)
        ReturnTypeMismatch (1):
          - get_handle (expected: unsigned int, got: long unsigned int)
```

R2 is still included in the formal model for cases where this transparency no longer holds. If the same primitive type name has a different size between v1 and v2, the mismatch becomes observable at the layout level. This can happen, for example, in cross-platform comparisons between 32-bit and 64-bit targets, or when compiler-specific extensions alter the representation of a primitive type.

These situations are outside the single-platform scope of this thesis. Nevertheless, the rule remains part of the overall layout specification, and the checker would reject such mismatches if they were encountered.

6.4.3. R3: Struct Layout

The third layout-level rule is the largest single rule in the chapter and the one most directly observed by consumers, because every member access, every base-class upcast, and every virtual call on a struct or class compiles to a byte offset (for members and base subobjects) or a vtable slot index (for virtual calls) that the consumer hardcodes from the v1 declaration. A shift of a single byte in any preserved member's offset, a renamed or reordered base class, or an added virtual method in the middle of the vtable can each silently misroute every consumer access to that struct.

The contract is that two same named structs in v1 and v2 must agree on five properties. We state them informally first, then give the formal definitions.

1. **Size and alignment.** $s_1 = s_2$ and $a_1 = a_2$.
2. **Member preservation.** Every v1 member must have a v2 counterpart at the same byte offset and bit offset.
3. **Member consistency.** Members at the same offset must have compatible types (under compat_m with $f - 1$) and equal bit sizes.

4. **Base classes.** v1 and v2 base classes must be in bijection by name, with matching offsets and virtuality. No bases may be added or removed.
5. **Virtual table.** v1's vtable must be a prefix of v2's. Slot additions at the end are permitted because consumers compiled against v1 do not index past the prefix they know about.

Formally, for two structs $T_1 = \text{TStruct}(n_1, s_1, a_1, \bar{m}_1, \bar{b}_1, \bar{v}_1)$ and $T_2 = \text{TStruct}(n_2, s_2, a_2, \bar{m}_2, \bar{b}_2, \bar{v}_2)$ with $n_1 = n_2$:

$$s_1 = s_2 \qquad \text{(size)} \tag{6.37}$$

$$a_1 = a_2 \qquad \text{(alignment)} \tag{6.38}$$

$$\forall\, m_1 \in \bar{m}_1,\ \exists\, m_2 \in \bar{m}_2,\ m_1.\mathit{off} = m_2.\mathit{off} \ \wedge\ m_1.\mathit{boff} = m_2.\mathit{boff} \qquad \text{(slot preservation)} \tag{6.39}$$

$$\forall\, m_1 \in \bar{m}_1, m_2 \in \bar{m}_2,\ \text{slot}(m_1) = \text{slot}(m_2) \ \rightarrow\ \text{compat}_m(\mathcal{T}_1, \mathcal{T}_2, f-1, m_1.\tau_n, m_2.\tau_n) \ \wedge\ m_1.\mathit{bsize} = m_2.\mathit{bsize} \qquad \text{(slot consistency)} \tag{6.40}$$

$$\text{BasesCompat}(\bar{b}_1, \bar{b}_2) \qquad \text{(base classes)} \tag{6.41}$$

$$\text{VtableCompat}(\bar{v}_1, \bar{v}_2) \qquad \text{(virtual table)} \tag{6.42}$$

where $\text{slot}(m) := (m.\mathit{off}, m.\mathit{boff})$ keys members on their byte and bit offset pair, BasesCompat requires name-bijection with matching offsets and virtuality, and VtableCompat requires $\forall\, i, \mathit{fn},\ \bar{v}_1[i] = \mathit{Some}\,\mathit{fn} \ \rightarrow\ \bar{v}_2[i] = \mathit{Some}\,\mathit{fn}$. The corresponding Rocq definition is:

```
Definition struct_spec (T1 T2 : list TypeDef) : Prop :=
  forall t1 t2, In t1 T1 -> In t2 T2 ->
    forall n1 s1 a1 ms1 bs1 vt1 n2 s2 a2 ms2 bs2 vt2,
      t1 = TStruct n1 s1 a1 ms1 bs1 vt1 ->
      t2 = TStruct n2 s2 a2 ms2 bs2 vt2 ->
      n1 = n2 ->
      s1 = s2 /\ a1 = a2 /\
      members_compat T1 T2 (List.length T1) ms1 ms2 /\
      bases_compat   bs1 bs2 /\
      vtable_compat  vt1 vt2.
```

where `members_compat`, `bases_compat`, and `vtable_compat` package the member, base-class, and vtable clauses respectively. The fuel passed to `members_compat` is $|\mathcal{T}_1|$, the safe upper bound discussed in the cross-name compatibility section.

Why slot equality is keyed on $(\mathit{off}, \mathit{boff})$**.** Bitfield members can share a byte offset, so matching members by byte offset alone collapses adjacent bitfields and produces spurious mismatches when the bitfield packing changes within a byte. The compound key $(\mathit{off}, \mathit{boff})$ correctly distinguishes them, in line with the bitfield placement rule of the Itanium ABI [28].

Decision procedure. The checker `check_struct_env` evaluates the five clauses in order and accumulates breaks into a single list, with a dedicated break constructor for each failing clause (e.g. `StructSizeMismatch`, `MemberRemoved`, `VTableSlotChanged`). Algorithm 10 shows the structure, which delegates each clause to a helper.

Algorithm 10 Struct check (check_struct_env)

Require: type environments $\mathcal{T}_1, \mathcal{T}_2$, fuel f, structs S_1, S_2
Ensure: list of type breaks

```
breaks ← [ ]
breaks ← breaks ++ CheckStructSizeAlign(S1, S2)        ▷ size and alignment
breaks ← breaks ++ CheckMembers(T1, T2, f, S1, S2)     ▷ members matched by (byte, bit) offset
breaks ← breaks ++ CheckBases(S1, S2)                  ▷ base classes by name, offset, virtuality
breaks ← breaks ++ CheckVtable(S1, S2)                 ▷ vtable slots preserved
return breaks
```

In Rocq:

```
Definition check_struct_env (env1 env2 : list TypeDef) (fuel : nat)
    (name : string) (s1 s2 a1 a2 : nat)
    (ms1 ms2 : list Member)
    (bs1 bs2 : list BaseClass)
    (vt1 vt2 : list string) : list TypeBreak :=
  check_struct_size_align name s1 s2 a1 a2 ++
  check_members env1 env2 fuel name ms1 ms2 ++
  check_bases   name bs1 bs2 ++
  check_vtable  name vt1 vt2.
```

The recursive member-type comparison inside `check_members` dispatches to `check_member_type_env` with fuel $f - 1$, which in turn invokes the cross-name layout relation discussed above.

Proof of correctness. The theorem states

$$\begin{aligned} &\mathit{check\_struct\_env}(\mathcal{T}_1, \mathcal{T}_2, f, n, s_1, s_2, a_1, a_2, \bar{m}_1, \bar{m}_2, \bar{b}_1, \bar{b}_2, \bar{v}_1, \bar{v}_2) = \emptyset \\ &\quad \Longleftrightarrow\ s_1 = s_2 \wedge a_1 = a_2 \wedge \text{members\_compat}(\mathcal{T}_1, \mathcal{T}_2, f, \bar{m}_1, \bar{m}_2) \wedge \\ &\qquad \text{bases\_compat}(\bar{b}_1, \bar{b}_2) \wedge \text{vtable\_compat}(\bar{v}_1, \bar{v}_2). \end{aligned} \tag{6.43}$$

The corresponding Rocq theorem is:

```
Theorem check_struct_correct :
  forall env1 env2 fuel name s1 s2 a1 a2 ms1 ms2 bs1 bs2 vt1 vt2,
    check_struct_env env1 env2 fuel name s1 s2 a1 a2 ms1 ms2 bs1 bs2 vt1
       vt2 = [] <->
    (s1 = s2 /\ a1 = a2 /\
     members_compat env1 env2 fuel ms1 ms2 /\
     bases_compat bs1 bs2 /\
     vtable_compat vt1 vt2).
```

The proof splits the four concatenated sub-checkers via `app_eq_nil`. Each sub-checker has its own correctness lemma.

- `check_struct_size_align_correct` follows from `Nat.eqb_eq`.
- `check_members_correct` proceeds by induction on $\bar{m}_1$. A helper lemma breaks the circularity between member and type comparison by taking the type-level correctness theorem as a hypothesis at the current fuel.
- `check_bases_correct` splits into presence and consistency, keyed on `base_name`.
- `check_vtable_correct` proceeds by induction on the v1 vtable list, checking each slot position.

The four conjuncts assemble back into the right-hand side of the biconditional, and the backward direction is symmetric. The full proof is in `rocq/Types.v` in the attached artifact. The list-level correctness of StructSpec is established as one conjunct of `check_types_correct` (Combined section below), which dispatches `check_struct_correct` through `check_type_pair_correct` for every same named struct pair in $\mathcal{T}_1 \times \mathcal{T}_2$.

Break example. A new field is inserted in the middle of an exported struct. The consumer, compiled against v1's two-field layout, reads the wrong bytes at runtime.

```
// v1/lib.cc, Config has two int fields
struct Config { int mode; int extra; };
namespace { Config stored{ 42, 99 }; }
extern "C" Config* get_config() { return &stored; }
```

```
// v2/lib.cc, double log_level inserted between mode and extra
struct Config { int mode; double log_level; int extra; };
namespace { Config stored{ 42, 1234.0, 99 }; }
extern "C" Config* get_config() { return &stored; }
```

```
// app/main.cc
#include <iostream>
struct Config { int mode; int extra; };
extern "C" Config* get_config();
int main() {
  Config* c = get_config();
  std::cout << "mode=" << c->mode << " extra=" << c->extra << std::endl;
  return 0;
}
```

The consumer's load `c->extra` compiles to a 4-byte read at offset 4, which is the offset of `extra` in v1. In v2, the `double log_level` is 8-byte aligned, so the compiler inserts 4 bytes of padding after `mode`. Offset 4 now falls inside that padding:

```
$ LD_LIBRARY_PATH=v1 ./app/app
mode=42 extra=99
$ LD_LIBRARY_PATH=v2 ./app/app
mode=42 extra=0
```

The consumer prints `extra=0` because offset 4 in v2 is padding, not a member. The struct grows from 8 to 24 bytes, and v1's `extra` at offset 4 has no counterpart in v2 at the same offset. The checker reports the size change and the missing member:

```
$ veriabi v1/lib.so v2/lib.so
  FAIL  Type layouts
        StructSizeMismatch (1):
          - Config (expected: 8, got: 24)
        StructAlignMismatch (1):
          - Config (expected: 8, got: 16)
        MemberRemoved (1):
          - Config.extra (no v2 member at same byte offset)
```

6.4.4. R4: Union Layout

The fourth layout-level rule applies to unions. Unlike structs, all union members share offset zero, so member matching is keyed on the member *name* instead of the byte offset. Size and alignment must still agree because a consumer that allocates space for a v1 union must not have v2 overrun that allocation. Adding new union members in v2 is permitted. Removing or retyping an existing member is not. Formally, for two unions $T_1 = \text{TUnion}(n_1, s_1, a_1, \bar{m}_1)$ and $T_2 = \text{TUnion}(n_2, s_2, a_2, \bar{m}_2)$ with $n_1 = n_2$:

$$\begin{aligned}\text{UnionSpec}(\mathcal{T}_1, \mathcal{T}_2, T_1, T_2) \;:=\; & s_1 = s_2 \;\wedge\; a_1 = a_2 \;\wedge \\ & (\forall\, m_1 \in \bar{m}_1,\; \exists\, m_2 \in \bar{m}_2,\; m_1.n = m_2.n) \;\wedge \\ & (\forall\, m_1 \in \bar{m}_1,\; m_2 \in \bar{m}_2,\; m_1.n = m_2.n \;\longrightarrow \\ & \quad \text{compat}_m(\mathcal{T}_1, \mathcal{T}_2, |\mathcal{T}_1|, m_1.\tau_n, m_2.\tau_n)).\end{aligned} \tag{6.44}$$

The corresponding Rocq definition is:

```
Definition union_spec (T1 T2 : list TypeDef) : Prop :=
  forall t1 t2, In t1 T1 -> In t2 T2 ->
    forall n1 s1 a1 ms1 n2 s2 a2 ms2,
      t1 = TUnion n1 s1 a1 ms1 ->
      t2 = TUnion n2 s2 a2 ms2 ->
      n1 = n2 ->
```

```
      s1 = s2 /\ a1 = a2 /\
      union_members_compat T1 T2 (List.length T1) ms1 ms2.
```

where `union_members_compat` packages the presence and type-compatibility clauses keyed on `mem_name`.

Decision procedure. The checker `check_union_env` evaluates the three clauses in order and accumulates breaks, as Algorithm 11 shows.

Algorithm 11 Union check (check_union_env)

Require: type environments $\mathcal{T}_1, \mathcal{T}_2$, fuel f, unions U_1, U_2
Ensure: list of type breaks
1: *breaks* $\leftarrow$ [ ]
2: **if** *size*$(U_1) \neq$ *size*(U_2) **then**
3: append UnionSizeMismatch(. . .)
4: **end if**
5: **if** *align*$(U_1) \neq$ *align*(U_2) **then**
6: append UnionAlignMismatch(. . .)
7: **end if**
8: *breaks* $\leftarrow$ *breaks* $+\!\!+$ CheckUnionMembers$(\mathcal{T}_1, \mathcal{T}_2, f, U_1, U_2)$ ▷ members matched by name
9: **return** *breaks*

In Rocq:

```
Definition check_union_env (env1 env2 : list TypeDef) (fuel : nat)
    (name : string) (s1 s2 a1 a2 : nat)
    (ms1 ms2 : list Member) : list TypeBreak :=
  (if Nat.eqb s1 s2 then [] else [UnionSizeMismatch  name s1 s2]) ++
  (if Nat.eqb a1 a2 then [] else [UnionAlignMismatch name a1 a2]) ++
  check_union_members env1 env2 fuel name ms1 ms2.
```

The helper `check_union_members` walks $\bar{m}_1$ and for each v1 member emits a `MemberRemoved` when no v2 member of the same name exists, or a `MemberTypeMismatch` when the name matched v2 member fails the env-aware type comparison via `check_member_type_env`.

Proof of correctness. The theorem states

$$\begin{aligned} &\mathit{check\_union\_env}(\mathcal{T}_1, \mathcal{T}_2, f, n, s_1, s_2, a_1, a_2, \bar{m}_1, \bar{m}_2) = \emptyset \\ &\quad \Longleftrightarrow\ s_1 = s_2 \wedge a_1 = a_2 \wedge \text{union\_members\_compat}(\mathcal{T}_1, \mathcal{T}_2, f, \bar{m}_1, \bar{m}_2). \end{aligned} \tag{6.45}$$

The corresponding Rocq theorem is:

```
Theorem check_union_correct :
  forall env1 env2 fuel name s1 s2 a1 a2 ms1 ms2,
    check_union_env env1 env2 fuel name s1 s2 a1 a2 ms1 ms2 = [] <->
    (s1 = s2 /\ a1 = a2 /\
     union_members_compat env1 env2 fuel ms1 ms2).
```

The proof splits the three concatenated sub-checkers via `app_eq_nil`. The size and alignment conjuncts follow from `Nat.eqb_eq`. The member-list conjunct is discharged by `check_union_members_correct`, which mirrors R3's member proof but keyed on `mem_name` instead of the byte offset. The full proof is in `rocq/Types.v` in the attached artifact. The list-level UnionSpec is established as one conjunct of `check_types_correct` below.

Break example. A union member is widened from `int` to `long` across versions, doubling the union's size from 4 to 8 bytes. The library stores an integer through the wider member, and a consumer compiled against v1's 4-byte layout places the union adjacent to a canary variable on the stack.

```
// v1/lib.cc, Data union with int and float, size 4
union Data { int i; float f; };
extern "C" {
    void store_int(Data* d, int v)  { d->i = v; }
    int  read_int(Data* d)          { return d->i; }
}
```

```
// v2/lib.cc, member i widened from int to long, size 8
union Data { long i; float f; };
extern "C" {
    void store_int(Data* d, long v) { d->i = v; }
    long read_int(Data* d)          { return d->i; }
}
```

```
// app/main.cc, compiled against v1'
#include <iostream>
union Data { int i; float f; };
struct Layout { Data d; int canary; };
extern "C" {
    void store_int(Data* d, int v);
    int  read_int(Data* d);
}
int main() {
    Layout buf;
    buf.canary = 99;
    store_int(&buf.d, 42);
    std::cout << "read_int = " << read_int(&buf.d) << std::endl;
    std::cout << "canary = " << buf.canary << std::endl;
    return (buf.canary == 99) ? 0 : 1;
}
```

The consumer's `Layout` struct places a 4-byte `Data` at offset 0 and a 4-byte `canary` at offset 4, totalling 8 bytes. The size change is visible directly in the DWARF `DW_AT_byte_size` field of the union DIE:

```
$ objdump --dwarf=info v1/lib.so | grep -A3 'DW_TAG_union_type'
 <1><33>: Abbrev Number: 4 (DW_TAG_union_type)
    <34>   DW_AT_name        : Data
    <38>   DW_AT_byte_size   : 4
$ objdump --dwarf=info v2/lib.so | grep -A3 'DW_TAG_union_type'
 <1><33>: Abbrev Number: 5 (DW_TAG_union_type)
    <34>   DW_AT_name        : Data
    <38>   DW_AT_byte_size   : 8
```

When v2's `store_int` executes `d->i = v`, it writes an 8-byte `long` starting at the address the consumer passed. The consumer passed `&buf.d`, a pointer to a 4-byte allocation. The 8-byte store writes the value 42 into the low 4 bytes (the union itself) and four zero bytes into the high 4 bytes, which on this layout are exactly the canary:

```
$ LD_LIBRARY_PATH=v1 ./app/app; echo "exit=$?"
read_int = 42
canary = 99
exit=0
$ LD_LIBRARY_PATH=v2 ./app/app; echo "exit=$?"
read_int = 42
canary = 0
exit=1
```

The v2 run prints `canary = 0`. The four zero bytes that `store_int` wrote past the end of the 4-byte union overwrote the adjacent integer. The read value itself is correct because 42 fits in both 4 and 8 bytes

on a little-endian machine, so the corruption is invisible if one only inspects the union. The checker surfaces the disagreement as

```
$ veriabi v1/lib.so v2/lib.so
  FAIL  Type layouts
        UnionSizeMismatch (1):
          - Data (expected: 4, got: 8)
        UnionAlignMismatch (1):
          - Data (expected: 4, got: 8)
        MemberTypeMismatch (1):
          - Data.i (expected: int, got: long int)
```

The `UnionSizeMismatch` is the primary violation this rule captures. The consumer's compiled `sizeof(Data)` no longer matches v2's layout, and any by-value use of the union risks an out-of-bounds write. The `MemberTypeMismatch` for `Data.i` is emitted by the member-type sub-check, which rejects the `int` to `long int` change.

6.4.5. R5: Enum Layout

The fifth layout-level rule enforces that two enums of the same name agree on their underlying integer size and on the value associated with every shared enumerator. Adding new enumerators in v2 is permitted (a consumer that never references them is unaffected), but renaming or re-assigning an existing enumerator is not, because the consumer compiled against v1 has the old integer values inlined into its switch statements, default arguments, and constant expressions. Formally, for two enums $T_1 = \text{TEnum}(n_1, s_1, \bar{e}_1)$ and $T_2 = \text{TEnum}(n_2, s_2, \bar{e}_2)$ with $n_1 = n_2$ and enumerator pairs $e = (e.n, e.val) \in \mathit{String} \times \mathbb{Z}$:

$$\begin{aligned} \text{EnumSpec}(T_1, T_2) \;:=\; & s_1 = s_2 \wedge \\ & (\forall\, e_1 \in \bar{e}_1,\, e_2 \in \bar{e}_2,\, e_1.n = e_2.n \;\rightarrow\; e_1.val = e_2.val). \end{aligned} \tag{6.46}$$

The corresponding Rocq definition is:

```
Definition enum_spec (T1 T2 : list TypeDef) : Prop :=
  forall t1 t2, In t1 T1 -> In t2 T2 ->
    forall n1 s1 es1 n2 s2 es2,
      t1 = TEnum n1 s1 es1 ->
      t2 = TEnum n2 s2 es2 ->
      n1 = n2 ->
      s1 = s2 /\ enumerators_compat es1 es2.
```

where `enumerators_compat` is the per-pair compatibility that enforces equal values on every name matched enumerator.

Decision procedure. The checker `check_enum` evaluates the size clause and then dispatches to `check_enumerators` for the value clause, as Algorithm 12 shows.

Algorithm 12 Enum check (check_enum)

Require: enum name, enums E_1, E_2
Ensure: list of type breaks
1: $\mathit{breaks} \leftarrow [\,]$
2: **if** $\mathit{size}(E_1) \neq \mathit{size}(E_2)$ **then**
3: append EnumSizeMismatch(...)
4: **end if**
5: **for all** enumerators $e_1 \in E_1, e_2 \in E_2$ with $\mathit{name}(e_1) = \mathit{name}(e_2)$ **do**
6: **if** $\mathit{value}(e_1) \neq \mathit{value}(e_2)$ **then**
7: append EnumValueMismatch(...)
8: **end if**
9: **end for**
10: **return** *breaks*

In Rocq:

```
Definition check_enum (name : string) (s1 s2 : nat)
    (es1 es2 : list (string * Z)) : list TypeBreak :=
  (if Nat.eqb s1 s2 then [] else [EnumSizeMismatch name s1 s2]) ++
  check_enumerators name es1 es2.
```

The helper `check_enumerators` walks $\bar{e}_1 \times \bar{e}_2$ via nested `flat_map`, emitting an `EnumValueMismatch` for every name matched pair whose values disagree.

Proof of correctness. The theorem states

$$\mathit{check\_enum}(n, s_1, s_2, \bar{e}_1, \bar{e}_2) = \emptyset \iff s_1 = s_2 \wedge \text{enumerators\_compat}(\bar{e}_1, \bar{e}_2). \tag{6.47}$$

The corresponding Rocq theorem is:

```
Theorem check_enum_correct :
  forall name s1 s2 es1 es2,
    check_enum name s1 s2 es1 es2 = [] <->
    (s1 = s2 /\ enumerators_compat es1 es2).
```

The proof splits the concatenated sub-checkers via `app_eq_nil`. The size conjunct reduces to `Nat.eqb_eq`, and the enumerator conjunct is discharged by `check_enumerators_correct`, which proceeds by `flat_map_empty` applied to both nested traversals and a case analysis on the name comparison of each pair (e_1, e_2). The full proof is in `rocq/Types.v` in the attached artifact.

Break example. The integer values of two enumerators are swapped between versions. The library exports a function that returns a `Color` value. The consumer dispatches on it using the v1 constants.

```
// v1/lib.cc
enum Color { Red = 0, Green = 1, Blue = 2 };
extern "C" { Color get_success_color() { return Green; } }
```

```
// v2/lib.cc, Green and Blue values swapped
enum Color { Red = 0, Blue = 1, Green = 2 };
extern "C" { Color get_success_color() { return Green; } }
```

```
// app/main.cc, compiled against v1
#include <iostream>
enum Color { Red = 0, Green = 1, Blue = 2 };
extern "C" { Color get_success_color(); }
int main() {
    Color c = get_success_color();
    const char* name = (c == Green) ? "Green" : (c == Blue) ? "Blue" : "other";
    std::cout << "success color = " << name << " (" << c << ")" << std::endl;
    return (c == Green) ? 0 : 1;
}
```

Both versions export the same mangled symbol and use the same 4-byte unsigned representation, so the change is invisible at the symbol layer. The difference is only in the enumerator values recorded in DWARF:

```
$ objdump --dwarf=info v1/lib.so | grep -A2 'DW_TAG_enumerator'
    <46>   DW_AT_name        : Red
    <4a>   DW_AT_const_value : 0
    <4c>   DW_AT_name        : Green
    <50>   DW_AT_const_value : 1
    <52>   DW_AT_name        : Blue
    <56>   DW_AT_const_value : 2
```

```
$ objdump --dwarf=info v2/lib.so | grep -A2 'DW_TAG_enumerator'
    <46>   DW_AT_name        : Red
    <4a>   DW_AT_const_value : 0
    <4c>   DW_AT_name        : Blue
    <50>   DW_AT_const_value : 1
    <52>   DW_AT_name        : Green
    <56>   DW_AT_const_value : 2
```

Both `get_success_color` implementations execute `return Green`, but v1 compiled this to the integer 1 and v2 to the integer 2. The consumer's equality check `c == Green` tests against its own compiled constant 1, so when v2 returns 2 the comparison fails and the consumer falls through to the `Blue` branch:

```
$ LD_LIBRARY_PATH=v1 ./app/app; echo "exit=$?"
success color = Green (1)
exit=0
$ LD_LIBRARY_PATH=v2 ./app/app; echo "exit=$?"
success color = Blue (2)
exit=1
```

The v2 library returned what it considers `Green` (integer 2), but the consumer's compiled constant for `Green` is 1, so it misidentifies the value as `Blue`. The checker surfaces the disagreement as

```
$ veriabi v1/lib.so v2/lib.so
  FAIL  Type layouts
        EnumValueMismatch (2):
          - Color::Green (expected: 1, got: 2)
          - Color::Blue (expected: 2, got: 1)
```

matching the `EnumValueMismatch` break constructor emitted by `check_enumerators` for each name matched pair whose values disagree.

6.4.6. R6: Pointer Layout

The sixth layout-level rule enforces that two named pointer types of the same name have layout-compatible pointees, using the env-aware cross-name relation compat_m. Unlike the signature-level pointer disjunction (which accepts on name match alone, deferring layout to the type layer), the layout-level pointer rule *is* the layout layer, so it must inspect the pointee. The recursion is bounded by the size of the v1 type environment $|\mathcal{T}_1|$, the safe upper bound discussed in the cross-name compatibility section. Formally, for two pointer types $T_1 = \text{TPointer}(n_1, p_1)$ and $T_2 = \text{TPointer}(n_2, p_2)$ with $n_1 = n_2$:

$$\text{PtrSpec}(\mathcal{T}_1, \mathcal{T}_2, T_1, T_2) \;:=\; \text{compat}_m(\mathcal{T}_1, \mathcal{T}_2, |\mathcal{T}_1|, p_1, p_2). \tag{6.48}$$

The corresponding Rocq definition is:

```
Definition pointer_spec (T1 T2 : list TypeDef) : Prop :=
  forall t1 t2, In t1 T1 -> In t2 T2 ->
    forall n1 pt1 n2 pt2,
      t1 = TPointer n1 pt1 ->
      t2 = TPointer n2 pt2 ->
      n1 = n2 ->
      member_type_compat T1 T2 (List.length T1) pt1 pt2.
```

Decision procedure. The pointer pair is dispatched through `check_member_type_env`, which is the boolean form of compat_m established in the cross-name compatibility paragraph. No standalone per-pair checker is needed; the env-aware comparison handles the pointer constructor as one of its inductive cases (`TPointer/TPointer`), recursing on the pointee names with $f - 1$.

Proof of correctness. The theorem follows directly from `check_member_type_env_correct`. When both type definitions are pointers with matching outer names, the boolean checker accepts if and only if the pointee names are layout-compatible under compat_m. The list-level PtrSpec is established as one conjunct of `check_types_correct` below.

The environment-aware variant of pointer compatibility is what enables consumer-friendly renaming at the type layer: a library may rename a pointer type (such as a typedef `IntPtr` aliasing `int*`) without breaking consumers, because the comparison recurses into the pointee's layout regardless of the outer pointer's name. This complements the signature-level pointer disjunction (Section 6.3) which accepts pointer parameters on name equality alone; the layout layer is the one that actually checks the pointee.

Break example. A pointer typedef retains its name across versions but its pointee type is replaced with a differently laid out struct. The consumer, compiled against v1, reads fields through the old layout.

```
// v1/lib.cc, Handle points to OldPayload { int x; int y; }
struct OldPayload { int x; int y; };
typedef OldPayload* Handle;
namespace { OldPayload stored{10, 20}; }
extern "C" { Handle get_handle() { return &stored; } }
```

```
// v2/lib.cc, Handle now points to NewPayload { double x; }
struct NewPayload { double x; };
typedef NewPayload* Handle;
namespace { NewPayload stored{10.0}; }
extern "C" { Handle get_handle() { return &stored; } }
```

```
// app/main.cc, compiled against v1: Handle -> OldPayload
#include <iostream>
struct OldPayload { int x; int y; };
typedef OldPayload* Handle;
extern "C" { Handle get_handle(); }
int main() {
    Handle h = get_handle();
    std::cout << "x = " << h->x << std::endl;
    return (h->x == 10) ? 0 : 1;
}
```

The typedef `Handle` appears identically in both versions' DWARF as a `DW_TAG_typedef` pointing to a `DW_TAG_pointer_type`:

```
$ objdump --dwarf=info v1/lib.so | grep -A3 'DW_TAG_typedef'
 <1><5a>: Abbrev Number: 5 (DW_TAG_typedef)
    <5b>   DW_AT_name        : Handle
    <62>   DW_AT_type        : <0x66>   (-> pointer_type -> OldPayload)
$ objdump --dwarf=info v2/lib.so | grep -A3 'DW_TAG_typedef'
 <1><53>: Abbrev Number: 5 (DW_TAG_typedef)
    <54>   DW_AT_name        : Handle
    <5b>   DW_AT_type        : <0x5f>   (-> pointer_type -> NewPayload)
```

The typedef name is unchanged. Only the chain's target has shifted from `OldPayload` to `NewPayload`. The consumer's compiled code reads `h->x` as a 4-byte integer at offset 0, but v2 stores a `double` there (8 bytes, IEEE 754 representation of 10.0). The 4-byte load picks up the low half of the double bit pattern, which for `10.0` is all zeros:

```
$ LD_LIBRARY_PATH=v1 ./app/app; echo "exit=$?"
x = 10
exit=0
$ LD_LIBRARY_PATH=v2 ./app/app; echo "exit=$?"
x = 0
exit=1
```

The checker pairs the two `TPointer("Handle", ...)` entries by their shared typedef name and recursively compares the pointees via `check_member_type_env`. Since `OldPayload` and `NewPayload` have different member layouts, the recursive descent reports the structural differences.

```
$ veriabi v1/lib.so v2/lib.so
  FAIL  Type layouts
        MemberRemoved (1):
          - OldPayload.y (no v2 member at same byte offset)
        MemberTypeMismatch (1):
          - OldPayload.x (expected: int, got: double)
```

The `MemberTypeMismatch` on `OldPayload.x` is emitted by the env-aware comparison when it descends from the pointer pair into the pointee pair: v1's `OldPayload.x` is `int` and v2's counterpart at the same offset is `double`. The `MemberRemoved` reports that `OldPayload.y` (offset 4) has no counterpart at the same offset in v2's single-member `NewPayload`. Without the pointer rule, these breaks would go undetected whenever the pointee structs carry different names across versions, since R3 only pairs structs by matching names.

6.4.7. Combined Layout-Level Specification

The six layout-level invariants combine into a single conjunction over the two type environments:

$$\begin{aligned}\text{TypesSpec}(\mathcal{T}_1, \mathcal{T}_2) \;:=\; &\text{BaseSpec}(\mathcal{T}_1, \mathcal{T}_2) \wedge \text{StructSpec}(\mathcal{T}_1, \mathcal{T}_2) \wedge \\ &\text{UnionSpec}(\mathcal{T}_1, \mathcal{T}_2) \wedge \text{EnumSpec}(\mathcal{T}_1, \mathcal{T}_2) \wedge \\ &\text{PtrSpec}(\mathcal{T}_1, \mathcal{T}_2) \wedge \text{TKindCon}(\mathcal{T}_1, \mathcal{T}_2).\end{aligned} \tag{6.49}$$

The corresponding Rocq definition is:

```
Definition TypesSpec (T1 T2 : list TypeDef) : Prop :=
  base_type_spec  T1 T2 /\
  struct_spec     T1 T2 /\
  union_spec      T1 T2 /\
  enum_spec       T1 T2 /\
  pointer_spec    T1 T2 /\
  type_kind_spec  T1 T2.
```

Decision procedure. Unlike the symbol and signature layers, where the combined checker is the concatenation of per-rule list-level checkers, the type layer routes every same named type pair through a single per-pair dispatch `check_type_pair`. This dispatch case splits on the constructor of each definition and invokes the appropriate per-pair sub-checker (`check_base_type`, `check_struct_env`, `check_union_env`, `check_enum`, or `check_member_type_env` for the pointer case), or emits a `TypeKindChanged` break when the two definitions are of different kinds. The list-level checker iterates over $\mathcal{T}_1 \times \mathcal{T}_2$ and dispatches every name matched pair:

```
Definition check_types (T1 T2 : list TypeDef) : list TypeBreak :=
  flat_map (fun t1 =>
    flat_map (fun t2 =>
      if String.eqb (typedef_name t1) (typedef_name t2)
      then check_type_pair T1 T2 (List.length T1) t1 t2
      else []
    ) T2
  ) T1.
```

As before, an $O(n \log n)$ FMap-backed variant `check_types_fast` is proved correct against the same specification and used by `veriabi` by default.

Proof of correctness. Formally, the combined theorem is

$$\mathit{check\_types}(\mathcal{T}_1, \mathcal{T}_2) = \emptyset \iff \text{TypesSpec}(\mathcal{T}_1, \mathcal{T}_2), \tag{6.50}$$

that is, the combined checker emits no breaks if and only if all six sub-specs (base, struct, union, enum, pointer, kind) hold simultaneously. The corresponding Rocq theorem is:

```
Theorem check_types_correct :
  forall T1 T2,
  check_types T1 T2 = [] <-> TypesSpec T1 T2.
```

The proof reduces the nested `flat_map` to a per-pair statement via `flat_map_empty` and then decomposes the conjunction.

In the forward direction, an empty checker output forces every same named pair to satisfy the per-pair dispatch `check_type_pair`, which by `check_type_pair_correct` lifts to the propositional relation `typedef_compat`. The proof then extracts each conjunct of TypesSpec by specializing `typedef_compat` to a same named pair and case analysing on the two constructors: matching constructors yield the corresponding spec clause (base, struct, union, enum, pointer), while mixed constructors are ruled out by the kind-tag mismatch branch and contribute the kind conjunct directly.

In the backward direction, TypesSpec provides each of the six sub-specs. For every same named pair, case analysis on the two constructors selects the corresponding sub-spec, and the per-pair correctness lemma (`check_base_type_correct`, `check_struct_correct`, `check_union_correct`, `check_enum_correct`, or `check_member_type_env_correct`) converts the sub-spec back into an empty per-pair output. The kind-tag clause discharges the mixed-constructor cases that would otherwise force a `TypeKindChanged` break.

The full proof is in `rocq/Types.v` in the attached artifact.

6.5. Combined ABI Specification

The full ABI compatibility predicate assembles the three layers into a single conjunction over the four components of an ABI bundle. An *ABI bundle* $A := (\mathcal{S}, \mathcal{F}, \mathcal{O}, \mathcal{T})$ packages a library's exported symbols, function signatures, object signatures, and type environment. Formally,

$$\begin{aligned} &\mathrm{ABISpec}(A_1, A_2) \;:= \\ &\quad \mathrm{SymbolsSpec}(\mathcal{S}_1, \mathcal{S}_2) \wedge \\ &\quad \mathrm{FunctionsSpec}(\mathcal{T}_1, \mathcal{T}_2, |\mathcal{T}_1|, \mathcal{F}_1, \mathcal{F}_2) \wedge \\ &\quad \mathrm{ObjectsSpec}(\mathcal{T}_1, \mathcal{T}_2, |\mathcal{T}_1|, \mathcal{O}_1, \mathcal{O}_2) \wedge \\ &\quad \mathrm{TypesSpec}(\mathcal{T}_1, \mathcal{T}_2), \end{aligned} \tag{6.51}$$

where the fuel argument supplied to the signature-level specifications is $|\mathcal{T}_1|$, the upper bound established in the cross-name compatibility paragraph. The Rocq definition conjoins the four layer-level specs:

```
Definition ABISpec (a1 a2 : ABI) : Prop :=
  let fuel := List.length (abi_types a1) in
  SymbolsSpec   (abi_symbols   a1) (abi_symbols   a2) /\
  FunctionsSpec (abi_types a1) (abi_types a2) fuel
                (abi_functions a1) (abi_functions a2) /\
  ObjectsSpec   (abi_types a1) (abi_types a2) fuel
                (abi_objects   a1) (abi_objects   a2) /\
  TypesSpec     (abi_types     a1) (abi_types     a2).
```

Decision procedure. The combined checker concatenates the four layer-level checkers, with each layer's break type tagged into the ABI-level sum `ABIBreak`. Algorithm 13 shows the top-level structure, and the Rocq definition follows.

Algorithm 13 Strict ABI check (check_abi)

Require: ABIs A_1, A_2 of the two library versions
Ensure: list of ABI breaks
1: *breaks* $\leftarrow$ [ ]
2: *breaks* $\leftarrow$ *breaks* $+\!\!+$ CheckSymbols(A_1, A_2)
3: *breaks* $\leftarrow$ *breaks* $+\!\!+$ CheckSignatures(A_1, A_2)
4: *breaks* $\leftarrow$ *breaks* $+\!\!+$ CheckLayouts(A_1, A_2)
5: **return** *breaks*

```
Inductive ABIBreak :=
| SymBreak  (b : SymbolBreak)
| FuncBreak (b : FunctionBreak)
| ObjBreak  (b : ObjectBreak)
| TyBreak   (b : TypeBreak).

Definition check_abi (a1 a2 : ABI) : list ABIBreak :=
  let fuel := List.length (abi_types a1) in
  map SymBreak  (check_symbols (abi_symbols a1) (abi_symbols a2)) ++
  map FuncBreak (check_function_signatures (abi_types a1) (abi_types a2)
                                           fuel
                                           (abi_functions a1) (
                                              abi_functions a2)) ++
  map ObjBreak  (check_object_signatures   (abi_types a1) (abi_types a2)
                                           fuel
                                           (abi_objects a1) (abi_objects
                                              a2)) ++
  map TyBreak   (check_types (abi_types a1) (abi_types a2)).
```

As with each layer, an $O(n \log n)$ variant `check_abi_fast` invokes the FMap-backed versions of all four sub-checkers and is proved correct against the same specification. `veriabi` uses it by default.

Proof of correctness. Formally, the combined theorem is

$$\mathit{check\_abi}(A_1, A_2) = \emptyset \iff \mathrm{ABISpec}(A_1, A_2), \tag{6.52}$$

that is, the combined checker emits no breaks if and only if all four layer-level specs (symbols, functions, objects, types) hold simultaneously. The corresponding Rocq theorem is:

```
Theorem check_abi_correct :
  forall a1 a2,
    check_abi a1 a2 = [] <-> ABISpec a1 a2.
```

The proof splits the concatenated output into four pieces via `app_eq_nil`, then strips the `map` wrappers via the auxiliary lemma `map_eq_nil_iff` (which states that `map f xs = []` iff `xs = []`). Each empty per-layer output is then converted into its layer-level spec by the corresponding theorem (`check_symbols_correct`, `check_function_signatures_correct`, `check_object_signatures_correct`, `check_types_correct`). The four sub-specs reassemble into `ABISpec`. The backward direction reverses the process.

The fast variant `check_abi_fast` satisfies the same biconditional under the standing assumption that names are unique on the v2 side of each list ($\mathcal{S}_2, \mathcal{F}_2, \mathcal{O}_2, \mathcal{T}_2$). Its proof has the same shape, dispatching to the fast variant of each layer-level correctness theorem.

The full proof is in `rocq/Types.v` in the attached artifact.

7

Tool Architecture & Implementation

The previous chapter defined the formal ABI model and the Rocq decision procedures that decide it. To make the model useful on real binaries, the verified Rocq code is extracted to OCaml and paired with a C++ frontend that turns a pair of compiled shared objects into the structured inputs the checker expects. This chapter gives a high-level view of the resulting pipeline: the system diagram and trust boundary, the role of each frontend component, and the OCaml bridge that connects the two.

7.1. System Overview

The tool, named `veriabi` [4], is a command-line program that takes two ELF shared objects as inputs and produces a structured compatibility report. A third optional argument, an LLVM IR file, enables the consumer-aware refinement of Chapter 8. The end-to-end pipeline is shown in Figure 7.1.

The two compiled shared objects are read by two independent readers, each accessing the binary file directly. The ELF reader extracts each binary's dynamic symbol table. The DWARF reader walks the same binaries' debug information to recover function signatures and object signatures, populating a type registry at the same time. The DWARF reader additionally uses the ELF reader's symbol list as a filter, restricting its work to the DIEs that correspond to exported symbols. This dependency is shown in Figure 7.1 as a dashed arrow between the two readers, distinct from the solid arrows. The resulting four data structures, two lists of symbols, two lists of signatures, and two type registries, cross the language boundary through the OCaml bridge, which marshals them into the values expected by the verified checker. The checker decides each compatibility rule and returns a list of breaks per rule, which is then returned as a human-readable report. The colour coding in the figure reflects the trust boundary: the blue stages are the unverified C++ frontend, the orange stage is the OCaml marshalling layer, and the green stage is the verified core extracted from Rocq.

The verified core is the OCaml code produced by Rocq extraction. It exports the per-rule and combined checker functions, and every correctness theorem in Chapter 6 is about one of those exported functions.

The frontend supports command-line flags for partial runs (`--check`), namespace filtering (`--ignore-ns`, `--include-ns`, `--include-ref`), weak-symbol filtering (`--ignore-weak`), the recursion fuel bound (`--fuel`, defaulting to 10), and the conventional `--help` and `--version` switches. The exit status follows a three-way convention: 0 on `COMPATIBLE`, 1 on `INCOMPATIBLE`, and 2 on an execution error such as a malformed flag, a missing input file, or an uncaught exception from one of the readers. A CI system can therefore distinguish "verdict known and incompatible" from "tool failed to produce a verdict at all".

Trust boundary. As mentioned before, the colour coding in Figure 7.1 reflects the trust boundary. The blue and grey stages are C++ code. They are unit-tested but unverified, and bugs in the readers can produce incorrect inputs to the checker. The green stage, the verified checker, returns a verdict that is decidable equality between its output and the empty list, formally tied to the corresponding specification. The frontend's responsibility is therefore precisely the construction of the four data structures, not the comparison logic.

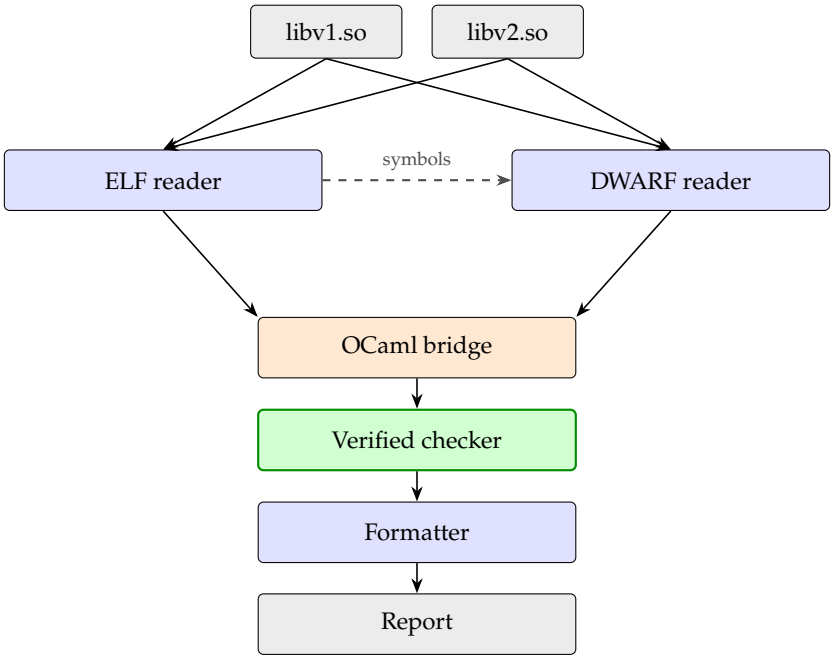


Figure 7.1: End-to-end pipeline of `veriabi`.

7.2. ELF Symbol Extraction

The ELF reader produces one `Symbol` record per dynamic-symbol-table entry, populated with the four fields the formal model requires: the linkage (mangled) name, the symbol kind (function, object, or no-type), the binding strength (strong or weak), and a definedness flag. The dynamic symbol table is a fixed-format ELF section read directly through the standard `libelf` interface. RTTI typeinfo symbols are detected by the `_ZTI` prefix of the mangled name and re-classified as a separate `TypeInfo` kind so the checker can emit `RttiChanged` breaks rather than the generic `MissingSymbol`.

Two filters reduce the raw symbol table to the exported ABI surface. The visibility filter drops hidden and internal symbols, since those are intentionally outside the exported interface. A second filter skips standard-library and compiler-internal mangling prefixes, `_ZSt` (anything in `std::`), `_ZN9__gnu_cxx` (libstdc++ internals), and the vtable, VTT, and thunk prefixes. These symbols are present in every C++ shared object as artifacts of the C++ runtime, not as part of the library author's intended ABI, and including them in the report would drown real breaks in noise.

A third filter is user-controlled. The `--ignore-ns`, `--include-ns`, and `--include-ref` flags accept comma-separated namespace prefixes that scope the report to, or away from, a specific subsystem. Excluding third-party namespaces such as `boost::` or `Qt::` therefore does not require disabling the rules themselves.

7.3. DWARF Debug Information Parsing

The DWARF reader produces the function and object signatures and populates the type registry. ELF symbols alone do not carry enough information to reconstruct them. The rest lives in DWARF 5 debug information emitted by the compiler with debug flags enabled. The reader uses `libdw` (`elfutils`) to walk the debug-information tree, which is organized as debug information entries (DIEs) grouped by compilation unit.

The walk is in two passes. A pre-index pass populates two name-keyed maps, one from fully qualified type names to defining-DIE offsets, and one from struct or class names to definition offsets (as opposed to forward-declaration offsets). The second map is necessary because the compiler may emit a forward declaration in one compilation unit and the definition in another. Without the pre-index, a base-class reference inside the first CU would resolve to the declaration and produce a placeholder `void`-typed layout. The extraction pass then matches each exported symbol against its DIE, following

`DW_AT_specification` and `DW_AT_abstract_origin` chains where necessary, and reads the return type, parameter list, or object type. Anonymous nested structs are flattened into the enclosing type at the appropriate offset, since the formal model expects a flat member list.

7.4. Type System and Registry

The type registry is the C++-side counterpart of the type environment $\mathcal{T}$ in the formal model. Where the model uses a finite list of `TypeDef` records keyed by name, the registry stores a hash map from a `TypeID` (the hash of the type's canonical form) to an instance of a `Type` subclass. The hierarchy is on purpose close to a one-to-one mirror of the Rocq inductive, as shown in Figure 7.2: each `Type::Kind` value corresponds to one constructor, with two exceptions. `ReferenceType` maps to the same `TPointer` constructor as `PointerType`, because lvalue and rvalue references are passed and returned in the same registers and stack slots as pointers at the binary level. `QualifiedType`, `ArrayType`, and `FunctionType` are present in the C++ hierarchy for completeness but are skipped during marshalling, since they are not part of the model's structural comparison.

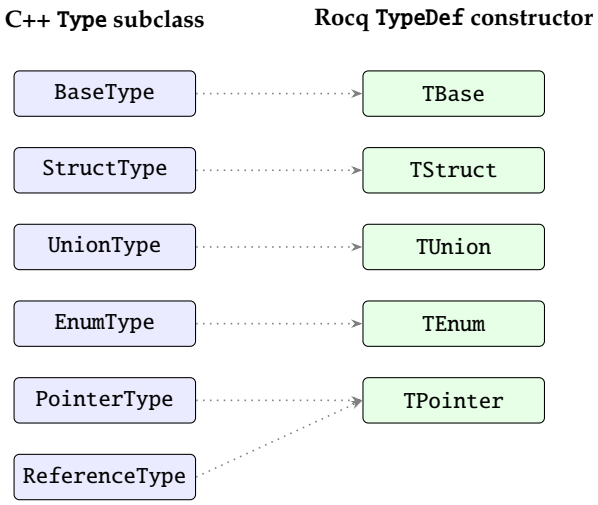


Figure 7.2: Correspondence between C++ `Type` subclasses (left) and the Rocq `TypeDef` constructors (right).

Type extraction is performed on demand. The registry is populated while function and object signatures are read. Each parameter, return type, and object type that names an aggregate causes the registry to allocate or look up the corresponding entry. This means the registry contains exactly the types reachable from exported signatures, which is fewer than the number of types declared in the binary's debug information. The set of *root* type IDs (those directly named by signatures, before pointer indirections are followed) is passed to the checker alongside the registry so that the type-layout rules of Chapter 6 can be restricted to ABI-visible types.

7.5. Signature Type Representation

Signatures cross the language boundary in a collapsed form. Each parameter and return type is reduced to a pair $(\mathit{depth}, \mathit{base})$. The depth counts pointer indirections and the base names the innermost named type. The signature-level rules need nothing more from a parameter, since deeper structure (struct members, vtable entries) lives in the type registry and is consulted only at the layout layer. References (`T&` and `T&&`) are folded into pointer depth on the C++ side, following the formal model. A reference parameter occupies a pointer-sized register or stack slot, so a distinct representation in the signature would create breaks no consumer can observe.

7.6. OCaml Bridge

The OCaml bridge is the layer that connects the C++ frontend to the Rocq-extracted checker. C++ structs are flattened into nested arrays of tuples and pushed across the boundary by `caml_callbackN`. The OCaml bridge unflattens them into the records and lists expected by the extracted checker, runs the

check, and returns a result tuple that the C++ side unpacks back into a `CheckResult`.

The Rocq side is described by an `Extraction` directive that names the checker functions to expose, including the per-rule entry points, the bundled `check_abi` and `check_abi_fast`, and the consumer-aware variant `check_abi_consumer`. The directive also customises a small number of extractions for performance. Rocq `string` extracts to OCaml `string` directly, avoiding the cons-cell representation that the default extraction would produce, which is very slow for the string-heavy workloads at the type layer. Rocq `nat` extracts to OCaml `int`.

The boundary is the only place in the system where an unchecked cast happens. Every input the bridge accepts must have the right type at the right index. A wrong type silently corrupts the input to the verified checker. This is a real cost of the design and is acknowledged. The soundness of the tool extends as far as "the bridge correctly marshalled the C++ data into the values the verified checker received", and no further. The bridge is small (around 500 lines) so it can be reviewed by hand against the extracted module signatures.

8

Consumer-Aware Analysis

The formal model of Chapter 6 is intentionally strict. It flags every difference that could break any consumer, regardless of whether a specific consumer actually uses the affected interface. This is the right default when the set of consumers is unknown. When the consumer is known, however, many reported breaks may be irrelevant because the program never touches the affected symbols, vtable slots, or member layouts. This chapter describes how the tool narrows its verdict for that case. Given the consumer's compiled LLVM IR as an additional input, the checker re-runs the same compatibility check and filters the results through a set of relevance predicates defined in Rocq. The filtering is proved sound with respect to the strict verdict, meaning it can only ever remove breaks, never introduce them.

The chapter follows the same structure as Chapter 6. It defines consumer usage formally, presents the relevance predicates and their decision procedures, and proves that filtering is sound with respect to the strict verdict. The LLVM-based extractor that builds the consumer-usage record is unverified, like the ELF and DWARF readers of Chapter 7.

8.1. Motivation

A few concrete examples make the gap between the strict and the consumer-aware compatibility checks visible.

Removed symbol. A library function `compute_normals` is removed in v2. The strict check reports `MissingSymbol`. If the consumer never calls `compute_normals`, nothing breaks at load time.

Struct size change. A library struct `Mesh` grows from 32 to 40 bytes in v2 because a member was added. The strict check reports `StructSizeMismatch`. If the consumer only holds pointers to `Mesh` that the library allocated, the layout change is invisible. The break only matters when the consumer itself allocates a `Mesh` on the stack, by value, or via `new`, because then the compiled code commits to a fixed size.

Vtable slot reorder. A polymorphic class `Renderer` has its second and third virtual methods swapped between versions. The strict check reports `VTableSlotChanged` for both slots. If the consumer only calls the first virtual method, the swap is invisible.

Member layout change. A library struct gains a bitfield, shifting subsequent members. The strict check reports `MemberBitOffsetMismatch` for every shifted member. If the consumer accesses none of those members directly, the shift is invisible.

A strict break is irrelevant whenever the consumer's compiled code contains no instruction that depends on the changed quantity. This chapter formalises that pattern as a set of predicates over a structured *consumer-usage record* and uses them to filter the strict output.

8.2. Consumer Usage from LLVM IR

The consumer is required to be available as a compiled LLVM bitcode or textual IR file. We chose LLVM IR rather than the consumer's ELF binary for two reasons. First, IR carries debug information annotations that resolve names to the C++ classes the consumer actually used, whereas machine code retains only addresses. Second, IR exposes typed memory operations (`getelementptr`, `alloca`, vtable loads) at a level where the relevant patterns can be matched syntactically, rather than reconstructed from optimised x86. This requires the consumer to be compiled with `clang` and to emit IR. The technique does not apply to consumers that exist only as stripped binaries.

The `IRReader` component, implemented against the LLVM C++ API, walks the IR module once and extracts four categories of usage.

Used external symbols. Every function or global declared but not defined in the module is recorded as a mangled-name string. These are the names against which the strict checker's symbol and signature breaks are filtered.

Vtable slot accesses. The IR pattern for a virtual call is recognisable. `clang` attaches type-based alias analysis (TBAA) metadata to each typed memory access, recording the type of location it touches, and a load of an object's vtable pointer carries a distinctive `vtable pointer` tag. The pattern consists of a `load` of the receiver's vptr carrying that tag, a `getelementptr` that indexes a slot, a second `load` of the function pointer, and an indirect `call`. The reader matches this pattern, recovers the class name from the receiver's static type through the DWARF-derived `DIType` chain, and records a `(class_name, slot_index)` pair for each call site.

Member accesses. A `getelementptr` whose source type is a struct or class encodes a member read or write, with the slot index identifying the member. The reader recovers the class name from the source type and records a `(class_name, byte_offset)` pair. A zero-offset member access without a `getelementptr` is distinguished from a vtable-pointer load by inspecting the TBAA tag.

Stack allocations. An `alloca` whose attached `DILocalVariable` metadata names a class or struct type indicates that the consumer creates an instance of that type on the stack or embeds it by value in another type. The reader resolves the class name through the local-variable debug record and records the type name.

The four lists together form the consumer-usage record.

8.3. The Consumer-Usage Record

On the verified side, the four lists are packaged into a single Rocq record:

```
Record ConsumerUsage := mkConsumerUsage {
  cu_used_symbols    : list string;
  cu_vtable_accesses : list (string * nat);
  cu_member_accesses : list (string * nat);
  cu_stack_allocs    : list string;
}.
```

The four fields correspond exactly to the four categories extracted from the IR. The C++ frontend builds this record at the marshalling boundary. The bridge converts each list element from C++ to the corresponding Rocq type and constructs the record.

A small set of primitive boolean predicates queries the record. Each is a one-line existence check:

```
Definition cu_uses_symbol cu name : bool :=
  existsb (String.eqb name) (cu_used_symbols cu).

Definition cu_uses_vtable_slot cu cls slot : bool :=
  existsb (fun p => andb (String.eqb cls (fst p))
```

```
                          (Nat.eqb slot (snd p)))
          (cu_vtable_accesses cu).

Definition cu_accesses_class_member cu cls : bool :=
  existsb (fun p => String.eqb cls (fst p)) (cu_member_accesses cu).

Definition cu_allocates_type cu name : bool :=
  existsb (String.eqb name) (cu_stack_allocs cu).
```

`cu_accesses_class_member` is intentionally coarse. It returns true if the consumer accesses *any* byte offset in the class, not specifically the offset that changed. Fine-grained per-offset filtering would require the verified core to consult the v1 type environment to translate the IR-recovered byte offset into the formal-model member identity. A single class may have several members at adjacent offsets, and the IR's offset is determined by the v1 layout that the consumer was compiled against. Coarse filtering is sound because it keeps strictly more breaks than the precise version would. Refining it is left as future work.

The other three predicates are exact. A symbol-name match is unambiguous because mangled names are unique. A stack allocation either names a class or it does not. A vtable slot is identified by the `(class, slot)` pair, which is the same pair recovered from the IR and the same pair the formal model uses.

8.4. Relevance Predicates

Each break category from Chapter 6 has a corresponding relevance predicate that decides, given a consumer-usage record, whether the break is observable.

Symbol-level breaks. Every symbol-level break carries the affected symbol name, so the relevance test reduces to checking whether the consumer references that name:

```
Definition symbol_break_relevant (cu : ConsumerUsage)
                                 (b : SymbolBreak) : bool :=
  match b with
  | MissingSymbol n | DuplicateSymbol n
  | KindMismatch n _ _ | DefinitionMismatch n _ _
  | StrengthMismatch n _ _ | RttiChanged n =>
      cu_uses_symbol cu n
  end.
```

Signature-level breaks. Function-signature and object-signature breaks reduce to the same name check. A change to a function's return type or a parameter type only matters if the consumer calls the function. A change to a global object's type only matters if the consumer references the object. The relevance predicates `func_break_relevant` and `obj_break_relevant` both consult `cu_uses_symbol` on the affected symbol's mangled name.

Type-layout breaks. The type layer is the only place where the relevance test is not a uniform name check. Each break category maps to the consumer-usage predicate that captures the observable channel through which it could affect the consumer:

```
Definition type_break_relevant (cu : ConsumerUsage)
                               (b : TypeBreak) : bool :=
  match b with
  | VTableSlotChanged cls slot _ _    => cu_uses_vtable_slot cu cls slot
  | VTableSlotRemoved cls slot _      => cu_uses_vtable_slot cu cls slot
  | StructSizeMismatch cls _ _        => cu_allocates_type cu cls
  | StructAlignMismatch cls _ _       => cu_allocates_type cu cls
  | MemberRemoved cls _               => cu_accesses_class_member cu cls
```

```
  | MemberTypeMismatch cls _ _ _      => cu_accesses_class_member cu cls
  | MemberBitOffsetMismatch cls _ _ _ => cu_accesses_class_member cu cls
  | MemberBitSizeMismatch cls _ _ _   => cu_accesses_class_member cu cls
  | _                                 => true
  end.
```

The default branch keeps any break for which the current usage record gives no evidence of irrelevance to the consumer. This is the safe default. `TypeKindChanged`, `BaseClassRemoved`, `BaseOffsetMismatch`, and similar structural breaks are kept regardless of the consumer's usage profile, because the consumer-usage record carries no information that would let us treat them as irrelevant.

The mapping between break category and predicate is the operational definition of "observable channel". A vtable slot change is observable if and only if the consumer calls that slot. A struct size change is observable if and only if the consumer allocates the type by value. The match expression makes the choice explicit for each break, which simplifies adding new break categories later. The only obligation is to extend the match with the appropriate predicate or default to `true`.

8.5. Filtering and Correctness

The filter functions are list filters keyed on the relevance predicates:

```
Definition filter_symbol_breaks (cu : ConsumerUsage) (bs : list
    SymbolBreak)
  := filter (symbol_break_relevant cu) bs.

Definition filter_func_breaks   (cu : ConsumerUsage) (bs : list
    FunctionBreak)
  := filter (func_break_relevant cu) bs.

Definition filter_obj_breaks    (cu : ConsumerUsage) (bs : list
    ObjectBreak)
  := filter (obj_break_relevant cu) bs.

Definition filter_type_breaks   (cu : ConsumerUsage) (bs : list TypeBreak)
  := filter (type_break_relevant cu) bs.
```

Each filter comes with a one-line correctness theorem that reduces to the standard-library `filter_In` lemma:

```
Theorem filter_symbol_breaks_correct :
  forall cu bs b,
    In b (filter_symbol_breaks cu bs)
    <-> (In b bs /\ symbol_break_relevant cu b = true).
Proof. intros. apply filter_In. Qed.
```

The three other filter theorems have identical shape and proof. Each one states that the filtered list contains exactly the breaks of the original list that are flagged as relevant to the consumer. Nothing is added, and nothing is silently dropped beyond what the relevance predicate marks as irrelevant.

End-to-end checker. The four filters are combined into a single ABI-level checker. Each strict break wraps into a tagged sum `ABIBreak`, and `abi_break_relevant` dispatches to the corresponding per-tier predicate. Algorithm 14 shows the consumer-aware checker at a high level, with the Rocq definitions below.

Algorithm 14 Consumer-aware ABI check (check_abi_consumer)

Require: ABIs A_1, A_2, consumer-usage record cu
Ensure: list of ABI breaks the consumer can observe
1: $breaks \leftarrow \textsc{CheckABI}(A_1, A_2)$ ▷ strict check first
2: **return** $[\, b \in breaks : \textsc{Relevant}(cu, b) \,]$ ▷ keep only relevant breaks

```
Definition abi_break_relevant (cu : ConsumerUsage) (b : ABIBreak) : bool
    :=
  match b with
  | SymBreak  b => symbol_break_relevant cu b
  | FuncBreak b => func_break_relevant   cu b
  | ObjBreak  b => obj_break_relevant    cu b
  | TyBreak   b => type_break_relevant   cu b
  end.

Definition check_abi_consumer (a1 a2 : ABI) (cu : ConsumerUsage)
  : list ABIBreak :=
  filter (abi_break_relevant cu) (check_abi a1 a2).
```

The consumer-aware checker first runs the strict `check_abi` from Chapter 6 and then filters its output. This sequencing is deliberate. The strict check is the trust anchor, and the filter only removes breaks. No verdict can appear in the consumer-aware report that does not appear in the strict report.

Soundness with respect to strict compatibility. The corollary that ties the two checkers together states that strict compatibility implies the consumer-aware checker reports nothing, regardless of the consumer:

```
Corollary abi_compat_implies_consumer_safe :
  forall a1 a2 cu,
    ABISpec a1 a2 -> check_abi_consumer a1 a2 cu = [].
```

The proof is one rewrite. The strict checker returns the empty list whenever the libraries are strictly compatible, and the filter has nothing to keep. Consumer-aware filtering can only ever shrink the verdict, never grow it.

8.6. Worked Example

This section runs the vtable-reorder case from Section 8.1 end to end and places the strict and consumer-aware reports side by side. A polymorphic class `Renderer` exports three virtual methods. In v2 the second and third are swapped, while the first keeps its slot.

```
// v1/lib.cc                              // v2/lib.cc (slots 1 and 2 swapped)
class Renderer {                          class Renderer {
public:                                   public:
  virtual int width()  {return 640;}        virtual int width()  {return 640;}
  virtual int height() {return 480;}        virtual int depth()  {return 24;}
  virtual int depth()  {return 24;}         virtual int height() {return 480;}
};                                        };
extern "C" {                              extern "C" {
  Renderer* create_renderer()               Renderer* create_renderer()
    { return new Renderer(); }                { return new Renderer(); }
  void destroy_renderer(Renderer* r)        void destroy_renderer(Renderer* r)
    { delete r; }                             { delete r; }
}                                         }
```

The strict check compares the two vtables slot by slot and reports both moved slots.

```
$ veriabi libv1.so libv2.so
  FAIL  Type layouts
        VTableSlotChanged (2):
          - Renderer slot 1 (expected: height, got: depth)
          - Renderer slot 2 (expected: depth, got: height)
  Result: INCOMPATIBLE
```

Two consumers link against v1. The first calls only `width()`, which keeps slot 0 in both versions. The second calls `height()`, which v1 places at slot 1.

```
// consumer_safe.cc
int main() {
  Renderer* r = create_renderer();
  int x = r->width();   // slot 0
  destroy_renderer(r);
  return x == 640 ? 0 : 1;
}
```

```
// consumer_affected.cc
int main() {
  Renderer* r = create_renderer();
  int x = r->height();  // slot 1
  destroy_renderer(r);
  return x == 480 ? 0 : 1;
}
```

Each consumer is compiled to LLVM IR with `clang -O1 -g` and passed as the third argument. For the first consumer the reader records a single vtable access at slot 0. Neither reported break touches slot 0, so the filter removes both and the verdict flips to compatible.

```
$ veriabi libv1.so libv2.so consumer_safe.ll
  Consumer analysis
    Virtual calls: 1
      Renderer slot 0
  PASS  Type layouts
  Result: COMPATIBLE
```

For the second consumer the reader records a vtable access at slot 1. The slot-1 break is kept and the slot-2 break is dropped, so the report shrinks from two breaks to one and the verdict stays incompatible.

```
$ veriabi libv1.so libv2.so consumer_affected.ll
  Consumer analysis
    Virtual calls: 1
      Renderer slot 1
  FAIL  Type layouts
        VTableSlotChanged (1):
          - Renderer slot 1 (expected: height, got: depth)
  Result: INCOMPATIBLE
```

The two consumer-aware runs share the same library pair and differ only in the consumer. The filter keeps exactly the break on the slot the consumer dispatches through and removes the rest. The first consumer is reported compatible because it never calls the reordered slots. The slot-1 and slot-2 breaks are real, and the first consumer is excused only because its recorded usage does not reach them.

8.7. Limitations

The IR reader is conventional C++ code and is part of the unverified side of the trust boundary, together with the ELF and DWARF readers. The risk is the mirror image of the strict side. On the strict side, the tool can wrongly blame a library by misreading binaries. On the consumer-aware side, it can wrongly excuse a break by misreading the consumer. All limitations below are in this unverified part of the system.

Scope of the soundness guarantee. The Rocq proofs establish that filtering is sound with respect to the strict verdict. The consumer-aware report is always a subset of the strict report, and the filter keeps exactly the breaks its relevance predicates accept. The proofs do not establish that a suppressed break is genuinely unobservable at runtime. The correspondence between a relevance predicate and real

consumer behaviour is a design argument, supported by the examples of Section 8.1, and it rests on the unverified front-end correctly recording what the consumer uses. A predicate that is too permissive, or a usage record that misses a real access, can suppress a break the consumer would in fact observe.

Toolchain coupling. The analysis consumes LLVM IR, which means the consumer must be compiled with `clang` or another toolchain that can emit compatible bitcode. Code built with GCC does not produce LLVM IR, so it is not directly supported. A future version could target GCC GIMPLE or work directly on stripped ELF binaries. This would remove the restriction, but it is outside the scope of the current implementation.

Optimisation-level sensitivity. The IR patterns matched by the `IRReader` depend on TBAA metadata that `clang` emits only when optimisation is enabled and strict aliasing is in effect. Virtual call detection in particular relies on the `vtable pointer` tag, which `clang` attaches to vptr loads under these conditions. The metadata is therefore present at `-O1` and above (including `-Og`) but absent at `-O0`, and it is suppressed at any level by `-fno-strict-aliasing`. The consumer must accordingly be compiled at `-O1` or higher with debug information and without `-fno-strict-aliasing`. At still higher optimisation levels the risk runs the other way, since calls may be inlined or devirtualised and the expected pattern may no longer exist. In practice, consumer code is analysed at `-O1` or `-O2` with debug information, which keeps both the metadata and the dispatch structure intact.

No inter-procedural reasoning. The usage analysis is purely intraprocedural. It looks only at patterns inside a single function. If a virtual call is hidden behind wrappers or forwarded through helper functions, it is not detected. Only direct call sites are recorded in the consumer usage model, so indirect flows are not captured.

Coarse member-access granularity. As described in Section 8.3, `cu_accesses_class_member` treats any access to a class as evidence that all of its member offsets are observable. This is sound but coarse. A more precise version would track the exact accessed offset and match it against the specific changed field. This would require consulting the v1 type layout to map offsets to fields. The refinement is straightforward and is part of future work.

Heap-allocation detection. The current implementation of `cu_allocates_type` only detects stack allocations through LLVM `AllocaInst` instructions with debug information, as described in Section 8.3. Heap allocations using `operator new` (`_Znwm`) and `operator new[]` (`_Znam`) are not yet detected.

This can lead to false negatives in the consumer-aware filter. If a struct size or alignment change is only used through heap allocation, the change may be incorrectly suppressed, even though the allocation still depends on the compile-time size of the type. A heap allocation bakes the size into the generated code in the same way as a stack allocation, either through pointer arithmetic or constructor dispatch.

The strict checker still reports the break correctly. Only the consumer-aware filtering is affected. The fix is to extend the IR reader to detect calls to `operator new` and `operator new[]` and recover the allocated type from debug information or from the constructor context. This change is local to the front-end and does not affect the formal model.

9

Evaluation and Discussion

This chapter addresses the five research questions introduced in Chapter 2, with each section focusing on one question. RQ1 and RQ3 were already addressed through the formal model and proofs presented in Chapter 6. Therefore, their evaluation sections are short and focus on what the evaluation adds. The remaining questions, RQ2, RQ4, and RQ5, are evaluated through experiments on a controlled benchmark corpus, a consumer example, and two production ASML libraries.

The evaluation inputs serve different purposes and should be interpreted accordingly. The controlled benchmark cases provide a known ground truth and are therefore used to evaluate correctness. The production libraries do not provide such a reference, meaning that these experiments are limited to measuring runtime, memory consumption, and output size. A higher number of reported diagnostics on an industrial library should not be interpreted as an indication of higher correctness.

Throughout the chapter `veriabi` is compared against two established tools, `libabigail`'s `abidiff` [27] and `abi_compliance_checker` [23]. Both are mature and widely used, and both differ from `veriabi` in important ways. In particular, neither tool is formally verified, both use a broader definition of ABI compatibility that includes checks outside the strict scope considered in this work, and their output formats differ significantly. The point of the comparison is not to claim that `veriabi` dominates either of them on every axis, but to characterise the trade-offs.

9.1. RQ1: Specifying the Breaking Changes

RQ1. Which changes in C++ libraries can cause ABI incompatibilities, and how can they be formally specified in a machine-checkable form?

The first part of this question is answered in Chapter 6, which lists the breaking changes across the symbol, signature, and layout layers and gives each one a machine-checkable rule in Rocq. The evaluation adds the evidence that these rules cover the changes that happen in practice. The controlled corpus of Section 9.2 groups its changes into five categories (S1–S5), and each documented change maps to one expected break in the manifest of Appendix A. Every in-scope change in the corpus matches a rule of the formal model, and the three out-of-scope changes mark where the current rules stop, at thread-local storage, array dimensions, and one anonymous-struct case. In answer to RQ1, the breaking changes are specified as the per-layer rules of Chapter 6, and the corpus shows these rules cover the changes a real library upgrade produces, apart from those three gaps.

9.2. RQ2: Detecting Breaks from Compiled Binaries

RQ2. How can we design and implement a checker that automatically detects these incompatibilities between two library versions, extracted directly from compiled binaries?

The design and implementation of the checker are covered in Chapters 5 to 7, from the extraction of ABI information out of ELF and DWARF to the verified decision procedure and the end-to-end

pipeline. What is left is to show that this checker actually detects the breaks present in a pair of compiled binaries. We run `veriabi`, `abidiff`, and `abi_compliance_checker` against a controlled corpus where every scenario has a known ground truth, either an enumerated set of ABI breaks of specified categories, or no break at all (a compatible refactor that the tool must not flag).

Setup. The corpus is the *abi_zoo* library, reproduced in full in Appendix A. It consists of a single pair of shared libraries with **27 documented ABI changes** between them. These are **21 breaks** (18 inside `veriabi`'s formal-model scope plus 3 deliberately out-of-scope) and **6 compatible refactors** used as negative controls. The 3 out-of-scope breaks (a thread-local variable type change, an exported array dimension change, and an anonymous struct that loses a member behind a typedef pointer) are included to exercise the DWARF extractor beyond what the formal model currently covers. One of the negative controls, a struct whose two same-typed members are reordered, is a deliberate case where the binary layout is unchanged. Because the ABI carries no member names (Section 3.1), a binary-layout checker must report it compatible, and it serves as a control rather than a break.

We chose to run the evaluation on a single library pair rather than a fragmented per-rule corpus because a real-world library upgrade always combines many changes at once. The ground truth is the manifest in Appendix A. Each change carries a unique numeric identifier, a category tag, and the expected `veriabi` break constructor (or the explicit note that the change is a compatible refactor). Each tool's output is scored against the manifest one change at a time.

Categories. Scenarios are partitioned into five categories that span the strict ABI rules of Chapter 6:

- **S1 Function signatures.** Return type, parameter list, variadicness (signature R1–R3).
- **S2 Struct and class layout.** Size, alignment, member offsets, member retypes (layout R3).
- **S3 Vtable and inheritance.** Vtable slot order, slot removal, base-class changes (layout R3, vtable and bases clauses).
- **S4 Symbols.** Exported names, kind changes, defined/undefined transitions, strength (symbol R1–R4).
- **S5 Types and enums.** Type-kind changes, base type widening, enum revaluations (layout R1, R2, R5).

Metrics. For each tool we report a single number, the count of documented breaks it catches. A break counts as caught when the tool's output mentions a diagnostic referring to the expected entity. We do not score the tools on the compatible refactors. `abidiff` and `abi_compliance_checker` are differential tools by design. They enumerate every change between two versions and let the user decide which ones matter, so they naturally report on compatible refactors too. Calling that a "false positive" would mischaracterise what those tools are built to do. `veriabi` differs in this respect. It commits to a formal definition of "break" and never prints anything that is not a break under that definition, but this is a difference of *intent* rather than correctness.

Results. Table 9.1 reports per-tool aggregate counts on *abi_zoo*, separated into the in-scope subset (18 breaks) and the full corpus (18 in-scope plus the 3 deliberately out-of-scope breaks). Table 9.2 breaks the in-scope counts down per category.

Table 9.1: Breaks caught per tool on *abi_zoo*.

Tool	In-scope breaks caught	Full-corpus breaks caught
`veriabi`	**18/18 (100%)**	19/21 (90%)
`abidiff`	16/18 (89%)	19/21 (90%)
`abi_compliance_checker`	15/18 (83%)	18/21 (86%)

Table 9.2: In-scope breaks caught per category.

Category	Breaks	`veriabi`	`abidiff`	`abi_compliance_checker`
S1 Function signatures	4	4/4 (100%)	4/4 (100%)	3/4 (75%)
S2 Struct and class layout	3	3/3 (100%)	3/3 (100%)	3/3 (100%)
S3 Vtable and inheritance	4	4/4 (100%)	3/4 (75%)	3/4 (75%)
S4 Symbols	4	4/4 (100%)	3/4 (75%)	3/4 (75%)
S5 Types and enums	3	3/3 (100%)	3/3 (100%)	3/3 (100%)
Total	18	**18/18 (100%)**	16/18 (89%)	15/18 (83%)

Discussion. On its claimed scope `veriabi` catches every break. This is the property formal verification is designed to deliver. When the rules are correctly defined, the mechanically checked rule set behaves exactly as the model promises. On the full corpus `veriabi` scores 19 of 21. It catches the anonymous-struct member removal (#27) but misses the two out-of-scope breaks the extractor does not yet reach, and neither is a model failure. Change #25 changes the type of a thread-local variable, which the object extractor does not yet read out of the dedicated TLS ELF section. Change #26 grows an exported array, which the `TypeDef` inductive does not yet record an array dimension for. Both are extensions of the DWARF reader that would not change any proof, and `abidiff` and `abi_compliance_checker` cover them in the meantime.

`abidiff` and `abi_compliance_checker` each catch fewer breaks. The breaks they miss are mainstream changes the formal model rules out. `abidiff` misses `Combo`'s base-class reorder and the strong-to-weak binding change on `__internal_helper`. `abi_compliance_checker` misses both of these plus `notify`'s variadic-flag toggle. These are concrete examples of the kind of break a formal model rules out. Each is a small, mainstream change that one of the two comparison tools simply does not look for. The comparison tools also report *additional* diagnostics on the six compatible refactors, such as a newly added function, a virtual appended at the end of a vtable, or a same-type member rename.

In answer to RQ2, the checker of Chapters 5 to 7 detects every in-scope break in the corpus directly from the compiled binaries, ahead of both comparison tools, and its only misses are the two out-of-scope changes that lie beyond the current extractor rather than beyond the model.

9.3. RQ3: Proving Soundness and Completeness

RQ3. Can formal methods be applied to prove the decision procedure sound and complete with respect to this specification?

This question is answered in Chapter 6. Every rule has a biconditional theorem stating that its checker returns no break exactly when the matching compatibility relation holds, and these combine into the top-level theorem `check_abi_correct`. Soundness and completeness are settled by the Rocq kernel, not by testing. The evaluation adds an independent check. On the controlled corpus, the proved checker flags every in-scope break and none of the six negative controls, which is what the soundness and completeness theorems say it should do. A tool that was only tested could pass the same corpus by chance. Here the corpus result and the proof agree, and it is the proof, not the corpus, that rules out the false positives and false negatives a corpus can only sample. In answer to RQ3, formal methods do prove the decision procedure sound and complete against the specification, and the evaluation agrees with that result.

9.4. RQ4: Narrowing the Verdict with Consumer Usage

RQ4. How can the tool take advantage of knowledge about actual consumer usage to distinguish between guaranteed breaks and incompatibilities that are harmless in practice for a given consumer?

The consumer-aware filter from Chapter 8 is evaluated using a controlled example rather than a large-scale industrial study, since the ASML libraries used in the industrial evaluation do not include consumer LLVM IR. Section 8.6 presents the complete workflow on a representative case. A polymorphic

class has its second and third virtual methods reordered between two versions, causing the strict checker to report two `VTableSlotChanged` breaks. Two consumers are then compiled against the first version of the library to obtain their LLVM IR.

The first consumer calls only a method whose slot the reorder does not touch. The checker records a single vtable access away from the reordered slots, filters both breaks, and the verdict turns from incompatible to compatible. The second consumer calls one of the reordered methods. The checker keeps exactly the break on the slot it uses and drops the other, so the report shrinks from two breaks to one and the verdict stays incompatible.

The demonstration shows the filter behaving as specified. It removes exactly the breaks the consumer cannot reach and keeps the rest, and two consumers sharing the same library pair receive different verdicts that match their usage. In answer to RQ4, consumer usage recovered from the compiled LLVM IR narrows the strict verdict to the breaks a given consumer can observe, as the two-consumer case shows. A quantitative evaluation on production consumers is left to future work, since it would require consumer LLVM IR that the current industrial corpus does not include.

9.5. RQ5: Applicability at Industrial Scale

RQ5. Can the tool be applied to real-world C++ libraries used at ASML, and what are the practical limitations of this approach when faced with industrial-scale binary sizes?

The tool itself is built in Chapters 5 to 7. This section measures whether it runs on production ASML libraries and what limits show up at industrial binary sizes. The industrial evaluation runs the same three tools against production C++ libraries from ASML's internal codebase. Because these libraries are real-world rather than synthetic, no enumerated ground truth exists, and we cannot apply the breaks-caught metric of Section 9.2. What we can measure is scalability and the structure of each tool's output. We make no correctness claims.

Setup. We evaluated two ASML libraries: `foundation` and `geometrytoolbox` (a much larger library). For each library we ran the three tools on the same v1/v2 pair on a single workstation[1]. Runtime and peak resident memory were measured with `/usr/bin/time -v`. All three tools were run with their default flags. `veriabi` was run at its default fuel parameter of $f = 10$.

Library characteristics. Table 9.3 summarises each library at the ELF level, independent of any tool, namely file size and `nm -dynamic` export count (the count of symbols with binding `T`, `D`, `W`, `B`, or `R`). These numbers anchor the per-tool results below.

Table 9.3: Characteristics of the evaluated ASML libraries.

Library	v1 `.so` size	v1 exports	v2 exports
`foundation`	79 MB	13,231	39,144
`geometrytoolbox`	554 MB	77,749	227,471

Runtime, memory, output size. Table 9.4 reports wall-clock runtime, peak resident memory, the size on disk of each tool's output, and the tool's overall verdict (compatible vs incompatible per the tool's own classification). `abidiff`'s report on `geometrytoolbox` is roughly 150 MB. We therefore report only its summary statistics rather than attempting per-entity clustering.

Self-classified incompatibility counts. Each tool exposes its own "broken" marker. `veriabi` emits a `FAIL` line per failing checker block, `abidiff` annotates entries with an explicit "ABI incompatible" note, and `abi_compliance_checker` reports a per-severity (High/Medium/Low) problem count. Table 9.5 reports the resulting counts. The three columns measure different things and are not directly comparable. The intended reading is per-tool, as a count of entities each tool thinks should be checked by a human reviewer.

[1] An AMD EPYC 74F3 with 216 GiB RAM running Ubuntu LTS.

Table 9.4: Per-tool runtime, peak memory, output size, and verdict.

Library	Tool	Runtime	Peak RSS	Output size	Verdict
`foundation`	`veriabi`	1 m 8 s	8.6 GB	906 KB	incompatible
	`abidiff`	2 m 15 s	2.3 GB	19.8 MB	incompatible
	`abi_compliance_checker`	≈3 m 30 s	1.1 GB	270 KB	incompatible
`geometrytoolbox`	`veriabi`	4 m 13 s	10.2 GB	350 KB	incompatible
	`abidiff`	2 h 11 m	15.2 GB	147 MB	incompatible
	`abi_compliance_checker`	≈30 m	5.8 GB	1.0 MB	incompatible

Table 9.5: Self-classified incompatibility counts per tool.

Library	`veriabi` break instances	`abidiff` removed + changed fns	`ACC` high-severity
`foundation`	428	263	1
`geometrytoolbox`	500	260	112

Per-tool category breakdowns. Tables 9.6–9.8 decompose each tool's `geometrytoolbox` count into its own taxonomy, by formal rule for `veriabi`, by diff direction for `abidiff`, and by severity for `abi_compliance_checker`. The three taxonomies are not equivalent, so the tables should be read down each, not across. The divergence in what counts as a category is itself a finding. The tools do not just disagree on counts, they disagree on what a category is.

Table 9.6: `veriabi` on `geometrytoolbox`, by rule category.

Rule category	Count
MissingSymbol	389
RttiChanged	6
ReturnTypeMismatch	1
ReturnLayoutChanged	17
ParamTypeMismatch	8
StructSizeMismatch	28
MemberRemoved	36
MemberTypeMismatch	5
UnionSizeMismatch	4
TypeKindChanged	2
BaseClassRemoved	2
BaseClassAdded	2
Total	**500**

Table 9.7: `abidiff` on `geometrytoolbox`, by diff direction.

Entity class	Removed	Changed	Added
Functions (debug)	196	64	113,365
Variables (debug)	0	0	59
Function symbols (no debug)	297	–	11,252
Variable symbols (no debug)	27	–	137

Removed symbols against ground truth. The three tools disagree sharply on how many symbols were removed between the two versions, which makes the raw counts in Table 9.5 hard to interpret. To anchor them, we computed a ground truth directly from the dynamic symbol tables. A symbol is *removed* if it is

Table 9.8: `abi_compliance_checker` on `geometrytoolbox`, by severity.

Problem class	High	Medium	Low
Removed symbols	92	–	–
Data-type problems	3	2	7
Symbol problems	17	2	2
Constants	–	–	0

defined in v1's `.dynsym` but absent from v2's. Table 9.9 compares each tool's removed-symbol count against this ground truth for `geometrytoolbox`.

Table 9.9: Removed exported symbols on `geometrytoolbox`: per-tool counts against the `nm` ground truth.

Source	Removed symbols
`nm`, all defined symbols (v1 not in v2)	544
of which strong (global binding)	37
of which weak	502
`abidiff` (debug + symbol-table removals)	≈520
`veriabi` (default)	389
`veriabi` (`--ignore-weak`)	41
`abi_compliance_checker` (high severity)	92

The raw removal count is dominated by weak symbols. Of the 544 v1-only entries, 502 are weak template instantiations the compiler emits opportunistically and that legitimately differ between builds, leaving only 37 strong-binding removals, the part a maintainer would treat as a real break. The three tools sit at different points on this spectrum. `abidiff` stays close to the raw total at 520 and surfaces all the weak noise. `abi_compliance_checker` filters hardest at 92 high-severity removals. `veriabi` with `--ignore-weak` reports 41, almost exactly matching the strong ground truth of 37 (the small surplus is RTTI typeinfo, which `veriabi` tracks as first-class symbols).

Runtime and memory. Table 9.4 gives both metrics for all three tools on both libraries. We read each metric across the tools in turn.

On runtime, `veriabi` is the fastest tool on both libraries. On `foundation` the three are close, at 1 m 8 s for `veriabi`, 2 m 15 s for `abidiff`, and about 3 m 30 s for `abi_compliance_checker`, where fixed overheads dominate. On the much larger `geometrytoolbox` the gap widens sharply, to 4 m 13 s for `veriabi`, about 30 minutes for `abi_compliance_checker`, and 2 h 11 m for `abidiff`, a roughly 31× speedup over `abidiff`. `abi_compliance_checker`'s own diff step is fast, and its cost is dominated by the `abi-dumper` pass that runs over each library first.

On peak memory the ordering is different. `veriabi` keeps the type registries of both library versions resident at once, so its peak is high. On `foundation` it uses 8.6 GB, the most of the three, against 2.3 GB for `abidiff` and 1.1 GB for `abi_compliance_checker`. On `geometrytoolbox` it uses 10.2 GB, between `abi_compliance_checker`'s 5.8 GB and `abidiff`'s 15.2 GB. So `veriabi` is consistently the fastest but never the lightest, trading memory for speed.

Output volume. `abidiff`'s report on `geometrytoolbox` is 147 MB, against 350 KB for `veriabi` and 1.0 MB for `abi_compliance_checker`, a ~400× ratio that puts `abidiff`'s output beyond what is practically reviewable as raw text. The pattern recurs on `foundation` at smaller scale (19.8 MB versus 906 KB). `abidiff` is differential by design and emits a record per difference, so its volume scales with the amount of symbols added and removed between library versions. `veriabi`'s output stays at kilobyte scale because its length tracks the number of *breaks*, not the number of *symbols*, and its namespace and weak-symbol filters can suppress further volume when needed.

In answer to RQ5, `veriabi` runs on both production ASML libraries and completes on the 554 MB

`geometrytoolbox` in about four minutes, an order of magnitude faster than `abidiff` and with a report two orders of magnitude smaller. The practical limits are its peak memory, which is the highest of the three tools on the smaller library, and its reliance on debug information, since a build that strips DWARF leaves nothing for the layout checker to compare.

9.6. Threats to Validity

Internal. Scoring whether a tool "caught" a break is done by searching the tool's output for an expected entity name or break constructor. This is a coarse rule. It can in principle give credit to a tool that mentions the entity for an unrelated reason. We mitigate this by inspecting the surrounding context manually whenever a hit looks suspicious, and the manifest in Appendix A uses entity names chosen to be unique enough that incidental mentions are rare.

External. The ASML libraries used in the industrial evaluation are not publicly available, which limits the reproducibility of the per-library numbers. The synthetic correctness corpus is fully reproducible. The *abi_zoo* sources, manifest, and build script are reproduced in Appendix A, and rerunning the evaluation regenerates each tool's output.

Construct. The strict ABI scope we evaluate against is narrower than the de facto ABI scope implemented by `libabigail` and `abi_compliance_checker`, which include diagnostics for changes that are technically not breaks under our formal model but might still matter to some users in some workflows. Reading the correctness numbers as "`veriabi` catches everything that matters" would over-interpret them. The correct reading is "`veriabi` catches everything inside the strict ABI scope, and the comparison tools catch a superset that also includes out-of-scope changes".

10

Conclusion

This thesis set out to answer whether it is possible to build an ABI compatibility checker for C++ shared libraries whose decision procedure is formally verified with respect to an explicit specification of compatibility. The answer it offers is concrete. `veriabi` is such a tool. Its specification of ABI compatibility is mechanised in Rocq, its decision procedure is proved sound and complete against that specification on a per-rule basis, and the resulting checker is extracted to OCaml and shipped inside a working C++ front-end that reads ELF and DWARF from real shared libraries. Each rule is paired with a concrete break example and the kind of binary-level failure it is meant to catch, so the formalisation stays connected to the systems-level behavior it models.

The evaluation supports a measured but real claim. On a controlled corpus where the ground truth is known by construction, `veriabi` catches every in-scope break, while `abidiff` and `abi_compliance_checker` each miss between two and three. The misses are not exotic. A base-class reorder, a strong-to-weak binding change, and a variadic-flag toggle are all exactly the kind of mainstream changes that a formal model rules out by construction. On the two production ASML libraries, the tool runs in minutes where the closest comparable tool takes hours, and it produces reports two orders of magnitude smaller.

10.1. Summary of Contributions

This thesis contributes:

1. **A mechanised specification of ABI compatibility** for a subset of the Itanium C++ ABI, covering the symbol, signature, and layout layers. The specification is small and readable, and it follows the structure of the binary contract it describes. Chapter 6 presents each rule with its motivation, its mathematical statement, its Rocq encoding, its decision procedure, and a concrete break example.
2. **A decision procedure proved sound and complete** against this specification. Each rule has a biconditional theorem, and these compose into a single theorem about the whole checker. The proofs use fuel-bounded induction over recursive type environments and a small library of reusable lemmas about filtering and membership.
3. **A C++ front-end** that reads ELF and DWARF from two shared libraries, builds a typed representation of their ABIs, and passes the result to the verified core through an OCaml-C bridge. It handles the DWARF quirks that any tool on production binaries must handle, such as forward declarations resolved across translation units, anonymous structs flattened into their enclosing type, and virtual inheritance.
4. **A consumer-aware extension**, also defined in Rocq, that takes the consumer's compiled LLVM IR as an extra input and narrows the strict verdict to the breaks the consumer would observe. The filter only removes breaks and never adds one, and its filtering is proved sound with respect to the strict verdict (Chapter 8).
5. **An evaluation** on a controlled corpus, the *abi_zoo* library with 21 documented breaks and 6 compatible refactors, and on two production ASML libraries, `foundation` and `geometrytoolbox`,

compared against `abidiff` and `abi_compliance_checker`. The *abi_zoo* sources, manifest, and build script are reproduced in Appendix A.

6. **An open-source release** of `veriabi` by ASML under the Mozilla Public License 2.0 [4], including the Rocq proofs, the C++ front-end, and the build infrastructure.

10.2. Limitations

Specification versus reality. The proofs establish that the checker matches the specification. They do not prove that the specification perfectly captures the Itanium ABI itself. The specification is still written by hand, based on the Itanium ABI document and on practical engineering knowledge. This is common in verified systems work. For example, CompCert proves correctness with respect to a formal model of C, not against every behaviour of real-world C compilers. In this thesis, the risk is reduced by keeping the specification relatively small and human-readable, by comparing the results against `abidiff` and `abi_compliance_checker` on the controlled corpus, and by checking the tool on real refactors from ASML libraries without observing unexpected diagnostics.

Scope of the formal model. The model covers symbol-level, signature-level, and layout-level breaks for a substantial subset of the Itanium C++ ABI. It does not model every ABI detail of the language. Exception unwinding tables (`.eh_frame`) are not included. Thread-local storage is handled only at the symbol level and not through the dedicated TLS ELF sections that contain the variable representation. Some compound types are also missing from the current `TypeDef` representation, particularly exported arrays whose dimensions are visible only through DWARF metadata. These limitations correspond to the two missed breaks on the *abi_zoo* corpus discussed in Section 9.2. They require extensions to the DWARF reader and the `TypeDef` representation rather than changes to the proofs themselves.

Front-end is unverified. The ELF/DWARF extractor and the LLVM IR reader are outside the verified part of the system. Bugs in these components can produce incorrect ABI representations, which may in turn lead to incorrect verdicts. The verified checker only guarantees correctness with respect to the extracted representation it receives as input. Errors in the proofs themselves would have been rejected by the Rocq kernel during proof checking. Keeping the extraction layer separate also makes it easier to replace or improve in future work without changing the verified core.

Consumer-aware filter limitations. Chapter 8 discusses several limitations of the consumer-aware analysis. These include dependence on LLVM IR, sensitivity to optimisation level, lack of inter-procedural reasoning, coarse member-access tracking, and the absence of heap-allocation detection. The last limitation currently causes false negatives in the filter. If a consumer only depends on a struct through `new` or `new[]` allocations, then a struct size or alignment change may be filtered out even though it is observable at runtime. The strict checker still reports the break correctly. The issue affects only the consumer-aware filtering stage.

Implementation choices outside the proofs. Some implementation choices are intentionally stricter than required by the ABI itself. For example, union members are matched by name in addition to structure. This can introduce false positives, but not false negatives. These choices are implementation decisions rather than consequences of the formal model.

10.3. Future Work

Five extensions stand out as natural next steps.

A refinement theorem against a formal Itanium semantics. The most significant extension would be a refinement theorem that connects the specification in this thesis to a formal operational semantics of the Itanium C++ ABI. Such a semantics is not currently available in a form that can be used directly in Rocq, and building it would already be a substantial research project.

With such a semantics in place, the guarantee provided by `veriabi` would change in nature. It would no longer only state that the checker matches the model, but that the model itself corresponds to actual

runtime behaviour of programs compiled against the ABI. This is similar in spirit to how CompCert relates its correctness theorem to a formal semantics of the target language. It would close the spec versus reality gap discussed in the Limitations section.

Extending the model to additional language features. Several missing language features can be added directly to the current specification without changing existing proofs. Thread-local variables that rely on the TLS section, exported arrays whose size is encoded in DWARF, exception unwinding tables (`.eh_frame`), and richer template specialisations are the main candidates.

Each of these fits as a small extension to the existing inductive definitions with localised proof updates. The thread-local and array cases are already visible in the evaluation as out of scope cases (Section 9.2), so they are natural first additions.

Consumer-aware extraction beyond LLVM IR. The consumer-aware analysis currently requires LLVM IR. Adding a second front-end for GCC GIMPLE would extend support to a wider range of C++ toolchains.

A more ambitious direction is to work directly on stripped ELF binaries and recover enough structure to reconstruct usage information without intermediate representations. A smaller immediate improvement in the current implementation is detecting heap allocations such as `operator new` and `new[]`.

Verified front-end. The ELF and DWARF readers and the LLVM IR reader are currently unverified C++ components. Verifying parts of this front-end would reduce the trusted base.

A practical starting point would be an ELF symbol table reader in Rocq. ELF is small and well specified, so it is suitable for early formalisation. A full DWARF reader would be more involved, but would directly produce the typed representation consumed by the verified core. The benefit is the same as above, reducing the space where bugs can affect results.

Refining the consumer-aware predicates. The current predicate for member layout access is intentionally coarse. Any access to a class is treated as evidence that all of its member offsets are observable.

A more precise version would track which offsets are actually accessed and relate them directly to changed fields. Combined with better detection of heap allocation sites, this would make the consumer-aware filter more precise without weakening soundness.

ABI compatibility is usually expected to be invisible in a software stack. New versions are shipped, existing consumers continue to run, and no changes are observed at runtime. When this assumption fails, the resulting errors are often silent at first and expensive to diagnose later.

The main contribution of this thesis is not to claim that ABI compatibility checking is solved, but to show that it can be treated in a structured way similar to other verified systems. The approach is to make the notion of compatibility explicit, implement a checker that decides it, and connect that checker to a formal proof of correctness. This provides a small but solid foundation on which further work can build, including closing the gap to an explicit ABI semantics, extending the model, and verifying more of the front-end.

References

[1] URL: https://dwarfstd.org/doc/DWARF5.pdf.

[2] *apidiff: Report Differences in the APIs of Two Versions of a Go Package*. URL: https://pkg.go.dev/golang.org/x/exp/cmd/apidiff (visited on 07/22/2026).

[3] Kevin Atkinson, Matthew Flatt, and Gary Lindstrom. "ABI compatibility through a customizable language". In: *SIGPLAN Not.* 46.2 (Oct. 2010), pp. 147–156. ISSN: 0362-1340. DOI: 10.1145/1942788.1868316. URL: https://doi.org/10.1145/1942788.1868316.

[4] James Bingen and Maximiliano Pin. *VeriABI: A Formally Verified ABI Compatibility Checker for C++*. 2026. URL: https://github.com/ASML-Labs/VeriABI (visited on 07/22/2026).

[5] *cargo-semver-checks: Lint Your Crate API Changes for Semver Violations*. URL: https://github.com/obi1kenobi/cargo-semver-checks (visited on 07/22/2026).

[6] Jens Dietrich, Kamil Jezek, and Premek Brada. "Broken Promises: An Empirical Study into Evolution Problems in Java Programs Caused by Library Upgrades". In: *Software Evolution Week – IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*. 2014, pp. 64–73.

[7] Alan A. T. Donovan et al. "Go at Google: Language Design in the Service of Software Engineering". In: *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH '12)*. Association for Computing Machinery, 2012, pp. 495–506. DOI: 10.1145/2384716.2384720.

[8] Darius Foo et al. "Efficient Static Checking of Library Updates". In: *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2018, pp. 791–796.

[9] GNOME Project. *API Stability*. https://developer.gnome.org/documentation/guidelines/maintainer/api-stability.html. (Visited on 05/27/2026).

[10] GNU Project. *ABI Policy and Guidelines*. https://gcc.gnu.org/onlinedocs/libstdc++/manual/abi.html. GNU libstdc++ documentation. (Visited on 05/26/2026).

[11] *japicmp: Comparison of Two Versions of a Jar Archive*. URL: https://github.com/siom79/japicmp (visited on 07/22/2026).

[12] Gerwin Klein et al. "seL4: Formal Verification of an OS Kernel". In: *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP '09)*. 2009, pp. 207–220. DOI: 10.1145/1629575.1629596.

[13] Nane Kratzke and Peter-Christian Quint. "Understanding Cloud-native Applications after 10 Years of Cloud Computing: A Systematic Mapping Study". In: *Journal of Systems and Software* 126 (2017), pp. 1–16. DOI: 10.1016/j.jss.2017.01.001.

[14] Ramana Kumar et al. "CakeML: A Verified Implementation of ML". In: *SIGPLAN Notices* 49.1 (2014), pp. 179–191. DOI: 10.1145/2578855.2535841.

[15] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California, Mar. 2004.

[16] Xavier Leroy. "Formal Verification of a Realistic Compiler". In: *Communications of the ACM* 52.7 (2009), pp. 107–115. DOI: 10.1145/1538788.1538814.

[17] Pierre Letouzey. "Programmation Fonctionnelle Certifiée: L'Extraction de Programmes dans l'Assistant Coq". PhD thesis. Université Paris-Sud, 2004.

[18] Linux man-pages project. *ld.so(8) – dynamic linker/loader*. https://man7.org/linux/man-pages/man8/ld.so.8.html. 2024. (Visited on 04/26/2026).

[19] Dirk Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux Journal* 2014.239 (2014), 2:1–2:2. ISSN: 1075-3583.

[20] NVIDIA Corporation. *ABI Compatibility — Carbonite SDK Documentation*. https://docs.omniverse.nvidia.com/kit/docs/carbonite/latest/docs/ABI.html. (Visited on 05/27/2026).

[21] Oracle Corporation. *Linker and Libraries Guide: ABI and Versioning*. https://docs.oracle.com/cd/E19120-01/open.solaris/819-0690/6n33n7fda/index.html. 2005. (Visited on 05/20/2026).

[22] A. Ponomarenko and V. Rubanov. "Backward compatibility of software interfaces: Steps towards automatic verification". In: *Programming and Computer Software* 38.5 (2012), pp. 257–267. ISSN: 1608-3261. DOI: 10.1134/S0361768812050052. URL: https://doi.org/10.1134/S0361768812050052.

[23] Andrey Ponomarenko. *ABI Compliance Checker*. 2019. URL: https://lvc.github.io/abi-compliance-checker/ (visited on 04/18/2026).

[24] Steven Raemaekers, Arie van Deursen, and Joost Visser. "Semantic Versioning versus Breaking Changes: A Study of the Maven Repository". In: *IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2014, pp. 215–224.

[25] *Revapi: An API Analysis and Change-Tracking Tool for Java*. URL: https://revapi.org (visited on 07/22/2026).

[26] Richard Smith, Daveed Vandevoorde, and Howard E. Hinnant. *A Module System for C++*. Tech. rep. P0142R0. ISO/IEC JTC1 SC22 WG21 – C++ Standards Committee, 2015. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0142r0.pdf.

[27] Sourceware Project. *Libabigail: The ABI Generic Analysis and Instrumentation Library*. 2025. URL: https://sourceware.org/libabigail/ (visited on 09/22/2025).

[28] The Itanium C++ ABI Project. *Itanium C++ ABI*. https://itanium-cxx-abi.github.io/cxx-abi/abi.html. 2019. (Visited on 05/20/2026).

[29] The LLVM Project. *LLVM Language Reference Manual*. URL: https://llvm.org/docs/LangRef.html (visited on 05/26/2026).

[30] The Rocq Development Team. *The Rocq Proof Assistant*. Version 9.1.1. 2025. URL: https://rocq-prover.org/.

[31] The Rust Project Developers. *The Rust Reference: Linkage*. https://doc.rust-lang.org/reference/linkage.html. 2025. (Visited on 04/18/2026).

[32] TIS Committee. *Executable and Linkable Format (ELF)*. Tech. rep. Portable Formats Specification, Version 1.2. Tool Interface Standard (TIS) Committee, 1995. URL: https://refspecs.linuxfoundation.org/elf/elf.pdf.

[33] Jianzhou Zhao et al. "Formalizing the LLVM Intermediate Representation for Verified Program Transformations". In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2012, pp. 427–440.

A

abi_zoo Reproducibility Package

This appendix reproduces the single-library corpus used as the controlled artifact in Chapter 9. The library is a single source file per version. Building each into a shared object gives the v1/v2 pair the evaluation runs against. Section A.1 lists the 27 documented changes between v1 and v2, and the two source files are reproduced verbatim afterwards.

A.1. Change manifest

The library contains **27 documented changes** between v1 and v2: **18 ABI breaks** across categories S1–S5, **6 compatible refactors** used as negative controls, and **3 out-of-scope breaks** that exercise the DWARF extractor beyond what the formal model currently covers.

Sections are listed in numerical-id order (matching the order the changes appear in `v1/lib.cc` and `v2/lib.cc` below), so the listing reads sequentially #1 through #27.

- **S4 – Symbols** (#1–#5: 4 breaks, 1 compat):
  - #1 `add_layer` removed
  - #2 compat: `compute_hash` added (pure addition)
  - #3 `init_state` changes from function to global integer (kind)
  - #4 `__internal_helper` binding strength `Strong` → `Weak`
  - #5 `Polygon` class renamed; `_ZTI7Polygon` typeinfo disappears
- **S1 – Function signatures** (#6–#10: 4 breaks, 1 compat):
  - #6 `get_count()` return type `int` → `long`
  - #7 `set_value` parameter type `int` → `double` (mangled name shifts)
  - #8 `log_event` gains a second parameter
  - #9 `notify` becomes variadic
  - #10 compat: `compat_widen` body change, signature identical
- **S2 – Struct and class layout** (#11–#15: 3 breaks, 2 compat):
  - #11 compat: `Point` members reordered with identical types, so the binary layout is unchanged (an API change, not an ABI break; see discussion below)
  - #12 `Color` channel width `uint8_t` → `uint16_t`
  - #13 `Header` gains a middle `subversion` field
  - #14 compat: `Coord` members renamed at source level only
  - #15 `Value` union grows by a `double` variant
- **S3 – Vtable and inheritance** (#16–#21: 4 breaks, 2 compats):

  - #16 `Shape` swaps vtable slots for `draw` and `area`
  - #17 compat: `Renderer` appends `flush()` at the end of the vtable
  - #18 `Sprite::set_color` removed; `update` shifts to slot 1
  - #19 `Diamond` removes `BaseB` from inheritance
  - #20 `Combo` reorders bases, shifting their offsets
  - #21 compat: `Child` reorders override declarations; vtable unchanged
- **S5 – Types and enums** (#22–#24: 3 breaks, 0 compats):
  - #22 `Mode` enum gains `Disabled = 0`; existing enumerators renumbered
  - #23 global `VERSION` object widens `int` → `long`
  - #24 `BUFFER_CAP` typedef `Size` widens `unsigned int` → `unsigned long`
- **Out-of-scope breaks** (#25–#27: 3 breaks): DWARF-extractor limitations rather than formal-model failures; see the discussion in Chapter 9.
  - #25 (S5) `per_thread_counter`: TLS variable type `int` → `char`
  - #26 (S5) `shared_buffer`: exported array dimension `[16]` → `[32]`
  - #27 (S2) anonymous struct via typedef pointer loses its `m2` member (the only one `veriabi` catches in practice; tagged out-of-scope because the libabigail regression suite shows it as an extractor blind-spot on a different DWARF shape)

Change #11 is an API change, not an ABI break. `Point` swaps the declaration order of two `int` members. The v1 layout is `{x@0, y@4}` and the v2 layout is `{y@0, x@4}`, which are byte-for-byte identical (two `int` slots at offsets 0 and 4). The ABI carries no member names, so this swap is indistinguishable at the binary layer from a pure rename, exactly like the `Color` example of Section 3.1. A binary-layout checker must therefore report it compatible, and `veriabi` correctly does, which is why it is counted as a compatible refactor rather than a break. The change is still observable to a consumer that accesses the fields by name, since a v1-compiled `p->x` reads the offset that now holds `y`, but that is an API incompatibility, not an ABI one. `abidiff` and `abi_compliance_checker` report it because they track member names, which is the differential behaviour discussed in Chapter 9.

A.2. v1/lib.cc

```cpp
// abi_zoo v1 -- baseline.
//
// Single library exercising every strict-ABI rule category.

#include <cstdint>
#include <cstddef>

//============================================================================
// S4 -- Symbols (5 changes)
//============================================================================

// [#1 break] Function present in v1, REMOVED in v2.
extern "C" int add_layer(int n) { return n + 1; }

// [#2 compat] New symbol added in v2; not present here.

// [#3 break] init_state is a function in v1, becomes a global int in v2 (kind
    change).
extern "C" int init_state(int seed) { return seed * 2; }

// [#4 break] __internal_helper has STRONG binding in v1; becomes WEAK in v2.
extern "C" int __internal_helper(int x) { return x; }

// [#5 break] Polymorphic class Polygon has typeinfo _ZTI7Polygon in v1;
```

```cpp
//             v2 renames the class to PolygonV2, so _ZTI7Polygon disappears.
class Polygon {
public:
    virtual ~Polygon() {}
    virtual int sides() const { return 0; }
};
extern "C" Polygon* make_polygon() { return new Polygon(); }


//============================================================================
// S1 -- Function signatures (5 changes)
//============================================================================

// [#6 break] Return type widens int -> long.
extern "C" int get_count() { return 42; }

// [#7 break] Parameter type changes int -> double (C++ mangled, name shifts).
int set_value(int x) { return x; }

// [#8 break] extern "C" parameter list grows by one.
extern "C" void log_event(const char* msg) { (void)msg; }

// [#9 break] extern "C" function becomes variadic.
extern "C" int notify(int code) { return code; }

// [#10 compat] Body changes but signature is preserved exactly.
extern "C" int compat_widen(int x) { return x + 1; }


//============================================================================
// S2 -- Struct and class layout (5 changes)
//============================================================================

// [#11 compat] Members reordered, same types: identical binary layout (API
    change only).
struct Point { int x; int y; };
extern "C" int point_x(const Point* p) { return p->x; }

// [#12 break] Member types widen 8->16 bits, struct size 4->8.
struct Color { std::uint8_t r, g, b, a; };
extern "C" std::uint8_t color_r(const Color* c) { return c->r; }

// [#13 break] New member inserted in the middle, shifting later offsets.
struct Header { int version; int flags; };
extern "C" int header_flags(const Header* h) { return h->flags; }

// [#14 compat] Member renamed at source level; binary layout identical.
struct Coord { int u; int v; };
extern "C" int coord_u(const Coord* c) { return c->u; }

// [#15 break] Union grows: new variant added increases union size.
union Value { int i; float f; };
extern "C" int value_i(const Value* v) { return v->i; }


//============================================================================
// S3 -- Vtable and inheritance (6 changes, 4 break + 2 compat)
//============================================================================

// [#16 break] Two virtuals swap order in v2.
class Shape {
```

```
public:
    virtual ~Shape() {}
    virtual void draw() const {}            // slot 1
    virtual double area() const { return 0; } // slot 2
};
extern "C" Shape* make_shape() { return new Shape(); }

// [#17 compat] Existing class gets a new virtual APPENDED at the end of the vtable
//              (slot index of every old method is preserved).
class Renderer {
public:
    virtual ~Renderer() {}
    virtual void clear() {}
};
extern "C" Renderer* make_renderer() { return new Renderer(); }

// [#18 break] A virtual is REMOVED, shifting all later slots up.
class Sprite {
public:
    virtual ~Sprite() {}
    virtual void set_color(int c) { (void)c; }   // slot 1 -- REMOVED in v2
    virtual void update() {}                     // slot 2 -> slot 1 in v2
};
extern "C" Sprite* make_sprite() { return new Sprite(); }

// [#19 break] Multiple inheritance: one base class is REMOVED in v2.
class BaseA { public: virtual ~BaseA() {} int a; };
class BaseB { public: virtual ~BaseB() {} int b; };
class Diamond : public BaseA, public BaseB { public: int d; };
extern "C" Diamond* make_diamond() { return new Diamond(); }

// [#20 break] Bases reordered, changing their byte offsets within the derived.
class BaseC { public: int c1, c2; };
class BaseD { public: int d1, d2; };
class Combo : public BaseC, public BaseD { public: int extra; };
extern "C" Combo* make_combo() { return new Combo(); }

// [#21 compat] Source reorders OVERRIDE declarations but they refer to the same
//              virtual slots inherited from the parent -- vtable identical.
class IParent {
public:
    virtual ~IParent() {}
    virtual void foo() {}
    virtual void bar() {}
};
class Child : public IParent {
public:
    void foo() override {}
    void bar() override {}
};
extern "C" Child* make_child() { return new Child(); }


//===========================================================================
// S5 -- Types and enums (3 changes)
//===========================================================================

// [#22 break] Enum used inside an exported struct; v2 renumbers values.
enum class Mode { Auto = 0, Manual = 1 };
struct Settings { Mode mode; int x; };
```

```
144 extern "C" Settings* make_settings() { return new Settings{Mode::Auto, 0}; }
145
146 // [#23 break] Global object type changes int -> long (S5 on object signature).
147 extern "C" int VERSION = 1;
148
149 // [#24 break] Base type widens via a typedef whose underlying type changed
150 //             (size 4 -> 8). Exported as object so it shows up.
151 using Size = unsigned int;
152 extern "C" Size BUFFER_CAP = 1024;
153
154
155 //============================================================================
156 // Out-of-scope breaks (3 changes) -- exercise the DWARF-extraction layer of
157 // veriabi that the formal model does not currently cover.
158 //============================================================================
159
160 // [#25 break / out-of-scope] Thread-local variable type changes int -> char.
161 extern "C" __thread int per_thread_counter = 0;
162
163 // [#26 break / out-of-scope] Exported array variable: dimension changes.
164 extern "C" char shared_buffer[16] = {0};
165
166 // [#27 break / out-of-scope] Anonymous struct accessed via a typedef pointer;
167 // v2 deletes a member of the anonymous struct, shrinking its size.
168 typedef struct { int m1; int m2; } *RecordPtr;
169 extern "C" RecordPtr make_record() { return nullptr; }
```

A.3. v2/lib.cc

```
1  // abi_zoo v2 -- 21 deliberate ABI breaks + 6 compatible refactors vs v1.
2  //
3  // Each numbered [#N] comment marks a change documented in the manifest.
4
5  #include <cstdint>
6  #include <cstddef>
7  #include <cstdarg>
8
9  //============================================================================
10 // S4 -- Symbols (5 changes)
11 //============================================================================
12
13 // [#1 break] add_layer REMOVED in v2.
14 // (no definition)
15
16 // [#2 compat] compute_hash ADDED in v2 (compatible: adding exports never breaks
       ).
17 extern "C" unsigned int compute_hash(const char* s) {
18     unsigned int h = 5381;
19     while (*s) h = h * 33 + (unsigned char)*s++;
20     return h;
21 }
22
23 // [#3 break] init_state changed from function to global int (kind change).
24 extern "C" int init_state = 0;
25
26 // [#4 break] __internal_helper now has WEAK binding.
27 extern "C" int __internal_helper(int x) __attribute__((weak));
28 extern "C" int __internal_helper(int x) { return x; }
29
30 // [#5 break] Polygon RENAMED to PolygonV2 -- typeinfo _ZTI7Polygon disappears.
31 class PolygonV2 {
```

```cpp
public:
    virtual ~PolygonV2() {}
    virtual int sides() const { return 0; }
};
extern "C" PolygonV2* make_polygon() { return new PolygonV2(); }


//============================================================================
// S1 -- Function signatures (5 changes)
//============================================================================

// [#6 break] Return type widens int -> long.
extern "C" long get_count() { return 42; }

// [#7 break] Parameter type changes int -> double. C++ mangled symbol shifts.
int set_value(double x) { return (int)x; }

// [#8 break] extern "C" parameter list grows from 1 to 2.
extern "C" void log_event(const char* msg, int level) { (void)msg; (void)level; }

// [#9 break] extern "C" function is now variadic.
extern "C" int notify(int code, ...) { (void)code; return code; }

// [#10 compat] Same signature; body uses a different but equivalent computation.
extern "C" int compat_widen(int x) { return x - (-1); }


//============================================================================
// S2 -- Struct and class layout (5 changes)
//============================================================================

// [#11 compat] Members reordered: x and y swapped (identical binary layout).
struct Point { int y; int x; };
extern "C" int point_x(const Point* p) { return p->x; }

// [#12 break] Channel width 8 -> 16, struct size 4 -> 8.
struct Color { std::uint16_t r, g, b, a; };
extern "C" std::uint16_t color_r(const Color* c) { return c->r; }

// [#13 break] New "subversion" field inserted between version and flags.
struct Header { int version; int subversion; int flags; };
extern "C" int header_flags(const Header* h) { return h->flags; }

// [#14 compat] Members renamed at source level; binary layout unchanged.
struct Coord { int first; int second; };
extern "C" int coord_u(const Coord* c) { return c->first; }

// [#15 break] Union gains a double variant, growing union size 4 -> 8.
union Value { int i; float f; double d; };
extern "C" int value_i(const Value* v) { return v->i; }


//============================================================================
// S3 -- Vtable and inheritance (6 changes: 4 break, 2 compat)
//============================================================================

// [#16 break] draw and area swapped: slot 1 is now area, slot 2 is draw.
class Shape {
public:
```

```
    virtual ~Shape() {}
    virtual double area() const { return 0; } // was slot 2, now slot 1
    virtual void draw() const {}              // was slot 1, now slot 2
};
extern "C" Shape* make_shape() { return new Shape(); }

// [#17 compat] New virtual flush appended AT THE END of the vtable.
class Renderer {
public:
    virtual ~Renderer() {}
    virtual void clear() {}
    virtual void flush() {}   // new virtual at end -- compatible
};
extern "C" Renderer* make_renderer() { return new Renderer(); }

// [#18 break] set_color removed; update is now slot 1 instead of slot 2.
class Sprite {
public:
    virtual ~Sprite() {}
    virtual void update() {}  // was slot 2 in v1, now slot 1
};
extern "C" Sprite* make_sprite() { return new Sprite(); }

// [#19 break] BaseB removed from Diamond's base list.
class BaseA { public: virtual ~BaseA() {} int a; };
class BaseB { public: virtual ~BaseB() {} int b; };
class Diamond : public BaseA { public: int d; };
extern "C" Diamond* make_diamond() { return new Diamond(); }

// [#20 break] Bases swapped in declaration order, changing offsets.
class BaseC { public: int c1, c2; };
class BaseD { public: int d1, d2; };
class Combo : public BaseD, public BaseC { public: int extra; };
extern "C" Combo* make_combo() { return new Combo(); }

// [#21 compat] Override order in source flipped; vtable unchanged because
//              both overrides target the same inherited slots.
class IParent {
public:
    virtual ~IParent() {}
    virtual void foo() {}
    virtual void bar() {}
};
class Child : public IParent {
public:
    void bar() override {}   // declared first now, but still slot 2
    void foo() override {}   // declared second, but still slot 1
};
extern "C" Child* make_child() { return new Child(); }


//============================================================================
// S5 -- Types and enums (3 changes)
//============================================================================

// [#22 break] Mode gains a Disabled = 0 enumerator; Auto and Manual renumbered.
enum class Mode { Disabled = 0, Auto = 1, Manual = 2 };
struct Settings { Mode mode; int x; };
extern "C" Settings* make_settings() { return new Settings{Mode::Auto, 0}; }

// [#23 break] VERSION widens int -> long. Object signature change.
```

```
extern "C" long VERSION = 1;

// [#24 break] Size typedef widens to unsigned long (size 4 -> 8).
using Size = unsigned long;
extern "C" Size BUFFER_CAP = 1024;


//============================================================================
// Out-of-scope breaks (3 changes) -- see v1 for context.
//============================================================================

// [#25 out-of-scope] TLS variable type int -> char.
extern "C" __thread char per_thread_counter = 0;

// [#26 out-of-scope] Array dimension grows 16 -> 32.
extern "C" char shared_buffer[32] = {0};

// [#27 out-of-scope] Anonymous struct loses its second member.
typedef struct { int m1; } *RecordPtr;
extern "C" RecordPtr make_record() { return nullptr; }
```

A.4. Build

```
CXX := g++
CXXFLAGS := -std=c++17 -g3 -gdwarf-5 -fPIC -shared

all: v1/libabi_zoo.so v2/libabi_zoo.so

v1/libabi_zoo.so: v1/lib.cc
	$(CXX) $(CXXFLAGS) $< -o $@

v2/libabi_zoo.so: v2/lib.cc
	$(CXX) $(CXXFLAGS) $< -o $@

clean:
	rm -f v1/libabi_zoo.so v2/libabi_zoo.so

.PHONY: all clean
```