UNIVERSITÀ
DI **PADOVA**

University of Padua, Department of Mathematics "Tullio Levi-Civita"
Master Thesis in Cybersecurity

# CIRO: Development of an Interpreter Aiding C++ Libraries Fuzzing

**Augusto Cesare Zanellato**
ID 2122416

Supervisor
**Prof. Losiouk Eleonora**
**University of Padua**

Co-supervisor
**Prof. Dr. Toffalini Flavio**
**Ruhr University Bochum**

Academic Year 2025/2026

# Contents

# Abstract

Fuzzing automatically generates and runs large numbers of inputs to trigger errors in programs under test. Testing a software library this way requires a harness: a small program that encodes a library usage pattern and feeds the fuzzer's inputs into the library. Because different harnesses exercise different functionalities and reach different library states, effective testing depends on generating many of them, and state-of-the-art tools do so automatically. This continuous production of harnesses comes at a cost: each one must be compiled. Compilation diverts computational resources away from input generation, and the generated code can reach hundreds of gigabytes on disk.

In this thesis we remove compilation from this loop. We present CIRO (C++ IR for Objects), an intermediate representation paired with a virtual machine, designed specifically to execute automatically generated C and C++ harnesses without recompiling them. Instead of compiling each new harness to native code, CIRO interprets it directly, turning a recurring per-harness compilation cost into a one-time investment. CIRO supports a subset of C and C++, can be easily integrated with existing harness generation tools, and has been evaluated on well-known libraries including LibTIFF, OpenCV, c-ares, and cJSON.

In our experiments, we find that CIRO achieves execution speeds comparable to those of native harnesses: on targets that spend most of each execution inside the library, interpreted harnesses match or even exceed native throughput, and even in the worst case observed, they retain more than 40% of it. At the same time, CIRO reduces per-harness disk usage by five to six orders of magnitude: each native harness used for testing occupied between roughly 320 and 390 MB of disk space, while the corresponding CIRO harness occupied between a few hundred bytes and a couple of kilobytes. Note that CIRO harnesses also require an interpreter executable, comparable in size to a single native harness, but this is built once per library and its cost is amortized over many harnesses.

# Sommario

Il fuzzing genera ed esegue automaticamente grandi quantità di input con l'obiettivo di provocare errori nei programmi in fase di test. Testare una libreria software in questo modo richiede un harness: un piccolo programma che codifica un pattern di utilizzo della libreria e le inoltra gli input prodotti dal fuzzer. Poiché harness diversi esercitano funzionalità diverse e raggiungono stati diversi della libreria, un testing efficace dipende dalla generazione di molti harness, e gli strumenti allo stato dell'arte li producono automaticamente. Questa produzione continua di harness ha però un costo: ognuno di essi deve essere compilato. La compilazione sottrae risorse computazionali alla generazione degli input, e il codice generato può raggiungere centinaia di gigabyte su disco.

In questa tesi eliminiamo la compilazione da questo ciclo. Presentiamo CIRO (C++ IR for Objects), una rappresentazione intermedia affiancata da una macchina virtuale, progettata specificamente per eseguire harness C e C++ generati automaticamente senza doverli ricompilare. Invece di compilare ogni nuovo harness in codice nativo, CIRO lo interpreta direttamente, trasformando un costo di compilazione ricorrente per ogni harness in un investimento una tantum. CIRO supporta un sottoinsieme del C e del C++, può essere integrato facilmente con gli strumenti esistenti per la generazione di harness ed è stato valutato su librerie note quali LibTIFF, OpenCV, c-ares e cJSON.

Nei nostri esperimenti osserviamo che CIRO raggiunge velocità di esecuzione paragonabili a quelle degli harness nativi: sui target che trascorrono la maggior parte di ogni esecuzione all'interno della libreria, gli harness interpretati eguagliano o addirittura superano il throughput nativo e, anche nel caso peggiore osservato, ne conservano oltre il 40%. Allo stesso tempo, CIRO riduce l'occupazione su disco per harness di cinque o sei ordini di grandezza: ogni harness nativo usato per i test occupava tra circa 320 e 390 MB di spazio su disco, mentre il corrispondente harness CIRO occupava tra poche centinaia di byte e un paio di kilobyte. Si noti che gli harness CIRO richiedono anche un eseguibile interprete, di dimensioni paragonabili a quelle di un singolo harness nativo, ma questo viene compilato una sola volta per libreria e il suo costo si ammortizza su molti harness.

# Introduction

Software reliability is a fundamental requirement for modern computing systems. Large code bases increasingly depend on third-party libraries to provide core functionality such as data processing and networking. This dependency landscape makes libraries a particularly important target for testing, since defects in widely reused components can propagate to many downstream projects. For example, Google funds OSS-Fuzz [23], a continuous fuzzing service for open source software, which has found thousands of bugs in popular libraries such as OpenSSL, SQLite, and FFmpeg.

Software fuzzing is an automated testing technique that generates large numbers of inputs for a program under test with the goal of exposing crashes, hangs, and other unexpected behaviors. Compared with manual testing, fuzzing can explore a much broader portion of the input space and can do so continuously with limited human intervention. For libraries, fuzzing is typically performed through harnesses, small programs that instantiate library objects, call library Application Programming Interfaces (APIs), and forward fuzzer-generated data to the code under test. The quality and diversity of these harnesses have a direct impact on the effectiveness of fuzzing: different harnesses expose different execution paths, state combinations, and corner cases. As a result, fuzzing a library with many varied harnesses is often desirable.

In practice, many fuzzing harnesses are still written manually. This gives developers precise control over the code under test, but it also requires time that could otherwise be spent fixing defects or improving the library. Automated harness generation tools reduce this manual effort by producing large corpora of test programs, yet they introduce a different engineering problem. A generator may emit many harnesses that are syntactically distinct but semantically similar, and each of them is usually compiled before it can be executed by a fuzzer. For C++ libraries, this compilation step can be expensive, especially when harnesses instantiate templates or statically link substantial parts of the target library. The combined effect is wasted CPU time, increased storage consumption, and lower fuzzing throughput.

This thesis investigates an alternative way to execute generated harnesses for C and C++ libraries without the need for compilation by introducing a dedicated Intermediate Representation (IR) and an interpreter for it. The central idea is to preserve the expressive power needed to model common library interactions while avoiding the cost of repeated compilation. To this end, we introduce CIRO, a C++ Intermediate Representation for Objects, designed to represent and execute relevant C and C++ constructs in a form suitable for fuzzing workflows. CIRO aims to bridge the gap between high-level harness generation and efficient runtime execution. Supporting C++ rather than only C is the

main challenge. For instance, templates are resolved and specialized at compile time, so an interpreter cannot instantiate them without generating and compiling new code; the standard library is itself heavily templated and largely inlined into its callers, leaving little stable surface for an interpreter to bind to; and C++ objects follow a far more intricate calling convention than plain C functions, involving name mangling, an implicit `this` pointer, overload resolution, and the passing of objects by value through the native Application Binary Interface (ABI). CIRO, therefore, supports the subset of these features that can be modeled without on-the-fly compilation, which is sufficient for a broad class of library-testing scenarios. From a scientific perspective, we contribute this low-level IR to model C and C++ APIs, supporting key language features such as data structures, classes, and function invocation including both C++ methods and C functions, together with the design of a Virtual Machine (VM) to efficiently execute fuzz drivers. The IR is designed to be human-readable and writable, yet unambiguous and suitable for being programmatically emitted by harness generation tools.[1] The VM is designed to be easily extensible to support new language features and to integrate with existing fuzzing tools without sacrificing performance. Our design enables more compact storage of generated harnesses and reduces the time spent in the compilation pipeline, allowing more resources to be devoted to the actual fuzzing process.

From an engineering perspective, we contribute techniques for extracting type and function information from C and C++ libraries using infrastructure provided by the compiler (Clang's libclang [29] and LibTooling [30]) and the build system (CMake's compile commands export feature), along with methods to automatically generate wrappers used by the interpreter. To improve the execution speed and reduce the overhead, we used tools such as perf [20] and Valgrind's Callgrind [4] to profile the VM in order to identify and optimize the most frequently executed code paths, reaching nearly native performance for interpreted harnesses. The interpreter also integrates with the libFuzzer and LibAFL [17, 8] fuzzing backends and supports features needed by harness generation tools such as libErator [33]. We study whether an interpreter-based approach can retain competitive execution performance while improving scalability in terms of storage and build time.

We evaluate CIRO by comparing interpreted harnesses against equivalent compiled harnesses across several properties. In our testing, the interpreted and compiled harnesses find exactly the same crashes, so the interpreter introduces neither false negatives nor spurious crashes. As a real-world case study, CIRO reproduces Heartbleed [6], showing that it can exercise production libraries and trigger known vulnerabilities.

Interpreted harnesses reach essentially the same code coverage as their compiled counterparts, staying within half a percentage point of native branch coverage on every target. Their fuzzing throughput is at parity with, or higher than, compiled harnesses on targets whose execution time is dominated by the target library, and remains above $40\,\%$ of native even in the worst case observed. Because harnesses are executed as scripts rather than compiled binaries, the per-harness build cost drops from seconds of compilation to a few milliseconds of script parsing, and the per-harness storage footprint drops from hundreds of megabytes to at most a few kilobytes.

[1] For example, when calling a function that has multiple overloads, a fully qualified name such as `poc::lib::sum_function(int,int)` is used.

## Thesis Outline

The rest of this thesis is organized as follows:

- Chapter 1 reviews prior work on automatic harness generation and situates CIRO among existing approaches;
- Chapter 2 introduces the background necessary to understand fuzzing, harness generation, and the fuzzing backends used by CIRO;
- Chapter 3 presents the design of CIRO and its supported language features;
- Chapter 4 describes the implementation details and integration with existing tooling;
- Chapter 5 evaluates the approach experimentally;
- Chapter 6 discusses the results and potential limitations of the approach, and outlines directions for future research and development;
- Chapter 7 summarizes the contributions of this thesis and reflects on the implications of the results.

# 1 Related Work

Automatically generating fuzzing harnesses for libraries has received considerable attention, since manually written drivers are costly to produce and tend to reach a coverage plateau. The central difficulty is to reconcile two competing goals: harnesses must be *diverse* enough to exercise many combinations of Application Programming Interface (API) calls and library states, yet *correct* enough to respect the intended usage of the API, so that the crashes they expose stem from genuine defects rather than from API misuse. One line of work derives harnesses from code that already uses the target library. FuzzGen [13] performs a whole-system analysis of a given environment, such as Debian or the Android Open Source Project, to infer a library's interface from all of its consumers; it builds an abstract API dependence graph and synthesizes libFuzzer drivers without human intervention. FUDGE [1] instead slices fragments of existing client code that invoke the target API, turning each fragment into a candidate driver paired with a build script and automatically scoring its quality; the approach scales to thousands of candidates, hundreds of which have been upstreamed into OSS-Fuzz. Because both systems learn from observed usage, the harnesses they produce are inherently bounded by the API patterns present in the available consumer code, and they cannot easily explore interactions that no existing client exercises.

A second line of work reduces or removes this dependence on consumer code. GraphFuzz [11] treats library fuzzing as a model-based problem: starting from a developer-provided specification, it models API calls as a lifetime-aware dataflow graph and synthesizes test cases by mutating that graph, so that object construction, use, and destruction remain consistent. OGHarn [24] dispenses with both consumers and hand-written specifications, extracting type and function information directly from the library's header files; it mutationally stitches together candidate harnesses and retains only those that satisfy a set of correctness oracles based on successful compilation, execution, and coverage gains, yielding a diverse set of semantically valid harnesses with very few false-positive crashes. libErator [33] similarly avoids consumer code and focuses on the allocation of computational resources: a dedicated static analysis of the library increases the likelihood of generating valid drivers, non-functional drivers are discarded early, and a selection step avoids redundant testing, striking a balance between the time spent generating drivers and the time spent fuzzing them.

Despite their differences, all of these approaches share the same execution model: every generated harness is compiled into a separate binary before it can be validated and fuzzed. This model has two drawbacks. Compiling many syntactically distinct but semantically similar harnesses leads to continuous recompilation that dominates the cost of

a campaign, and because each harness is a self-contained binary, near-identical drivers are rebuilt from scratch rather than sharing a reusable execution engine. For C++ libraries, the problem is aggravated, since template instantiation and the static linking of large portions of the target make each individual compilation expensive.

Hopper [5] avoids both drawbacks by redefining the execution model as *interpreter fuzzing*: instead of compiling generated harnesses, it describes API usage in a Domain Specific Language (DSL) and runs those descriptions with a single interpreter linked against the library under test, learning intra- and inter-API constraints to keep the generated programs meaningful. This removes the recompilation bottleneck and makes the interpreter reusable across harnesses. Nexzzer [16] adopts the same execution model, pairing an intermediate API description language, *Liblang*, with a general interpreter, and adds a hybrid relation-learning strategy that infers and progressively evolves implicit API relations, such as the requirement that an initialization function precede its consumers or that an object not be used after it is freed, which are not visible in function prototypes or in static analysis of consumer code. This allows it to generate diverse sequences while filtering out most API-misuse crashes. Both Hopper and Nexzzer, however, target C-style APIs and do not support C++ features such as classes, templates, and virtual calls. CIRO occupies the same interpreter-based, compilation-free niche, but is designed from the outset to represent and execute these C++ constructs, including method dispatch through the mangled C++ Application Binary Interface (ABI) and the passing of objects by value, which the existing C-oriented interpreters do not handle.

# 2 Background

## 2.1 Fuzzing

Fuzzing is a dynamic software testing technique that aims to discover bugs and vulnerabilities by executing a program with a large number of generated inputs [19]. Instead of relying only on manually written test cases, a fuzzer repeatedly produces inputs, runs the target program, and observes whether the execution exposes anomalous behavior such as crashes, assertion failures, memory leaks, undefined behavior, or hangs.

Fuzzing is effective because it automates the exploration of a program's input space, which is difficult to cover manually. A developer can usually write tests for the expected behavior of a program, but it is much harder to anticipate all malformed inputs, unusual state combinations, and corner cases that can appear in real deployments. By executing many inputs quickly, fuzzing can reach paths that were not considered during manual test design, and it has therefore become a practical technique for finding vulnerabilities in widely used software.

Figure 2.1 summarizes the coverage-guided loop that a fuzzer repeatedly executes. The fuzzer maintains an input corpus, from which an input generator derives new test cases; each generated input is then run during the execution stage, and the resulting behavior is examined by an analysis stage. The analysis reports coverage information back to the input generator, which uses it to bias subsequent inputs toward code paths that have not yet been explored. The work in this thesis is mainly concerned with executing harnesses inside this loop rather than with designing a new fuzzer, so the execution model it introduces must fit the stages shown here.

The literature on fuzzing is extensive, with multiple possible approaches to input handling. The simplest strategy is blackbox fuzzing, where inputs are generated without any knowledge of the target program or of the expected input format [19]. This strategy is easy to implement, but it is often inefficient for structured inputs: most random inputs are rejected during parsing or validation, so little of the deeper program logic is exercised. Modern fuzzers therefore often use greybox, or coverage-guided, fuzzing [34, 2].

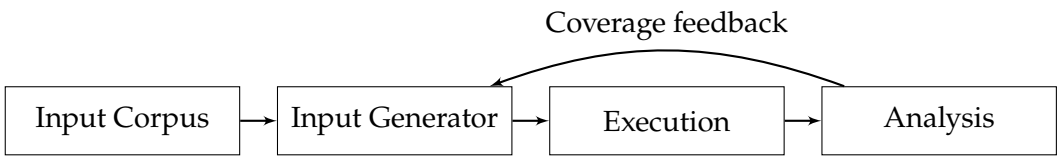



**Figure 2.1:** Fuzzing loop

In this setting, the fuzzer observes which code paths were executed by each input and gives priority to mutations that increase coverage. This feedback loop keeps the search focused on inputs that expose new behavior rather than on purely random exploration.

Coverage guidance is commonly combined with mutation-based fuzzing. Starting from an initial corpus of valid or partially valid inputs, the fuzzer creates new test cases by applying transformations such as bit flips, byte insertions, deletions, and substitutions [7]. Inputs that trigger new coverage are retained in the corpus and become starting points for later mutations. Another family of approaches, generation-based fuzzing, builds inputs from a grammar or a model of the expected format [26, 12]. This can reach deeper code paths when the input structure is known, because generated inputs can satisfy constraints that random mutations would violate. More advanced systems may also use taint analysis or concolic execution to reason about which input bytes influence program behavior and how to satisfy difficult path constraints.

## 2.2 Sanitizers and Coverage Instrumentation

By instrumenting the target program during compilation and adding runtime checks, sanitizers detect classes of bugs that may otherwise remain silent. Fuzzers are most useful when faults are detected as soon as the triggering input is executed. In C and C++ fuzzing, the most common sanitizers include:

- **AddressSanitizer (ASan)**, which detects out-of-bounds accesses, invalid `free`, use-after-free, use-after-return, and memory leaks [22];
- **UndefinedBehaviorSanitizer (UBSan)**, which detects undefined behavior such as integer overflow and null pointer dereferences [32];
- **MemorySanitizer (MSan)**, which detects reads from uninitialized memory [28].

When a sanitizer detects an error, it reports the issue and typically terminates the program. The fuzzer can then preserve the input that caused the failure, allowing the bug to be reproduced and minimized.

Coverage instrumentation, instead, records which parts of the target program are executed by each input. Clang's SanitizerCoverage [31] is commonly used for this purpose. The compiler inserts instrumentation that records executed basic blocks or control-flow edges, and the fuzzer uses this information as feedback for future mutations. This is the mechanism that allows coverage-guided fuzzers to prefer inputs that explore new behavior.

For CIRO, we use these two forms of instrumentation because interpreted harnesses still need to participate in a standard fuzzing workflow. The target library must expose failures through sanitizers, and the fuzzing backend must receive enough execution feedback to guide input generation.

## 2.3 Fuzz Harnesses and Harness Generation

A fuzz harness is a small program that connects the fuzzer to the code under test. Instead of fuzzing an entire application end-to-end, the harness isolates a function, method, or component of a target library and translates the raw bytes produced by the fuzzer into

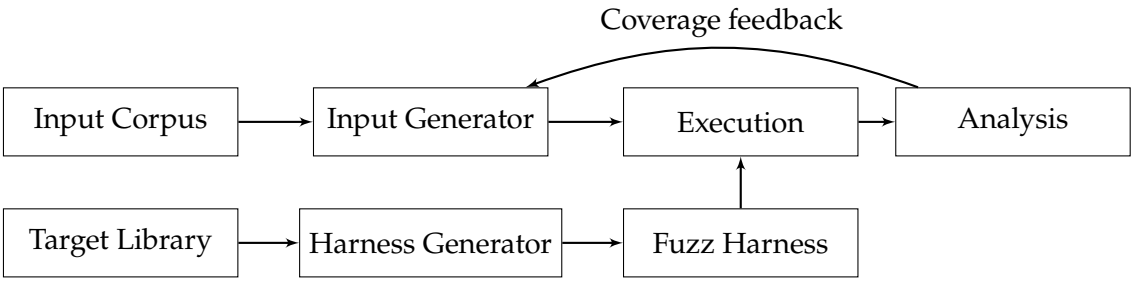



**Figure 2.2:** Fuzzing loop with automated harness generation

meaningful calls to that code. A typical harness reads the fuzzer input, deserializes it into values expected by the target API, invokes the target functionality, and then performs any required cleanup or state reset.

Harness indirection matters because libraries are rarely executable on their own. Their behavior depends on how clients instantiate objects, call functions, and combine operations. A good harness therefore exposes a useful portion of the library interface while avoiding unnecessary work such as network access, filesystem access, or unrelated application setup. The harness also determines which states can be reached during fuzzing, so harness quality directly affects the effectiveness of the campaign.

Harnesses can be written manually, but doing so requires knowledge of both the target library and the fuzzing framework. In the libFuzzer ecosystem, for example, a harness implements a function named `LLVMFuzzerTestOneInput`, which receives a byte buffer from the fuzzer and calls the code under test. Manual harnesses give the developer control over this translation, but they are time-consuming to write and hard to scale to large libraries with many entry points.

Automated harness generation addresses this bottleneck by analyzing the target library and emitting harnesses programmatically. A generator can extract function signatures, infer how inputs should be deserialized, and produce boilerplate compatible with the selected fuzzing framework. This makes it possible to fuzz many parts of a library without writing each harness by hand. At the same time, it creates the problem that motivates CIRO: generated harnesses are usually compiled before they can be executed, and compiling many similar C++ harnesses can dominate the cost of a fuzzing workflow.

Figure 2.2 shows where automated harness generation sits in the fuzzing loop: a harness generator analyzes the target library and emits the fuzz harness that the execution stage runs, while the input generator keeps supplying inputs from the corpus. CIRO changes only the execution part of this pipeline rather than its overall shape. The harness generator still describes how to exercise the target library, but the generated program is represented in an intermediate form that can be interpreted instead of compiled.

## 2.4 Fuzzing Backends Used by CIRO

CIRO is designed to be compatible with existing fuzzing infrastructure, so it does not implement a new fuzzer from scratch. Instead, it focuses on executing generated harnesses without compiling them into native binaries, while still integrating with the familiar workflows of libFuzzer [17] and LibAFL [8].

libFuzzer [17] is an in-process, coverage-guided fuzzing library integrated with LLVM.

A libFuzzer target is linked with the target code and the harness, and the fuzzer repeatedly calls the harness inside the same process. This avoids the overhead of starting a new process for each input and makes it possible to execute many test cases per second, while crashes and sanitizer failures are observed immediately by the fuzzer. Its main limitation is that it is a relatively fixed in-process engine and the project is in maintenance mode, with no new features planned [17].

LibAFL [8] is a more modular fuzzing framework written in Rust. Instead of exposing a single fixed architecture, it separates the main parts of a fuzzer into reusable components such as corpora, feedback mechanisms, schedulers, mutators, executors, and observers. This design allows a fuzzing setup to be adapted to different targets and experiments without rewriting the entire loop. Compared with libFuzzer, this flexibility makes LibAFL better suited to experimental integrations, but it also introduces more setup complexity and requires careful component selection.

# 3 Design

CIRO is designed to be a drop-in replacement for compiled harnesses in existing fuzzing workflows. It is not designed as a standalone fuzzer. Its role is to execute generated harnesses without compiling each one into a separate native binary, while still integrating with existing fuzzing infrastructure. A harness generator must describe allocations, input decoding, calls to the target API, and cleanup logic, but the description can be executed by a reusable interpreter instead of being compiled into a new binary for every candidate harness. The main design problem is therefore to define an intermediate representation that is compact enough to generate and interpret efficiently, while still preserving the low-level control required by native C and C++ library fuzzing.

Native library fuzzing imposes constraints that are different from those of ordinary scripting languages. A useful harness representation must allocate objects with the correct size and alignment assumptions, pass arguments through the native ABI, select overloaded C++ functions unambiguously, model constructors and destructors, and allow sanitizers to observe real memory misuse. CIRO addresses these requirements through a small harness language, a metadata extraction pipeline, and an interpreter that executes scripts against the target library in the same process as the fuzzer.

## 3.1 Design Objectives

We designed CIRO aiming to achieve the following objectives:

**(O1) Being Minimal.** The interpreter should not be a full programming language. The implemented type system is intentionally simple, delegating correctness concerns to the harness generator. This allows the interpreter to omit most safety checks, resulting in faster execution.

**(O2) Semantic Compatibility with C and C++.** The Intermediate Representation (IR) should be able to express the same harnesses that would be written in C or C++. The interpreter should not change the semantics of the target library, and it should not hide low-level details such as raw pointers and memory ownership.

**(O3) Integration with Fuzzing Backends.** The interpreter should be able to integrate with existing fuzzing backends, such as libFuzzer and LibAFL. The harnesses should be able to consume raw bytes from the fuzzer and report coverage information back to the fuzzer.

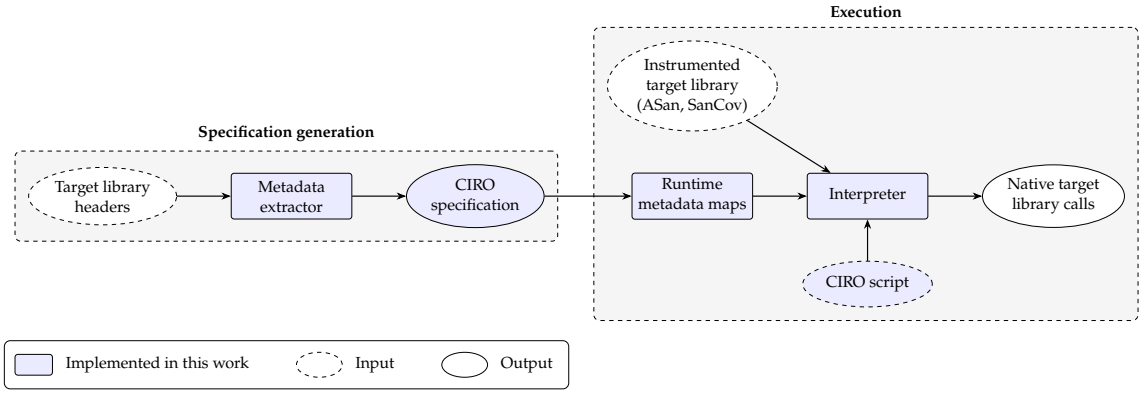



**Figure 3.1:** High-level CIRO architecture

**(O4) Performance.** The interpreter should be fast enough to not be a bottleneck in the fuzzing process. It should be able to execute harnesses with minimal overhead compared to compiled harnesses.

## 3.2 System Architecture

Figure 3.1 shows the CIRO architecture, which separates three concerns that are normally mixed inside a compiled harness. The script describes what the harness should do, generated metadata describes how target declarations can be called, and the interpreter provides execution state, input consumption, memory management, and diagnostics.

Target headers and build configuration are the source of static information. The extractor reads declarations with Clang-based tooling and emits C++ code that registers types, functions, methods, constructors, destructors, fields, and Foreign Function Interface (FFI) layouts. The generated code is compiled and linked into the interpreter or fuzzer binary, where static registration functions populate runtime maps.

Runtime execution begins from a script rather than from a generated C++ source file. The parser lowers the script into a flat sequence of instructions, and the interpreter executes those instructions over concrete fuzzer input. Native calls are performed through metadata in the runtime maps, so the interpreter core is independent of the particular target library selected at build time.

## 3.3 Harness Language

The CIRO script language is a small DSL for describing library harnesses. The language is intentionally closer to a low-level intermediate representation than to a general-purpose programming language, in line with objective **O1**. A script can allocate typed objects, allocate untyped pointer variables,[1] consume bytes from the fuzzer input, construct objects, call functions and methods, access fields and arrays, perform simple integer arithmetic, branch with labels and jumps, and free owned variables.

**Function Calling.** Listing 3.1 illustrates how function calls are made in CIRO. The callee is identified by a complete signature string so that the runtime can distinguish overloaded

[1] Untyped pointer variables are used to represent pointers to opaque types, akin to `void*`.

```
ret = "::ares_init(ares_channeldata **)"(&channel)

fn_alias ares_init "::ares_init(ares_channeldata **)"
ret = ares_init(&channel)
```

**Listing 3.1:** Function Calls in CIRO

```
alloc m_1 ::cv::Mat

type_alias mat ::cv::Mat
alloc m_2 mat
```

**Listing 3.2:** Types in CIRO

functions and namespaced declarations without performing C++ name lookup inside the script language, preserving the unambiguous overload selection required by objective **O2**. Full signature strings are precise but verbose. The language also supports aliases that make handwritten scripts easier to write while keeping runtime lookup unambiguous. Function aliases add no runtime overhead because they are resolved at parse time.

**Types.** Type names follow the same philosophy as function names. As shown in Listing 3.2, type keys are normally fully qualified names such as `::cv::Mat` or `::int`. The name qualification avoids ambiguity between libraries and namespaces, and it keeps the language parser simple because type resolution is delegated to the runtime metadata maps. As with functions, type aliases can be used to make handwritten scripts more concise.

**Object Construction.** Listing 3.3 illustrates how objects are constructed in CIRO. The interpreter uses an explicit signature to find the correct constructor to call, and it passes a backing buffer as an implicit first argument to the constructor.

**Method Calling.** Listing 3.4 illustrates how method calls are made in CIRO. Methods are represented as function entries with an implicit object argument, and the first argument to a method call is always the object on which the method is invoked. The interpreter uses the object's runtime type to resolve the correct method to call.

**Input Handling.** Listing 3.5 illustrates how input is handled in CIRO. Harness scripts must explicitly describe how they are consuming input. The interpreter does not infer how fuzzer bytes should be split into arguments. A script must state which allocated buffers receive input bytes, which values are constructed, and which cleanup operations are required after the target call.

```
// data_matrix declaration ignored for brevity
alloc data_matrix_conv ::cv::_InputArray
construct data_matrix_conv "const cv::Mat &" (data_matrix)
```

**Listing 3.3:** Object Construction in CIRO

```
alloc adder ::poc::lib::Adder
construct adder "int,int" (10, 20)
ret = adder."add(int,int) const"(2, 5)
```

**Listing 3.4:** Method Calling in CIRO

```
alloc size ::size_t
size = 64
alloc data ::uint8_t size
consume data
alloc_ptr out

"::target_parse(const uint8_t *,size_t,void **)"(data, size, &out)

free size, data, out
```

**Listing 3.5:** Input Consumption in CIRO

Other possible ways of handling input include allocating a buffer containing all the remaining fuzzer bytes and constructing a C++ `std::istringstream` from that buffer. These are described in detail in Appendix A.

**Control Flow.** As shown in Listing 3.6, control flow is deliberately minimal, in keeping with objective **O1**. Labels and conditional jumps are enough to express loops, bounds checks, and cleanup paths while preserving the flat instruction representation used by the interpreter. Structured constructs such as `if` and `while` are not implemented as keywords because they can be expressed with labels and jumps, as assembly languages do.

**Pragmas.** Pragmas are special instructions that configure the interpreter before execution starts, such as instantiating variadic functions with a callable signature or setting parameters for the structure-aware input mutator. Details about pragmas are described in Appendix A.

## 3.4 Generated Metadata

Metadata are information about the target library, more specifically about the target library's API composed of functions, types and their associated methods, constructors,

```
alloc i ::int
i = 3
i_loop_start:
print "i is: \n"
hexdump i
i--
jump nz i i_loop_start
free i
```

**Listing 3.6:** Control Flow in CIRO

destructors and fields. The metadata is needed to bridge the gap between the target library and the interpreter, which is target-independent. One of CIRO's strengths is that it does not require handwritten bindings for every library function; this is achieved by having an automated metadata extractor that generates C++ code registering entries into global maps for functions, variadic functions, and types. Type entries contain constructors, methods, the destructor, public field accessors, and the libffi representation of the record layout.

Automated metadata generation is essential for scalability. Adding a target library should mainly require making the library available as a build target, selecting headers for extraction, linking the generated metadata into the interpreter or fuzzer, and writing scripts that exercise useful API sequences. Target specificity remains in generated metadata, build configuration, and scripts rather than in the interpreter core.

The call boundary between the interpreter and the target library is implemented with libffi [15]. The library gives the interpreter a uniform way to call native declarations whose static C++ types are not known to the interpreter binary. The design avoids generating a custom call thunk for every signature, but it also makes correctness depend on accurate metadata and the correct implementation of ABI quirks. Records passed by value, references, function pointers, variadic functions, and complex C++ layouts require careful modeling or explicit exclusion.

Generated object files containing each target's metadata use functions marked with `__attribute__((constructor))` to automatically register function and type entries at program startup. These constructors run before `main()` and populate the global maps with the target's callable declarations. These are built and linked as static libraries. Since the object files are useful only for the registration side effects of their constructors, and no symbol in them is referenced directly, the linking step may discard their contents. Whole-archive linking is therefore used for all the generated code, forcing the object files to be retained so that their registration constructors run before script execution.

## 3.5 Memory and Value Model

To preserve the low-level semantics of C and C++ libraries, as required by objective **O2**, the interpreter must allocate concrete memory for objects, arrays, and buffers. The interpreter uses a descriptor-based model to represent runtime values. Each variable is represented by an instance of the struct `runtime::pointer` that contains the address of the allocated memory, its size, and optional type metadata. In case a variable is an array, the descriptor contains the number of elements, the size of each element and the total size.

The descriptor-based model fits the native fuzzing domain because most operations eventually need an address and a size. Method dispatch needs the runtime type, array access needs element size and count, destructor dispatch needs ownership and type metadata, and function calls need to distinguish passing a pointer value from passing the address of a pointer-sized cell.

Memory management is handled by generally allocating the variables on the heap unless they are views into memory owned by another variable or can be safely allocated on the stack, which is usually done for some small, temporary values. Typed allocations use the size stored in runtime type metadata; pointer allocations reserve pointer-sized cells; strings are null-terminated buffers. Using the heap as backing storage allows sanitizers

to observe the same classes of memory misuse that would occur in a compiled harness without having ASan instrumentation on the interpreter itself. Variable lifetime management is also explicit; this means that a script should free variables when it wants to end their native lifetime. Freeing a typed variable calls its registered destructor before releasing the underlying allocation; invalid frees are ignored to simplify cleanup paths after conditional execution. To prevent memory leaks caused by runtime errors (crashes and sanitizer errors), the interpreter state destructor still frees any valid variables that remain at the end of execution.

In order to reduce the overhead of heap allocations (objective **O4**), we experimented with inlining small allocations under 8 bytes into the pointer descriptor itself,[2] but the performance gains were negligible and the resulting complexity increase was significant. The gain is likely so small because the GNU heap allocator implementation already has heavy optimizations in place for small allocations.

## 3.6 Interpreter Execution Model

The interpreter execution model is designed to separate the initialization of the interpreter from the execution itself, in pursuit of objective **O4**: this allows parsing the script and building the program object once, and then executing it multiple times with different inputs, each having its own independent state.

The parser lowers a script into a program object whose main component is a flat vector of instructions. A program counter indexes this vector during execution. Flat instruction execution keeps jumps simple, lets labels resolve to instruction positions, and allows the fuzzer to parse the script once and reuse the same program for every input.

Variables are resolved to numeric slots before execution; this reduces the overhead of variable lookups from string hashing and map access to a simple array index. The parser maintains the mapping from variable names to indexes, and each runtime state stores a vector of pointer descriptors plus validity flags.

Expression evaluation uses an explicit result stack. An expression can either return a view into existing memory or write into a destination descriptor already placed on top of the stack. Assignment uses this mechanism by first evaluating the destination as an addressable location and then evaluating the source into that location. Temporary values are allocated as scratch memory and released after each root instruction.

Manual control flow follows naturally from the flat program representation. Conditional exits and aborts stop execution according to tests over arbitrary buffers or integer values. Conditional jumps update the program counter when a test succeeds. The language uses this low-level control model because harnesses primarily need validation checks, loops over input-derived values, and cleanup branches rather than general structured programming.

## 3.7 Execution Entry Points

CIRO provides two different entry points for executing harnesses. The first is a standalone interpreter that runs a script with optional input bytes, and the second is an entry point

[2] Such small allocations occur frequently in the interpreter when allocating temporary values.

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data,
                                      size_t Size) {
  // Call function under test with Data
  return 0;
}
```

**Listing 3.7:** libFuzzer Interface

compatible with existing fuzzing frameworks that executes the same parsed script for every generated input. The two entry points share the same interpreter core but expose it through different interfaces. A crash found during fuzzing can therefore be reproduced by running the same script and input through the standalone interpreter, assuming the same target libraries and instrumentation mode are used.

**Standalone Interpreter.** The standalone interpreter `IflInterpreter` is useful for deterministic reproduction, debugging, and differential tests. It can be invoked from the command line with a script and an optional input file, and it will execute the script with the provided input. The interpreter supports dumping debug information related to the script parsing along with printing a verbose execution trace, which includes the program counter, variable values, and any errors encountered during execution.

**Fuzzing Interpreter.** The fuzzing interpreter `IflFuzzer` exposes a libFuzzer-compatible entry point that executes the same parsed script for every generated input. This allows CIRO to integrate with existing fuzzing frameworks and leverage their input generation and coverage reporting capabilities, fulfilling objective **O3**. The fuzzer passes raw bytes to the interpreter, which then executes the script with those bytes as input. The interpreter can also report coverage information back to the fuzzer, allowing it to retain inputs that produce new coverage and continue the fuzzing process.

The central interface expected by libFuzzer is the `LLVMFuzzerTestOneInput` function as shown in Listing 3.7. The fuzzer passes raw bytes to this function, and the harness is responsible for translating them into calls to the target library:

This interface is important for CIRO because it is a de facto standard supported by many fuzzing frameworks. Preserving compatibility with this convention makes it possible to reuse existing fuzzing engines such as libFuzzer and LibAFL while changing only how the harness is executed.

The implementation presented in this thesis uses LibAFL's compatibility layer for harnesses that follow the libFuzzer convention. This keeps the interface close to the signature shown above while allowing the interpreter to be embedded in a more extensible fuzzing framework. The trade-off is that CIRO does not use the full flexibility of LibAFL. It relies on the compatibility layer so that development can focus on the CIRO interpreter and leave more specialized LibAFL features for future work.

The fuzzing interpreter also optionally supports structured custom mutation as needed by libErator [33]. The mutator configuration is script-driven via pragmas describing the expected input layout configuration inside the script itself, allowing the fuzzer to mutate dynamic fields while still preserving the required structure. Scripts that do not declare such a layout use the backend's normal mutation strategy.

## 3.8 Sanitizer Diagnostics

To help with triaging and debugging crashes, support for sanitizer-aware diagnostics has been added to the interpreter core. This makes it possible to connect native failures back to script execution and to report the current program counter, source line and variable values at the time of the crash.

When ASan support is available, the interpreter and fuzzer install a death callback that attempts to recover the current interpreter state. The recovery process is implemented using stack scanning because there is no way to provide a context variable to the sanitizer death callback, even though it is a fragile way to recover the state. The design assumes that the state object is located on the stack between the current function frame and `environ[0]`, which is conveniently placed in the stack after all the function frames. The stack scanning is implemented using a vendored version of `memmem` that is defined with `__attribute__((disable_sanitizer_instrumentation))` to prevent the various sanitizers from reporting errors about invalid stack accesses.

# 4 Implementation

The implementation of CIRO follows the separation introduced in Chapter 3. Target-specific information is generated ahead of execution, while script parsing and interpretation are handled by a reusable runtime. The repository therefore contains a build layer that prepares target libraries and metadata, an extraction layer that translates C and C++ declarations into registration code, a runtime layer that stores callable metadata, and an interpreter layer that executes CIRO programs for either deterministic reproduction or fuzzing.

The implementation is organized around reuse. The standalone interpreter and the fuzzer executable link the same interpreter library, use the same parser, and dispatch native calls through the same runtime metadata. The difference between reproduction and fuzzing lies mostly in the entry point: one mode reads a script and optional input file from the command line, while the other parses the script once and repeatedly executes it on fuzzer-generated inputs.

## 4.1 Build Organization

To orchestrate the build process and manage dependencies, we chose to use the CMake build system because it provides a robust and flexible framework for managing complex builds, including handling custom code generation tasks as part of the build graph, both as dependencies and as dependees. The top-level configuration builds the runtime support library, the LibTooling-based layout extractor, generated ANTLR parser code, the interpreter library, generated instrumentation libraries for selected targets, and either the standalone interpreter or the fuzzer target. The reusable interpreter library links parser machinery, interpreter code, runtime metadata support, libffi, and formatting utilities.

Target-library selection is controlled through the `TARGET_LIBRARIES` option. An empty list enables all configured targets, while an explicit semicolon-separated list restricts the build to selected libraries as described in Table 4.1. Selective builds are important because large C and C++ libraries can make metadata generation and instrumentation expensive, especially when sanitizer or coverage flags are enabled.

Build types and options propagate instrumentation choices to both CIRO and the target libraries. The `Sanitize` build type enables AddressSanitizer and UndefinedBehaviorSanitizer for the code under test. The `libFuzzer` build type compiles target libraries with fuzzer-compatible sanitizer flags and builds `IflFuzzer`. Optional coverage and debug-symbol settings are also forwarded to target-library builds so that fuzzing campaigns and coverage experiments observe the intended code.

**Table 4.1:** Supported target libraries.

| TARGET_LIBRARIES value | Library | Version | Description |
|---|---|---|---|
| OPENSSL_101F | OpenSSL[21] | 1.0.1f | Vendored copy of an OpenSSL version vulnerable to Heartbleed [6] |
| OPENSSL | OpenSSL[21] | System provided | Used for some differential tests |
| RE2 | RE2 [10] | 2025-11-05-1-ge7aec59 | Used for some differential tests |
| POCLIB | Toy library | N/A | Used for some differential tests |
| OPENCV | OpenCV [3] | 4.13.0-48-g55a8996ffe | Used as a fuzzing target |
| CARES | c-ares [27] | v1.31.0-317-g16c873c2 | Used as a fuzzing target |
| CJSON | cJSON [9] | v1.7.19 | Used as a fuzzing target |
| LIBTIFF | LibTIFF [14] | v4.7.1-54-g4f1fb36d | Used as a fuzzing target |

Different target libraries require different integration patterns. In-tree targets such as `poclib` are normal CMake subdirectories. cJSON is built directly from its C source file. RE2 and Abseil can be incorporated as subdirectories. OpenCV, c-ares, and LibTIFF are configured through external projects and exposed as imported static targets. System OpenSSL is discovered with CMake package logic, while OpenSSL 1.0.1f is represented by imported prebuilt static targets.

Instrumentation is exposed through three CMake functions. The default instrumentation function `interfuzz_instrument` extracts declarations from the include directories associated with a target. The manual instrumentation function `interfuzz_manual_instrument` performs the same task but limits extraction to explicit headers, which is useful for large libraries or headers that contain unsupported declarations. The linking function `interfuzz_link` links the generated metadata library into an executable with whole-archive semantics so that registration constructors are retained by the linker.

## 4.2 Runtime Metadata

Runtime metadata is implemented in the support library under `src/lib`. The main data structures represent functions, variadic functions, types, and interpreter pointers. Generated code and handwritten standard entries populate the same maps, so the interpreter does not need to distinguish target-provided functions from hand-instrumented utility functions once metadata has been registered.

As shown in Listing 4.1, `runtime::function` stores the information required for a libffi call: a prepared call interface, return type metadata, argument type metadata, argument storage, and a raw function pointer. Calling the wrapper ultimately invokes `ffi_call`. The wrapper owns dynamically allocated argument metadata and storage, so its destructor releases the arrays created during registration.

```
struct function {
  ffi_cif cif;
  ffi_type *ret_type;
  ffi_type **arg_types;
  size_t num_args;
  void **args;
  void (*func_ptr)(void);
};
```

**Listing 4.1:** `runtime::function` structure

```
struct variadic_function {
  ffi_type *ret_type;
  ffi_type **fixed_arg_types;
  size_t num_fixed_args;
  void (*func_ptr)(void);
};
```

**Listing 4.2:** `runtime::variadic_function` structure

Variadic functions use a separate runtime representation because libffi needs the concrete variable arguments before it can prepare a call interface. A script-level `#variadic` pragma provides the missing argument types. The visitor instantiates the variadic metadata with those concrete types and inserts the resulting ordinary function into the runtime function map.

As shown in Listing 4.3, `runtime::type` stores the metadata associated with a C or C++ type. A type entry contains the object size, constructor map, method map, optional destructor callback, public field accessors, and a libffi struct descriptor. Constructor wrappers use placement new and receive the destination buffer as their first argument. Methods are represented as function entries with an implicit object argument.

`runtime::pointer` is the interpreter's universal memory descriptor. As shown in Listing 4.4, the descriptor stores a raw address, total byte size, element count, element size, optional runtime type, and a flag indicating whether the allocation is a pointer-sized cell. Function-call preparation uses this flag to distinguish a pointer value from the address of a cell containing a pointer value, which is necessary for pointer-to-pointer APIs.

Global metadata maps store registered functions, variadic functions, and types. The accessors are function-local statics, which avoids initialization order problems between generated object files and runtime code. Constructor priorities order the process so that

```
struct type {
  size_t size;
  function_map constructors;
  function_map methods;
  std::optional<std::function<void(void *)>> destructor;
  std::unordered_map<std::string, std::function<pointer(void *)>>
    fields;
  ffi_type libffi_type;
};
```

**Listing 4.3:** `runtime::type` structure

```
struct pointer {
  void *ptr;
  size_t size;
  size_t element_count;
  size_t element_size;
  type *typ;
  bool is_ptr = false;
};
```

**Listing 4.4:** `runtime::pointer` structure

types are inserted first, functions are inserted next, methods and constructors are attached to types afterwards, and libffi call interfaces are prepared after metadata population.

Handwritten standard entries cover the primitive and helper cases that should exist independently of any target library. The implementation registers primitive integer and character types, selected standard C++ types such as string, string view, and input string stream, and helper functions such as `strlen` and `free`. These entries let scripts use common values without requiring every target library to provide equivalent declarations.

## 4.3 Parser and Program Construction

CIRO script parsing is implemented using ANTLR4, which is a powerful parser generator that can handle complex grammars and generate code in multiple programming languages. The ANTLR grammar for CIRO scripts is defined in the `src/Ifl.g4` file, which specifies the syntax rules for the language, including statements, expressions, labels, and comments. CMake automatically invokes ANTLR to generate lexer and parser code as part of the build process. The grammar accepts initial pragmas, statement lines, expression lines, labels, and comments, and it is configured to treat keywords case-insensitively.

When parsing a script, the ANTLR-generated parser produces a parse tree that represents the Abstract Syntax Tree (AST) of the script; this is then lowered by a parser visitor into a `parser::program` object. As shown in Listing 4.5, the program stores the flat instruction vector, type aliases, function aliases, jump-label table, variable-name table, and fuzzer mutator parameters necessary for correct program execution. Lowering through a visitor keeps parse-tree details out of the interpreter and gives parser errors a natural place to attach source locations.

The mapping of variable names to numeric slots is performed during visiting, which allows the interpreter to use a simple vector of descriptors for variable storage. The visitor also tracks variable validity and rejects redeclaration while a variable is live. The `free` statement makes a name available for later declaration, and runtime checks ensure that invalid variables are not used during execution. These checks are implemented despite the fact that the interpreter is designed to be *unsafe*, because during development a use-after-free issue in a script caused hard-to-debug crashes, and the runtime checks provide a more user-friendly error message.

Two built-in variables are inserted at the beginning of every program. The `nullptr` variable contains a null pointer value, and the `empty_cb` variable contains a no-op callback pointer. These built-ins are useful for target APIs that accept null arguments or optional

```
class program {
public:
  std::vector<std::unique_ptr<ifl::interpreter::instruction>>
   instructions;
  std::unordered_map<std::string, ifl::runtime::type *>
   type_aliases;
  std::unordered_map<std::string, ifl::runtime::function *>
   function_aliases;
  std::unordered_map<std::string, size_t> jump_labels;
  std::vector<std::string> variable_names;

  size_t fuzzer_mutator_fixed_size;
  std::vector<size_t> fuzzer_mutator_counter_sizes;

  bool use_custom_mutator = false;
};
```

**Listing 4.5:** `parser::program` structure

callbacks.[1]

Alias resolution also occurs during visiting. A function alias is resolved against the function alias map, and a type alias is resolved against the type alias map. Both alias and direct references are stored as references to the corresponding metadata objects to prevent having to repeat lookup in the global maps.

Labels are resolved after instruction construction. The visitor associates a label with the next emitted instruction, and `program::resolve_jumps()` replaces textual jump targets with program-counter positions. Undefined labels become parser exceptions, which prevents execution from reaching an unresolved branch.

Literal parsing is implemented in parser utilities rather than left entirely to ANTLR tokenization. String parsing removes delimiters and accepts explicit escapes for newline, carriage return, tab, and backslash. Integer parsing uses `std::from_chars`; decimal tokens are first interpreted as signed 64-bit values and then, if they are out of range, as unsigned 64-bit values whose bits are stored in the expression object.

## 4.4 Interpreter State

Each execution receives a fresh `interpreter::state` object. As shown in Listing 4.6, the state stores the marker needed for the interpreter-aware sanitizer death callback as described in Section 3.8, immutable program reference, output stream, program counter, variable descriptors, variable validity flags, result stack, scratch allocations, input pointer, input size, and number of consumed input bytes. Fresh state per input prevents one fuzzer iteration from leaking variables or consumption offsets into the next iteration.

The state constructor initializes the built-in variables described in Section 4.3 and marks them valid. User variables start invalid and become valid only through allocation, consumption, construction, or assignment behavior that produces a concrete descriptor.

The state destructor performs cleanup for variables that remain valid at the end of exe-

[1] Due to the System V ABI used by Linux, we can pass a callback that has no arguments defined in place of callbacks with defined arguments, and the call still succeeds.

```
class state {
  uint8_t needle[needle_size];
  const parser::program &program;

  std::ostream &output;

  size_t pc;
  std::vector<ifl::runtime::pointer> variables;
  std::vector<uint8_t> variable_valid;
  std::stack<ifl::runtime::pointer *> stack;
  std::unordered_set<void *> scratch;

  const uint8_t *input;
  const size_t input_size;
  size_t consumed_input;
};
```

**Listing 4.6:** `interpreter::state` structure

```
class instruction {
  std::optional<parser::source_location> loc;

public:
  virtual void execute(state &) const = 0;
  virtual ~instruction() = default;
  void set_source_location(const parser::source_location &loc);
  const parser::source_location &source_location() const;
};
```

**Listing 4.7:** `interpreter::instruction` interface

cution. Typed variables call their registered destructor before memory is released. Non-fuzzer builds can emit warnings for variables that were not explicitly freed, while fuzzer builds suppress these warnings to avoid polluting the fuzzer output stream.

Scratch allocations support temporary expression results. When an expression needs to return a value but the caller has not provided a destination buffer, the expression allocates scratch memory and records it in the state. The main execution loop frees all scratch allocations after every root instruction and asserts that the result stack is empty, keeping temporary lifetime local to one statement.

## 4.5 Instruction Execution

All root instructions expose the common interface shown in Listing 4.7. Source locations are attached to root instructions so that parser and interpreter errors can report the script line associated with the failing operation. Nested expression instructions use the same execution interface but communicate through the result stack.

Allocation instructions create descriptors for native memory. Typed allocation uses `calloc` with the size stored in runtime type metadata. Dynamic array allocation first evaluates a size expression and then delegates to the same typed allocation helper. Pointer allocation reserves pointer-sized cells and marks the descriptor as a pointer-cell allocation.

String allocation copies a parsed string literal into a null-terminated buffer.

Input-consumption instructions translate fuzzer bytes into script variables. The basic consume instruction copies exactly the size of an already allocated descriptor and advances the consumed-input count. Another instruction allocates a buffer containing all remaining input and a variable storing its size. A separate stream instruction constructs an `std::istringstream` over the remaining input.

The `free` instruction releases one or more variables. A valid typed variable calls its destructor callback before memory is freed, then its descriptor is reset and its validity flag is cleared. Invalid variables are ignored, which makes cleanup blocks robust when earlier branches may have skipped an allocation.

Assignment uses the expression stack to avoid a separate lvalue and rvalue hierarchy. The destination expression first turns an empty stack descriptor into an addressable location. The source expression then writes its value into that destination. Array identifiers on the right-hand side are wrapped as addresses by the visitor so that array values decay to pointer values where appropriate.

Constructor instructions require a valid destination variable with runtime type metadata. The instruction looks up the constructor signature in the type's constructor map and dispatches it through the same method-call helper used for ordinary methods. The generated constructor wrapper performs placement new in the allocated destination buffer.

Early exits and aborts are implemented by a shared templated instruction. The instruction evaluates an expression, applies a selected test, and throws either a controlled exit exception or an abort exception when the test succeeds. The standalone interpreter treats controlled exits as successful termination and aborts as execution errors. The fuzzer maps aborts to a negative return value so that the current input is not retained in the corpus.

Jump instructions evaluate a condition and update the program counter when the condition holds. The execution loop increments the program counter after every instruction, so resolved jump targets account for that increment.

## 4.6 Expression Evaluation and Native Calls

Identifier expressions either return a descriptor view or copy bytes into an existing destination. An invalid variable raises an interpreter exception. Integer and string literals write into a caller-provided destination when one exists, or allocate scratch storage when the expression is evaluated as a value.

Address-of, array access, and field access produce descriptor views over native memory. Address-of writes the address of an evaluated descriptor into a pointer-sized cell. Array access computes an offset from the base pointer and element size. Field access evaluates the object expression, checks for runtime type metadata, and invokes a generated field accessor.

Integer operations are deliberately small in scope. Binary operations cover addition, subtraction, multiplication, division, and modulo over values of size 1, 2, 4, or 8 bytes, converted to 64-bit signed integers, which are then truncated to the destination size. Unary increment and decrement also operate on these sizes of integer buffers, but do so in place. This implementation exploits by design some *undefined behavior* in C++ to avoid having to implement a full integer type system. The interpreter does not check for overflow, underflow, or division by zero, and it does not distinguish between signed and unsigned

```
while (state.pc < program.instructions.size()) {
    program.instructions[state.pc]->execute(state);
    for (auto ptr : state.scratch) {
        free(ptr);
    }
    state.scratch.clear();
    assert(state.stack.size() == 0);

    state.pc++;
}
```

integers. The interpreter also does not support floating-point operations, which are left for future work.

Function-call expressions store a reference to a `runtime::function` and a vector of argument expressions. At execution time, the call checks arity, chooses a return destination, evaluates arguments into temporary descriptors, fills the libffi argument array, and invokes the registered function wrapper. Functions require a destination for the return value unless their return value is `void`.

Method calls extend the same machinery with an implicit object argument. The object expression must produce a descriptor with runtime type metadata. The method signature is looked up in the type's method map, the object address is stored as the first argument, and explicit arguments are prepared with the same pointer and value rules used for free functions.

## 4.7 Standalone Interpreter

`IflInterpreter` is the deterministic execution entry point. The command line accepts a script path, an optional input file, a run count, and a debug flag. Exit codes distinguish success, bad arguments, script loading failures, parser errors, and execution errors.

Interpreter startup installs the sanitizer death callback when available and then parses command-line arguments. The runner reads the script, reads the input file if provided, constructs the ANTLR input stream, lexer, token stream, and parser, and visits the parse tree to obtain a `parser::program`. Debug mode can print parser and execution information useful while developing scripts and investigating issues.

Program execution is a direct loop over the flat instruction vector. The loop creates a fresh state for each requested run, executes the instruction at the current program counter, frees scratch memory after the instruction, checks that the expression stack is empty, and then advances the program counter.

Exception handling distinguishes expected script control flow from unexpected execution failures. Controlled exits stop the current run successfully. Aborts and other interpreter exceptions are reported with the current program counter and source location. Parser exceptions are reported before execution begins.

## 4.8 Fuzzer Entry Point

`IflFuzzer` exposes the libFuzzer-compatible entry point described in Listing 3.7 so that CIRO scripts can run inside a standard coverage-guided fuzzing loop. The executable can

```
[fixed bytes]
[counter 0][dynamic field 0]
[counter 1][dynamic field 1]
...
```

**Listing 4.8:** Structured input layout as expected by libErator

be linked either with native libFuzzer support or with the LibAFL compatibility layer, depending on the selected build configuration.

Fuzzer initialization parses the script once. The initializer searches the argument list for the script file option, removes that option before handing control back to the fuzzing engine, parses the script, lowers it into a program, copies custom mutator settings from the program, and installs the sanitizer death callback. Missing or invalid scripts are treated as initialization failures rather than as fuzz inputs.

Each fuzz iteration creates a fresh state over the input bytes supplied by the engine and executes the same instruction loop used by the standalone interpreter. Script output is directed to a stream that discards data so that debug prints do not slow down or pollute fuzzing runs. Interpreter aborts reject the current input, while normal completion returns success to the fuzzer.

The fuzzer reuses the parsed program but never reuses execution state. Program reuse avoids parser overhead in the hot path, and state freshness preserves the semantics of a normal libFuzzer harness where each input should begin from a clean harness state unless the target library itself keeps global state.

## 4.9 Custom Mutator

Some automated harness generation tools like libErator [33] produce inputs with a structured layout that is not well served by a byte-level mutator. The CIRO fuzzer therefore supports a custom mutator that understands the structured layout expected by libErator-generated harnesses. The mutator is optional and gets enabled when the script contains the corresponding configuration pragmas `#fuzzer_mutator_fixed_size` and `#fuzzer_mutator_counter_sizes`. Listing 4.8 shows the structured layout expected on the input data by the custom mutator. The configuration parameters specify the size of the fixed-length prefix followed by a static count of dynamic length fields, each preceded by a counter that indicates its length.

The structured input layout consists of a fixed prefix followed by repeated counter and field pairs. Each counter is decoded little-endian into a `size_t`, and the following dynamic field is interpreted as having that length.

Mutation chooses between mutating the fixed prefix and mutating one dynamic field. Dynamic-field mutation uses the fuzzer's ordinary byte-level mutator on a temporary buffer, then rewrites the associated counter to match the new field length. Candidate seeds are rejected when they would be empty or larger than the maximum size requested by the fuzzer engine.

Crossover parses dynamic fields from both parents. When one parent is malformed, the implementation falls back to the valid parent. When both parents are valid, the mutator chooses the fixed prefix from one parent and combines dynamic fields around a randomly

selected cut point, rewriting counters as it builds the output.

## 4.10 Metadata Extraction

The extractor lives under `extractor` and is split across two cooperating tools. Most of the work is driven from Python through the libclang bindings, which take care of orchestration, declaration filtering, data modeling, and template rendering. A companion C++ tool built on LibTooling supplies the one piece Python cannot obtain on its own: the precise record layout information required for the libffi struct descriptions.

This division reflects the different strengths of the two Clang libraries. Both expose the AST for programmatically analyzing and transforming C and C++ source code, but at different levels. libclang is a stable C API that offers high-level access to the tree and, crucially, bindings for other languages, including the Python bindings we rely on here. LibTooling, in contrast, is a C++ library with a lower-level and more flexible interface that surfaces details libclang leaves out, such as the exact byte layout of records. Because CIRO implements record passing by value through libffi, and libffi needs that layout to build its struct descriptions, the extractor defers to LibTooling wherever this information is required.

Extraction is configured entirely from CMake; from there, extraction proceeds as a pipeline. It begins with header discovery, which recursively scans the selected include directories for the usual C and C++ header suffixes; when explicit headers are given, they filter this set while preserving the requested order where possible. The compile arguments used to parse those headers come from the compilation database when one is available, and are otherwise reconstructed from the include directories of the CMake target.

Rather than parsing each header in isolation, the extractor generates a temporary stub translation unit that includes all of the selected headers. Both the layout extractor and the Python libclang traversal parse this single stub, so the two tools see exactly the same set of declarations. The stub is deleted once extraction completes, unless the debugging configuration asks for it to be kept.

The C++ layout extractor walks this translation unit and emits JSON describing the fields and byte offsets of every record declared in a whitelisted header. Its traversal understands scalar fields, nested records, constant arrays, base classes, virtual-base information, dynamic classes with their vptr slots, and bitfields, recorded as the bytes they occupy; templated records, which have no single concrete layout, are skipped. In parallel, the Python AST walk collects the declarations that CIRO can actually model: free functions, constructors and destructors, public classes and structs, public and static methods, namespaces, and public non-bitfield fields. It filters these by source header and discards everything that is either not actually usable by API consumers such as deleted methods, non-public methods and fields or not supported by the interpreter itself, such as templates and any symbol named in the exclusions file.

The collected declarations populate a small data model built from Python dataclasses for functions, type methods, fields, constructors, and types. Functions are keyed by their name together with the canonical spelling of their arguments, so that overloads stay distinct, while types are tracked internally by Clang cursor hash and emitted under the fully qualified string keys that CIRO scripts use to refer to them. Each argument and return type is then mapped from its libclang type kind to the corresponding libffi symbol: pointers

and references collapse to a pointer type, scalar integers and floating-point types map to their libffi equivalents, enums resolve through their underlying integer type, and records refer back to the libffi struct description of their registered runtime type. Any type that cannot be mapped raises an error, so that unsupported declarations fail visibly during generation instead of silently producing broken bindings.

Finally, the data model is rendered into C++ registration code through templates. Generated type entries carry the object size, an optional destructor, field accessors, and the libffi struct layout; function entries carry the signature key, return and argument types, argument storage, and the raw function-pointer cast; and method entries wrap member function pointers into a form callable through libffi. Duplicate keys are caught when the generated maps are merged and reported either at generation time or, once the code is linked, at initialization time.

## 4.11 Validation and Support Infrastructure

The proof-of-concept library is part of the repository because it provides a stable target for features that are difficult to validate only against external libraries. The library contains overloaded functions, variadic functions, constructors, destructors, const and non-const methods, public fields, inheritance, a deliberately unsafe function for sanitizer validation, and a type with a deleted destructor.

Unit tests cover parser utilities and selected visitor behavior, including string parsing, invalid string escapes, integer parsing, and fuzzer mutator pragmas. These tests check small pieces of behavior in isolation.

Differential tests validate the central execution claim more directly. Test targets compare a raw native C++ harness, a C++ harness that calls generated CIRO metadata directly, and an interpreted CIRO script. Matching observable output across these paths gives evidence that generated metadata matches the target API, libffi calls match direct calls, and interpreted execution matches compiled harness behavior.

Example scripts exercise real target integrations such as cJSON, c-ares, LibTIFF, OpenCV, and OpenSSL 1.0.1f. The Heartbleed script demonstrates that a CIRO harness can reproduce a known vulnerability in a real library build. These scripts also serve as regression material for parser, extractor, linker, and runtime behavior.

Docker and bootstrap scripts provide reproducible build environments. The container configuration can build the LibAFL libFuzzer compatibility runtime, OpenSSL 1.0.1f with required instrumentation, the standalone interpreter, and the fuzzer image.

# 5 Evaluation

This chapter evaluates the effectiveness of CIRO when compared against traditional compiled fuzzing harnesses. The evaluation is performed by answering the following research questions:

- **RQ1 (Correctness):** Does CIRO faithfully reproduce a harness's crashes, without introducing false positives or false negatives?
- **RQ2 (Accuracy):** Does an interpreted harness reach the same code coverage in the target library as its compiled counterpart?
- **RQ3 (Performance):** What is the fuzzing throughput (executions/second) of interpreted harnesses relative to compiled ones, and how much overhead does the interpreter impose? Is the throughput competitive enough that build and storage savings are not erased by slower execution?
- **RQ4 (Build-time Savings):** How much build-time and storage savings does CIRO provide when compared to compiled harnesses?

## 5.1 Experimental Setup

The evaluation was performed on the following machines:

- `EM-I620E-NVME`: a bare-metal machine equipped with an AMD EPYC 8534P CPU providing 64 cores and 128 threads, 576 GB of RAM and two 3.84 TB NVMe SSDs running Ubuntu 24.04.4 LTS with Linux kernel 6.8.0-134-generic. To ensure consistent performance measurements, Hyperthreading and C-States were disabled on the BIOS settings, and the whole machine was dedicated to the evaluation. This machine was used mainly for the fuzzing campaigns.
- `padoverflow`: a bare-metal machine equipped with an AMD Ryzen Threadripper 7980X CPU providing 64 cores and 128 threads, 256 GB of RAM and two 2 TB NVMe SSDs running Ubuntu 24.04.4 LTS with Linux kernel 6.17.0-35-generic. This machine was used to run the build-time savings experiments.

The experiments themselves were run in Docker containers running an image based on Ubuntu 26.04 with the Clang 22.1.2 compiler toolchain and LibAFL 0.15.4-159-gdf8f6ff0.

## 5.2 Benchmark Selection and Methodology

The coverage and throughput experiments use five fuzzing harnesses drawn from four widely used C and C++ libraries, chosen to span different input domains and different amounts of work performed per execution:

- **cares-fuzz** exercises c-ares by parsing a raw Domain Name System (DNS) record with `ares_dns_parse` and then re-serializing it with `ares_dns_write`. It is a simplified version of `ares-test-fuzz.c` from the c-ares source tree;
- **cares-fuzzname** initializes a c-ares resolver channel and feeds the fuzzer input to `ares_set_servers_csv`, which parses a comma-separated list of name servers to use for resolving queries. This driver is a verbatim implementation of `ares-test-fuzzname.c` from the c-ares source tree;
- **cjson** parses the input as a JSON document with `cJSON_ParseWithLength` and serializes it back with `cJSON_Print`. This driver is a simplified version of `cjson_read_fuzzer.c` from the cJSON source tree;
- **libtiff** opens the input as an in-memory TIFF image using the C++ Input/Output interface for LibTIFF, reads its tags, and decodes it through `TIFFReadRGBAImage` and `TIFFReadScanline`. This driver is a verbatim implementation of `tiff_read_rgba_fuzzer.cc` from the LibTIFF source tree which is also used in OSS-Fuzz;
- **opencv** wraps the input in a `cv::Mat`, then in a `cv::InputArray`, and finally decodes it as an image through `cv::imdecode`. This driver is a verbatim implementation of `imdecode_fuzzer.cc` from the OSS-Fuzz project.

The cJSON harness performs comparatively little work per execution, the two c-ares harnesses and the OpenCV image decoder spend most of each execution inside the target library, and the LibTIFF driver combines substantial decoding work with the most complex harness-side control flow of the five; this spread is what lets us characterize how the interpreter overhead behaves as a function of the work done inside the target library relative to the work done by the harness itself.

For every harness we build two variants that differ only in how the harness itself is executed: a *native* variant, in which the harness is written in C or C++ and compiled into a dedicated binary (the baseline used in practice today), and an *interpreted* variant, in which the same harness is expressed as a CIRO script and executed by the interpreter. Both variants link against the same exact build of the target library, with the same sanitizer and coverage instrumentation, so that any observed difference is attributable to the execution mechanism and not to the library build. All campaigns use the LibAFL libFuzzer-compatible backend described in Section 3.7.

Each configuration was run for 24 hours and repeated 5 times, starting from the same initial corpus. Each individual run was pinned to a single CPU core and had access to 4 GB of RAM. Source-level coverage was sampled once per minute by using a dedicated native driver compiled with `-fprofile-instr-generate -fcoverage-mapping` together with `llvm-cov` to replay the generated corpus over the instrumented target library, and fuzzing throughput was taken from the backend's own execution counters. Unless stated otherwise, reported figures are the mean over the five repetitions, and coverage-over-time plots are shown with 95 % confidence intervals.

## 5.3 Correctness

**RQ1** asks whether CIRO faithfully reproduces a harness's crashes, without introducing false positives or false negatives. The central correctness claim behind CIRO is that executing a harness through the interpreter is behaviorally equivalent to compiling and running the same harness natively. We validate this claim in three complementary ways: through differential testing during development, through a real-world vulnerability reproduction, and through a systematic comparison of the results of the fuzzing campaigns.

**Differential testing.** As described in Section 4.11, each validation target is expressed in three execution styles that share the same intent: a raw native C++ harness, a C++ harness that calls the generated metadata directly, and an interpreted CIRO script. The three styles are executed on the same inputs and their observable outputs are compared for an exact match. Agreement across the three paths simultaneously validates the three components on which correctness depends: that the extracted metadata corresponds to the real API, that the libffi call boundary reproduces the native ABI, and that the interpreter implements the script semantics faithfully.

**Vulnerability reproduction.** To confirm that CIRO can in fact drive a production library through its real ABI and expose a genuine, security-relevant defect, we successfully reproduced Heartbleed [6] (CVE-2014-0160) against OpenSSL 1.0.1f by re-implementing the classic compiled harness as a CIRO script.

**Crash parity in the campaigns.** Across the 24-hour campaigns of Section 5.2, the native and interpreted variants of each harness reported the same set of crashing objectives, so the interpreter introduced neither spurious crashes (false positives) nor missed crashes (false negatives). In particular, we have not found any crashes during the campaigns except for multiple crashes caused by Out of Memory (OOM) conditions in both LibTIFF and OpenCV,[1] both of which were successfully found by the native and interpreted variants.

## 5.4 Coverage Parity

**RQ2** asks whether an interpreted harness reaches the same code coverage in the target library as its compiled counterpart. Because both variants link against the same instrumented library, coverage differences can only arise if the interpreter fails to reproduce some behavior of the native harness, or if it executes sufficiently more slowly that it explores less of the input space within the time budget.

Table 5.1 reports the branch coverage attained at the end of the 24-hour campaigns, averaged over the five repetitions. Among the metrics produced by `llvm-cov` we focus on branch coverage because it is the most informative: it measures how many of the target's conditional outcomes were actually taken, and is therefore more sensitive to unexercised behavior than function, line, or region coverage. Across all five harnesses the

[1] The LibTIFF crashes are specific to the C++ interface for I/O operations, when using the C API a memory usage limit can be optionally set to mitigate this class of bugs. The issue was reported to the project maintainers but we have not received a response. In case of OpenCV we are still investigating the underlying issue and we are planning to responsibly disclose it to the maintainers.

**Table 5.1:** Final branch coverage of the native and interpreted harness variants, as a percentage of the target library, reported as the mean over five 24-hour campaigns with 95 % confidence intervals.

| Harness | Native (%) | Interpreted (%) |
|---|---|---|
| cares-fuzz | $18.27 \pm 0.02$ | $18.21 \pm 0.07$ |
| cares-fuzzname | $18.97 \pm 0.06$ | $18.61 \pm 0.16$ |
| cjson | 41.32 | 41.04 |
| libtiff | $47.21 \pm 0.79$ | $46.71 \pm 0.61$ |
| opencv | $17.11 \pm 0.28$ | $16.79 \pm 0.08$ |

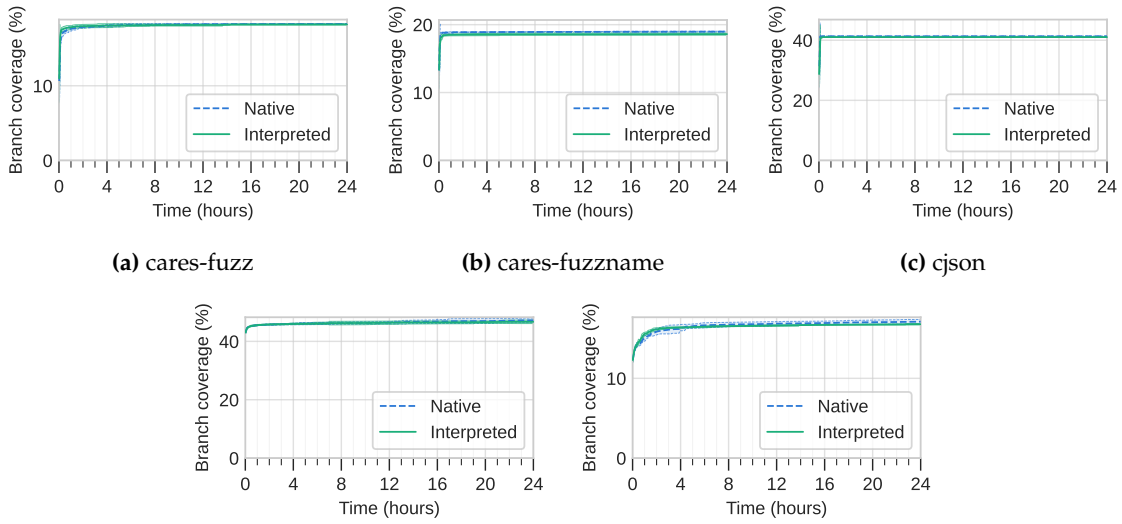
interpreted variant reaches branch coverage within half a percentage point of the native one; the largest gap, on LibTIFF, is half a point and the smallest, on cares-fuzz, only six hundredths of a point. On the cares-fuzz, libtiff, and opencv harnesses the confidence intervals of the two variants overlap, so those gaps fall within the run-to-run variation of the campaigns. cJSON is a different case: branch coverage is identical across all five repetitions of each variant and the confidence interval is zero, so the residual gap of 0.28 percentage points, while small, is exact and perfectly reproducible rather than a sampling artifact. Although it is not the largest gap in absolute terms, it is the only one that is statistically exact rather than subject to run-to-run variation. As discussed below, it reflects the smaller number of iterations the interpreted cJSON harness completes rather than any behavior it fails to reproduce.

The coverage-over-time curves in Figure 5.1 show that this parity is not merely an artifact of the final measurement: the interpreted harnesses have essentially the same growth trajectory as the native ones throughout the campaign and reach a closely comparable plateau. The residual gaps emerge early and then stay roughly constant, so they do not reflect a divergence that grows over time. Where the interpreter runs slower than the native harness the interpreted campaign completes fewer iterations within the fixed time budget and therefore settles marginally below the native plateau; the remaining differences are of the same order as the stochastic variation between fuzzing runs, which for a high-variance target such as LibTIFF is itself wider than any of the observed gaps. In no case does the difference indicate a behavior that the interpreter is unable to reproduce, consistent with the correctness results of Section 5.3.

## 5.5 Fuzzing Throughput

**RQ3** asks how the fuzzing throughput of interpreted harnesses compares to that of compiled ones, and therefore how much overhead the interpreter imposes. Table 5.2 reports the sustained throughput of each variant, computed as the total number of executions divided by the total wall-clock time of the campaign and averaged over the five repetitions.

This answers **RQ3**: the interpreter is not a bottleneck for the class of targets where fuzzing time is actually spent inside a non-trivial library, and even in the worst case observed the interpreter retains more than 40 % of native throughput. As Section 5.4 showed, this penalty does not translate into a meaningful loss of coverage over a realistic time budget, and, as the next section discusses, it is offset by substantial savings in build time and storage.




**(a)** cares-fuzz **(b)** cares-fuzzname **(c)** cjson

**(d)** libtiff **(e)** opencv

**Figure 5.1:** Branch coverage over time for the native and interpreted variant of each harness. Each curve is the mean of five 24-hour campaigns and the shaded band is the 95 % confidence interval.

**Table 5.2:** Sustained fuzzing throughput (executions per second), computed per run as total executions over total runtime and reported as the mean over five 24-hour campaigns with 95 % confidence intervals.

| Harness | Native (exec/s) | Interpreted (exec/s) | Interpreted / Native |
|---|---|---|---|
| cares-fuzz | 707 ± 25 | 857 ± 30 | 1.21× |
| cares-fuzzname | 4406 ± 161 | 4784 ± 68 | 1.09× |
| cjson | 23 427 ± 4009 | 14 613 ± 309 | 0.62× |
| libtiff | 6650 ± 960 | 2883 ± 780 | 0.43× |
| opencv | 2288 ± 95 | 2125 ± 299 | 0.93× |

**Table 5.3:** Storage footprint of each harness under the two execution models. The compiled model produces one statically linked, whole-archive binary per harness; the interpreted model reuses a single interpreter binary (`IflFuzzer`), built once per target library, together with a small per-harness CIRO script. The two c-ares harnesses share the same interpreter binary.

| Harness | Compiled binary | Interpreter | CIRO script |
|---|---|---|---|
| cares-fuzz | 323 MB | 330 MB | 839 B |
| cares-fuzzname | 323 MB | 330 MB | 523 B |
| cjson | 321 MB | 328 MB | 287 B |
| libtiff | 324 MB | 332 MB | 2359 B |
| opencv | 389 MB | 402 MB | 512 B |

## 5.6 Build-Time and Storage Savings

**RQ4** asks how much build-time and storage cost CIRO avoids relative to compiled harnesses. The saving stems directly from the architecture described in Section 3.2: the interpreter and the instrumented target library are built *once*, after which every new harness is a small script that requires no compilation or linking, whereas the conventional approach compiles and links a fresh native binary for every candidate harness.

**Storage.** Because each compiled harness statically links the instrumented target library, the resulting binaries are a few hundred megabytes in size, but they differ from the others only in a small amount of harness-specific code. The interpreted model builds this bulk *once*: as Table 5.3 shows, a single interpreter binary per target library, itself comparable in size to one compiled harness, is reused for every harness targeting that library, and each individual harness is expressed as a CIRO script of only a few hundred bytes to a couple of kilobytes. The per-harness storage cost therefore drops from a fresh multi-hundred-megabyte binary to a script five to six orders of magnitude smaller. The crossover is immediate: as soon as a second harness targets the same library the interpreted model already stores less overall, and a generator that emits thousands of harness variants stores thousands of kilobyte-scale scripts against a single shared interpreter, rather than thousands of near-duplicate multi-hundred-megabyte executables.

**Build time.** The interpreted approach replaces the per-harness compile-and-link step with the negligible cost of parsing a script at start-up. Table 5.4 reports, for each harness, the wall-clock time to compile and link the native harness, the one-time cost of building the interpreter for the target library, and the time the interpreter spends parsing the script at start-up. The two compile-time figures assume that the target library and all other dependencies are already built, so they measure only the construction of the executable; the interpreter figure additionally includes the generation of the metadata for the target library. Each value is the mean over five builds with its 95 % confidence interval, and the interpreter build time is identical for the two c-ares harnesses because they use the same executable.

Compiling a native harness takes between roughly 2 s and 3.6 s, and this cost is paid afresh for every new harness. Building the interpreter is a one-time cost per library: it is more expensive than a single native compilation, ranging from 3.1 s for cJSON to 14.5 s for OpenCV, because it also generates the metadata for the whole exposed API, so its magni-

**Table 5.4:** Build-time cost of each harness under the two execution models: the wall-clock time to compile and link the native harness, the one-time interpreter build for the target library (including metadata generation), and the interpreter's per-run script-parsing time. Compile times assume the target library and all other dependencies are prebuilt. Each figure is the mean over five builds with 95 % confidence intervals; the interpreter build time is shared by the two c-ares harnesses.

| Harness | Native compile (s) | Interpreter build (s) | Script parse (ms) |
|---|---|---|---|
| cares-fuzz | $2.00 \pm 0.07$ | $5.05 \pm 0.36$ | $2.60 \pm 0.20$ |
| cares-fuzzname | $1.95 \pm 0.07$ | $5.05 \pm 0.36$ | $2.61 \pm 0.20$ |
| cjson | $2.15 \pm 0.04$ | $3.14 \pm 0.09$ | $1.96 \pm 0.21$ |
| libtiff | $2.23 \pm 0.15$ | $6.50 \pm 0.22$ | $3.59 \pm 0.07$ |
| opencv | $3.59 \pm 0.17$ | $14.53 \pm 1.45$ | $2.22 \pm 0.14$ |

tude is directly correlated with the size of the API surface the library exposes. OpenCV is the most expensive in both columns, reflecting its heavy use of C++ features, which enlarges both the native harness and the metadata that must be generated and compiled into the interpreter. Once the interpreter exists, however, each additional harness costs only the script-parsing time that has been measured to be between roughly 2 ms and 3.6 ms, so around three orders of magnitude below a native compilation. The one-time interpreter build is therefore amortized almost immediately: it pays for itself after only a handful of harnesses per library, in particular two for cJSON, three for the c-ares and LibTIFF targets, and five for OpenCV, and beyond that point the total build cost of the interpreted approach grows only in milliseconds per harness while the compiled approach keeps growing in seconds.

# 6 Discussion

This chapter puts the experimental results of Chapter 5 into perspective, examines the limitations of the current approach, and outlines directions for future work.

## 6.1 Discussion of the Results

Taken together, the experiments answer the four research questions in CIRO's favor. Interpreted harnesses reproduce the behavior of compiled ones exactly, introducing neither false positives nor false negatives and reproducing a real vulnerability against a production library (**RQ1**). They reach the same branch coverage as compiled harnesses to within half a percentage point on every target (**RQ2**), sustain competitive throughput (**RQ3**), and collapse the per-harness build and storage cost by several orders of magnitude (**RQ4**).

The throughput results are best read through the lens of where fuzzing time is actually spent. The interpreter adds a fixed dispatch cost per script instruction, so its relative impact depends entirely on how much work the target library performs per execution. For the harnesses whose per-iteration cost is dominated by the native library call, namely both c-ares harnesses and the OpenCV image decoder, this cost is amortized to the point of being unmeasurable: interpreted throughput is statistically indistinguishable from native for OpenCV and even consistently higher for both c-ares harnesses. While we did not isolate a definitive cause for the latter, the most plausible explanation lies in incidental differences between the two executables: they embed the exact same build of the target library, but at different positions and surrounded by different code, and for executions dominated by library calls, such layout effects, for example on instruction-cache behavior, can shift throughput by several percent in either direction. This does not imply an intrinsic speed advantage of interpretation; it does, however, indicate that on these targets the interpreter's own dispatch cost is small enough to be hidden by second-order effects. The two harnesses that do show a visible penalty are slowed down for different reasons. The LibTIFF driver contains by far the most complex control-flow logic of the five, so a larger share of each iteration is spent executing interpreted harness instructions rather than native library code. cJSON, by contrast, spends very little time inside the native calls themselves: with executions this short, the per-execution cost is likely dominated by the setup of the interpreter state rather than by instruction dispatch. Even in these worst cases, the interpreter retains more than $40\,\%$ of native throughput. Since the libraries for which fuzzing is most valuable tend to fall in the first category (realistic parsers and decoders that perform substantial work per input), the interpreter can be considered not to pose a bottleneck for most applications.

The coverage results show that this throughput penalty, where present, does not translate into a corresponding loss of exploration. The interpreted variants follow the same coverage trajectory as the native ones and settle within half a percentage point of the native plateau on every target. Even for cJSON, whose residual gap is exact and reproducible across repetitions, the difference amounts to $0.28$ percentage points of branch coverage after $24$ hours and is explained by the smaller number of iterations completed within the time budget, not by any behavior the interpreter fails to exercise. Over a realistic fuzzing campaign, executing fewer iterations per second costs far less coverage than the raw throughput ratio would suggest, because coverage growth is strongly sublinear in the number of executions.

Finally, the savings measurements quantify the scalability argument that motivated CIRO. Building the interpreter for a library costs more than compiling a single native harness (between roughly $1.5$ and $4$ times as much, growing with the library's API surface), but this cost is paid once and is repaid after between two and five harnesses. From that point on, each additional harness costs milliseconds of script parsing and at most a few kilobytes of storage instead of seconds of compilation and hundreds of megabytes of near-duplicate binary. This is exactly the cost profile required by automatic harness generation, where a tool may emit thousands of candidate harnesses for the same library: the marginal cost of trying one more harness becomes negligible. The modest and bounded throughput cost is therefore a favorable trade for the substantial gains in scalability that motivated the approach.

## 6.2 Limitations

CIRO intentionally avoids becoming a full programming language. The language is designed to express harness behavior, not arbitrary application logic: features such as structured loops, rich type checking, automatic ownership analysis, and high-level containers are deliberately missing. This keeps the interpreter small, fast, and easy to reason about, but it also shifts responsibility onto the script author or the harness generator, which must manage low-level details such as memory ownership and object lifetimes explicitly.

For the same reason, CIRO does not claim complete C++ feature coverage. The current design supports the practical subset of declarations needed by the experiments (free functions, methods, constructors, destructors, fields, and records), while templates and virtual method calls are not supported, copy and move constructors and some function-pointer signatures are skipped or only partially handled, and variadic functions must be instantiated explicitly by scripts. An exclusions file allows the extractor to ignore declarations that use unsupported features, so that no broken metadata entries are generated, but the corresponding parts of a library's API remain out of reach for interpreted harnesses.

Generated scripts are also not memory-safe by construction. Native fuzzing often requires raw pointers, pointer-to-pointer arguments, manual construction, and manual destruction, and CIRO exposes these operations directly. The interpreter provides enough bookkeeping to execute them consistently and to report many mistakes, but it deliberately leaves room for scripts to exercise unsafe target behavior under sanitizer instrumentation: this is, after all, the purpose of a fuzzing harness.

A number of implementation constraints should also be kept in mind when interpreting the evaluation results or extending the system. Descriptors containing pointers do not

currently preserve the runtime type of the field, which limits the usefulness of pointers; non-void function calls require a destination even when the result is unused; and integer operations are always performed in 64-bit signed arithmetic and truncated to the destination size. Moreover, the sanitizer state recovery mechanism depends on stack scanning, and generated metadata relies on whole-archive linking so that its registration constructors are not removed by the linker.

Finally, extending CIRO carries a coordination cost: a new language feature touches the grammar, the visitor, the interpreter, the build configuration, and the tests, while onboarding a new target library requires a build target, exposed include directories, static linkage, instrumentation configuration, and possibly exclusions and differential tests. The guiding principle that keeps this cost contained is that all target-specific behavior lives in generated metadata, scripts, and build configuration, while the interpreter core remains reusable. Within these boundaries, the design deliberately prioritizes rapid harness execution, native sanitizer behavior, and observability over completeness.

## 6.3 Future Work

This thesis has shown that CIRO is a viable approach to fuzzing C and C++ libraries without the need to compile a harness. However, there are several areas where CIRO can be improved and extended; in the following, we discuss some potential directions for future work.

**Just-In-Time Compilation.** A way to close the performance gap observed for the cheapest targets in Section 5.5 could be to implement a Just-In-Time (JIT) compiler for the harnesses: this would improve execution speed while maintaining the benefits introduced by interpreting the harness, such as reduced build time and storage requirements. Possible approaches include using an existing JIT compilation framework such as LLVM ORCv2 [18], which would require lifting the harness to LLVM IR, or by directly emitting machine code using a library such as Xbyak [25].

**Coverage Reporting.** The current approach relies only on the coverage information gathered on the target library, but it would be useful to also report coverage information for the interpreted harness itself, especially when the harness has a complex control flow. This would enable a more detailed understanding of the harness's behavior and could help identify potential issues or areas for improvement.

**Integration with Harness Generation Tools.** An obvious next step would be to integrate CIRO with existing harness generation tools, such as libErator [33], for example by adding a server execution mode to the interpreter that receives new harnesses and executes them on the fly, checking whether they actually increase coverage on the target library and providing feedback to the harness generation tool. This would result in a more seamless and efficient workflow for fuzzing C and C++ libraries.

**Support for More C++ Features.** In the current implementation, CIRO does not support certain C++ features, such as virtual function calls and template instantiations. Adding support for these features would widen the range of C++ libraries and applications that

CIRO can fuzz. Virtual function call support would require extracting the correct vtable layout from each type and then using the vtable pointer stored in each object to resolve the function to call. Template instantiation cannot be fully implemented in the interpreter, since doing so would essentially require implementing a C++ compiler; a possible approach could be to specify a set of templates to pre-compile and extract metadata from, allowing the interpreter to instantiate them at runtime.

# 7 Conclusions

Fuzzing is one of the most effective techniques for finding security vulnerabilities in C and C++ libraries, and automatic harness generation promises to extend its reach by producing large numbers of candidate harnesses with little human effort. The conventional workflow, however, compiles and links every candidate harness into a dedicated binary that statically embeds the instrumented target library, so each harness costs seconds of build time and hundreds of megabytes of disk space. At the scale at which generation tools operate, the compilation pipeline itself becomes the bottleneck, consuming resources that could otherwise be spent on fuzzing.

This thesis presented CIRO, an approach that removes the per-harness compilation step entirely. CIRO consists of a low-level IR that models C and C++ APIs, including records, fields, methods, constructors, and destructors, together with an interpreter that executes harnesses expressed in this IR directly against the target library. Metadata describing the library's callable declarations is extracted automatically with Clang tooling, native calls are dispatched through libffi so that the real ABI is exercised, and the interpreter integrates with the libFuzzer and LibAFL fuzzing backends while preserving native sanitizer behavior. A harness thus becomes a script of at most a few kilobytes that is parsed in milliseconds, while the interpreter and the instrumented library are built once per target.

The experimental evaluation, conducted on five harnesses drawn from four widely used libraries with 24-hour campaigns repeated five times, supports the central claim that interpretation is a faithful and practical substitute for compilation. Interpreted harnesses are behaviorally equivalent to their compiled counterparts: they report the same crashes, introduce neither false positives nor false negatives, and reproduce a real vulnerability, Heartbleed, against a production build of OpenSSL. They reach branch coverage within half a percentage point of the native variants on every target. Their throughput is at parity with, or even above, native throughput on targets whose execution time is dominated by the library, and remains above 40 % of native in the worst case observed. In exchange, the per-harness cost collapses: milliseconds of script parsing instead of seconds of compilation, and at most a few kilobytes of storage instead of a multi-hundred-megabyte binary, with the one-time interpreter build amortized after a handful of harnesses per library.

These results indicate that interpreted harness execution is a viable foundation for fuzzing at the scale that automatic harness generation enables: the cost of trying one additional harness drops to almost nothing, precisely the property that generation tools require. The current implementation supports a practical subset of C++, and the remaining gaps point to natural extensions, discussed in Section 6.3, such as JIT compilation of hot harnesses, richer coverage reporting, direct integration with harness

generation tools, and support for virtual calls and pre-compiled template instantiations. In automated harness generation workflows, decoupling harness authoring from harness compilation, as CIRO does, is a step toward making large-scale, fully automated fuzzing of C and C++ libraries routine.

# A CIRO Language Reference

This appendix is a reference for the CIRO harness language, the small DSL used to describe library fuzzing harnesses that the interpreter executes against a target library. The language is named *IFL* inside the implementation: its grammar lives in `src/Ifl.g4`, scripts use the `.ifl` extension, and the generated parser uses the IFL namespace. This document uses the public name *CIRO* for the language and reserves *IFL* for the concrete artifacts of the implementation.

The reference is normative with respect to the grammar in `src/Ifl.g4` and the behavior implemented by the parser visitor (`src/parser/visitor.cpp`) and the interpreter (`src/interpreter/`). It catalogs the lexical structure, the overall program structure, pragmas, the statement and expression forms, the operators, and the type system, giving for each construct its syntax, its runtime semantics, and a short example drawn from the example scripts shipped with the implementation. The motivation for these design choices is discussed in Chapter 3, and the parser and interpreter internals, including the runtime value and memory model, are described in Chapter 4; this appendix does not repeat that material and cross-references it where the runtime behavior depends on it.

Throughout, the meta-syntax `<x>` denotes a placeholder, `[x]` an optional element, and `(x)*` a possibly-empty repetition. Terminal keywords and punctuation are written verbatim.

## Grammar Summary

Listing A.1 gives a condensed IR grammar in EBNF form. It is a readability-oriented rendering of `src/Ifl.g4`; the ANTLR source remains the authoritative definition.

## Lexical Structure

**Case insensitivity.** The lexer is configured with `caseInsensitive=true`, so keywords match regardless of case: `alloc`, `ALLOC`, and `Alloc` are equivalent. The same option makes the identifier and type-name character classes, which are spelled with lower-case letters only, accept upper-case letters as well. Identifiers and type names themselves, however, remain case-sensitive: `data` and `Data` name distinct variables.

**Identifiers.** An identifier (`ID`) matches `[a-z_][a-z0-9_<>]*`. Identifiers name script variables, labels, and aliases. The angle brackets are permitted so that template-like names

```
program     ::= pragma* line+
pragma      ::= '#variadic' fn-string type-list-string
              | '#fuzzer_mutator_fixed_size' int
              | '#fuzzer_mutator_counter_sizes' int (',' int)*
line        ::= [label] (statement | expression) [comment] | comment
label       ::= id ':'

statement   ::= 'alloc' id type-ref [int | expr]  // single | array
              | 'alloc_str' id string
              | 'alloc_ptr' id [int | expr]       // single cell | array
              | 'alloc_consume' id id             // buffer var, size var
              | 'consume_stream' id
              | 'consume' expr
              | 'free' id (',' id)*
              | 'construct' id string '(' [args] ')'
              | 'fn_alias' id fn-string
              | 'type_alias' id type
              | 'print' string
              | 'hexdump' expr [expr]           // value [, size]
              | expr '=' expr                   // assignment
              | 'jump'  test expr id            // condition, target label
              | 'exit'  test expr [string]
              | 'abort' test expr [string]

expression  ::= int | string | id
              | fn-ref '(' [args] ')'             // function call
              | expr '.' string '(' [args] ')'    // method call
              | expr '.' id                       // field access
              | '&' expr                          // address-of
              | expr '[' expr ']'                 // array access
              | expr ('++' | '--')                // in-place unary
              | expr ('+'|'-'|'*'|'/'|'%') expr   // integer binary
              | 'sizeof' '(' (type | expr) ')'
args        ::= expr (',' expr)*

test        ::= 'zero'|'z'   | 'nonzero'|'nz' | 'less'|'l'
              | 'less_equal'|'le' | 'greater'|'g' | 'greater_equal'|'ge'
type-ref    ::= type | id
fn-ref      ::= fn-string | id
type        ::= '::' [a-z0-9_:<>]+
id          ::= [a-z_][a-z0-9_<>]*
int         ::= [0-9]+
string      ::= '"' ... '"'
comment     ::= '//' ...
```

**Listing A.1:** Condensed CIRO grammar

can be used as alias identifiers.

**Type names.** A type token (`TYPE`) starts with `::` and continues with `[a-z0-9_:<>]+`, for example `::int`, `::uint8_t`, `::size_t`, or `::cv::Mat`. The leading `::` makes a type name a fully qualified key into the runtime type metadata and removes ambiguity between namespaces and libraries.

**Integer literals.** An integer literal (`INTEGER`) is a sequence of decimal digits `[0-9]+`; there is no hexadecimal, octal, sign, or separator syntax. A literal is first interpreted as a signed 64-bit value and, if it does not fit, as an unsigned 64-bit value whose bit pattern is stored in the expression. A literal that fits in neither is a parse error.

**String literals.** A string literal (`STRING`) is delimited by double quotes, `"..."`. The recognized escape sequences are exactly `\n`, `\r`, `\t`, and `\\`; any other escape sequence, or a trailing backslash, is reported as an invalid string literal. String literals serve two distinct roles: ordinary text values (for `print`, `alloc_str`, and as expression operands) and *signature strings* (below).

**Signature strings.** Functions, methods, and constructors are identified by full C or C++ signature strings rather than by bare names. A free function is named by its complete signature, for example `"::ares_init(ares_channeldata **)"`; a method is named by its member signature including qualifiers, for example `"add(int) const"`; and a constructor argument list is given as a bare type list such as `"int,int"`. Encoding the full signature lets the runtime distinguish overloaded and namespaced declarations without performing C++ name lookup inside the interpreter.

**Comments and layout.** A comment starts with `//` and runs to the end of the line. Horizontal whitespace (spaces and tabs) is ignored. Line breaks are significant: each statement, expression, or label occupies its own logical line.

## Program Structure

A program begins with an optional block of pragmas, followed by one or more lines. A line is either a comment, or an optional label followed by a single statement or a single bare expression, optionally trailed by a comment. A bare expression line is typically a function or method call invoked for its side effects.

**Labels.** A label is an identifier followed by a colon, for example `loop_start:`. A label binds to the next emitted instruction and may appear on its own line or immediately before a statement on the same line. Labels are the targets of `jump` and are resolved to instruction positions after the whole program has been parsed; a jump to an undefined label is a parse error.

**Built-in variables.** Two variables are declared automatically at the start of every program and are always valid: `nullptr`, which holds a null pointer value, and `empty_cb`,

```
#variadic "::fmt(const char *, ...)" "int, void*"
alloc_ptr ptr_var
"::fmt(const char *, int, void*)"("%d %f", 42, ptr_var)
```

**Listing A.2:** Instantiating and calling a variadic function

which holds a no-argument callback pointer suitable for target APIs that accept optional callbacks. These names are reserved and should not be redeclared.

## Pragmas

Pragmas configure features that must be known before execution begins and must appear before any statement or expression line.

**`#variadic`.**

*`#variadic "<variadic-signature>" "<type>(, <type>)*"`*

Instantiates a registered variadic function for a concrete set of trailing argument types. The first string is the signature of a variadic function whose parameter list ends with `...`; the second string is a comma-separated list of the concrete types that replace `...`. The pragma registers a new ordinary callable whose signature is the variadic signature with `...` textually replaced by the supplied types, which can then be called like any other function. This is required because the underlying FFI layer needs the concrete variable arguments before it can prepare a call interface. Listing A.2 instantiates a `printf`-style variadic function and then calls the instantiated signature.

**`#fuzzer_mutator_fixed_size` and `#fuzzer_mutator_counter_sizes`.**

*`#fuzzer_mutator_fixed_size <int>`*
*`#fuzzer_mutator_counter_sizes <int>(, <int>)*`*

These pragmas enable and configure the structured custom mutator. The first declares the size in bytes of a fixed input prefix; the second declares the byte widths of the length counters that precede each dynamic field. Declaring either pragma switches the fuzzer to the custom mutator, which preserves the structured layout while mutating dynamic fields and rewriting their counters. The mutator and its input layout are described in Section 3.7 and in the implementation chapter.

## Names, Types, and Aliases

A *type reference* is either a type token (`::Name`) or an identifier previously bound by `type_alias`. A *function reference* is either a signature string or an identifier previously bound by `fn_alias`. Aliases make handwritten scripts more readable while keeping runtime lookup unambiguous: they are resolved at parse time against the runtime metadata maps, and an alias that names an unknown type or function is a parse error.

```
fn_alias decode "img::decode(const Buffer &, int)"
type_alias image ::img::Image

alloc result image
result = decode(raw, 1)
```

**Listing A.3:** Type and function aliases

**Table A.1:** CIRO statement forms.

| Syntax | Purpose |
|---|---|
| `alloc <id> <type-ref>` | Allocate one typed object |
| `alloc <id> <type-ref> <int>` | Allocate a typed array, literal count |
| `alloc <id> <type-ref> <expr>` | Allocate a typed array, dynamic count |
| `alloc_str <id> "<str>"` | Allocate a null-terminated string |
| `alloc_ptr <id>` | Allocate one pointer cell |
| `alloc_ptr <id> <int> \| <expr>` | Allocate an array of pointer cells |
| `consume <expr>` | Fill a buffer from fuzzer input |
| `alloc_consume <id> <id>` | Allocate from all remaining input |
| `consume_stream <id>` | Build an input stream over remaining input |
| `free <id>(, <id>)*` | Destroy and release variables |
| `construct <id> "<types>" (<args>)` | Construct an object in place |
| `<expr> = <expr>` | Assignment |
| `print "<str>"` | Print a literal string |
| `hexdump <expr> [<expr>]` | Hex-dump a value |
| `jump <test> <expr> <label>` | Conditional jump |
| `exit <test> <expr> ["msg"]` | Conditional successful exit |
| `abort <test> <expr> ["msg"]` | Conditional abort |
| `fn_alias` / `type_alias` | Declare an alias (Section above) |

**`type_alias <id> <::Type>`** Binds `<id>` to the registered type `<::Type>`. The alias can subsequently be used anywhere a type reference is expected (for example as the type in `alloc`).

**`fn_alias <id> "<signature>"`** Binds `<id>` to the registered function named by `<signature>`. The alias can subsequently be used as the callee of a function call.

Listing A.3 declares one alias of each kind and then uses both: the type alias names the allocation type and the function alias names the callee.

## Statements

Table A.1 summarizes the statement forms; the subsections that follow describe their semantics.

### Allocation

**`alloc`.** The `alloc` statement declares a variable and allocates native memory for it. With a type reference alone it allocates a single object of that type; with a trailing integer or expression it allocates an array of that many elements. The allocation is zero-initialized

```
alloc i ::int             // a single int
alloc a ::uint8_t 16      // array, literal count
alloc buf ::uint8_t size // dynamic count
```

**Listing A.4:** Typed and array allocation

```
alloc_ptr channel
"::net_open(net_channel **)"(&channel)
```

**Listing A.5:** Pointer cells

and sized from the type's runtime metadata. Allocating a non-pointer object means the variable owns the underlying storage. The three forms are shown in Listing A.4.

**`alloc_str`.** `alloc_str <id> "<str>"` allocates a null-terminated buffer initialized with the (escape-processed) contents of the string literal. It is the convenient way to obtain a C string argument.

**`alloc_ptr`.** `alloc_ptr <id>` reserves a single pointer-sized cell; the variable is marked as a pointer cell so that the runtime can later distinguish passing a pointer value from passing the address of a pointer cell, which is needed for pointer-to-pointer APIs. With a trailing integer or expression it reserves an array of pointer cells. A freshly allocated pointer cell is a common destination for the return value of an allocating target function, as in Listing A.5, where the cell receives an out-parameter.

### Input Consumption

CIRO never infers how fuzzer bytes are split into arguments; a script states this explicitly.

**`consume`.** `consume <expr>` copies exactly as many bytes as the size of the expression's buffer from the current position in the fuzzer input into that buffer, and advances the consumed-input cursor. Requesting more bytes than remain is a runtime error.

**`alloc_consume`.** `alloc_consume <buf-id> <size-id>` declares two variables: a buffer holding all of the remaining fuzzer input, and a `size`-typed variable holding that buffer's length. It is the usual entry point for harnesses that treat the whole input as one blob. Listing A.6 takes the whole input as a sized buffer and passes it to a parser.

**`consume_stream`.** `consume_stream <id>` constructs a `std::istringstream` over the remaining fuzzer input and binds it to the variable, for APIs that read from a C++ input stream.

```
alloc_consume data size
alloc_ptr doc
doc = "::json_parse(const char *, size_t)"(data, size)
```

**Listing A.6:** Consuming the whole input

```
alloc adder ::math::Adder
construct adder "int,int" (10, 20)
// ret declaration omitted for brevity
ret = adder."add(int) const"(2)
```

**Listing A.7:** Construction and a const method call

### Lifetime: `free`

`free <id>(, <id>)*` releases one or more variables. For a valid typed variable the registered destructor runs before the underlying memory is freed; the variable's slot is then reset and marked invalid, which also makes the name available for re-declaration by a later `alloc`. Freeing an invalid (for example never-allocated, or already-freed) variable is silently ignored, which keeps cleanup paths robust when earlier branches may have skipped an allocation.

### Construction: `construct`

`construct <id> "<arg-types>" (<args>)` constructs a C++ object in place inside an already-allocated typed variable. The argument-type string selects the constructor overload from the type's constructor metadata, and the parenthesized expressions supply the arguments. The generated constructor wrapper performs a placement-new into the variable's storage. Listing A.7 allocates an object, constructs it with a two-argument constructor, and then calls a `const` method on it.

### Assignment

`<dest-expr> = <src-expr>` evaluates the destination to an addressable location and then writes the source value into it. If the right-hand side is an identifier that names an array variable, the array decays to a pointer to its first element before being stored, mirroring C array-to-pointer decay. Assignment is used both to set scalar values and to capture function return values.

### Diagnostics: `print` and `hexdump`

`print "<str>"` writes the (escape-processed) string literal to the interpreter's output stream. `hexdump <expr>` writes a hexadecimal dump of the value's buffer, and `hexdump <expr> <size-expr>` dumps an explicit number of bytes, which is useful for pointers and untyped regions whose intrinsic size is not the region of interest. In fuzzer builds the output stream is discarded, so these statements are diagnostics for the standalone interpreter and do not affect fuzzing behavior.

### Control Flow: `jump`, `exit`, `abort`

CIRO control flow is deliberately low level: a conditional jump together with labels is sufficient to express loops, bounds checks, and cleanup paths over the flat instruction sequence. There are no structured `if` or `while` constructs.

All three statements share the same condition tests (see the *Test Conditions* section below) and evaluate an expression to obtain the buffer the test inspects.

```
alloc i ::int
i = 3
i_loop_start:
print "i is: \n"
hexdump i
i--
jump nz i i_loop_start  // loop while i != 0
free i
```

**Listing A.8:** A counted loop with a jump

**Table A.2:** Condition tests for `jump`, `exit`, and `abort`.

| Keyword | Alias | Holds when |
|---|---|---|
| `zero` | `z` | every byte of the operand is zero |
| `nonzero` | `nz` | any byte of the operand is non-zero |
| `less` | `l` | operand (signed 64-bit) `< 0` |
| `less_equal` | `le` | operand (signed 64-bit) `<= 0` |
| `greater` | `g` | operand (signed 64-bit) `> 0` |
| `greater_equal` | `ge` | operand (signed 64-bit) `>= 0` |

**`jump <test> <cond-expr> <label>`** Transfers control to `<label>` when the test holds for the condition expression; otherwise falls through to the next instruction.

**`exit <test> <expr> ["msg"]`** Stops the current run when the test holds, optionally printing the message first. An exit is treated as *successful* termination by the standalone interpreter.

**`abort <test> <expr> ["msg"]`** Stops the current run with an error when the test holds, optionally printing the message first. The standalone interpreter treats an abort as an execution error, and the fuzzer rejects the current input so that it is not retained in the corpus.

Listing A.8 combines a label, an in-place decrement, and a conditional jump to implement a counted loop.

## Test Conditions

The condition keywords used by `jump`, `exit`, and `abort` are listed in Table A.2. The `zero` and `nonzero` tests examine the whole buffer byte-by-byte and therefore apply to operands of any size. The ordering tests read the operand as a signed 64-bit integer and compare it against zero; the unsigned forms of `>` and `>=` are intentionally omitted because they are redundant (an unsigned value is `> 0` exactly when it is non-zero, and `>= 0` always).

## Expressions

Expressions produce values or addressable locations. Table A.3 summarizes the forms; the operators are listed in precedence-relevant groups as they appear in the grammar.

**Table A.3:** CIRO expression forms.

| Form | Meaning |
|---|---|
| `<int>` | Integer literal |
| `"<str>"` | String literal |
| `<id>` | Variable reference |
| `<fn-ref>(<args>)` | Function call |
| `<expr>."<sig>"(<args>)` | Method call |
| `<expr>.<field>` | Field access |
| `&<expr>` | Address-of |
| `<expr>[<expr>]` | Array element |
| `<expr>++ \| <expr>--` | In-place increment / decrement |
| `<expr> + - * / % <expr>` | Integer binary operation |
| `sizeof(<type>)` | Type size (compile time) |
| `sizeof(<expr>)` | Buffer size (run time) |

```
alloc i ::int
alloc_ptr pi
alloc_ptr ppi
i = 5
pi = &i     // pi points at i
ppi = &pi   // ppi points at pi
```

**Listing A.9:** Address-of and pointer-to-pointer

**Literals and identifiers.** An integer literal evaluates to its 64-bit value; a string literal evaluates to a text value; an identifier evaluates to a view of the named variable's storage. A reference to an invalid (unallocated or freed) variable is a runtime error.

**Function calls.** `<fn-ref>(<args>)` calls a free function identified by a signature string or a function alias. Arguments are evaluated left to right, and the call is dispatched through the runtime FFI metadata. A function with a non-void return type must be called in a context that supplies a destination (typically the right-hand side of an assignment); a call to a function with a non-void return whose result is unused still requires a destination.

**Method calls and field access.** `<expr>."<sig>"(<args>)` calls a method on an object, passing the object's address as an implicit first argument; the method signature selects the overload and includes qualifiers such as `const`. `<expr>.<field>` accesses a public field of an object through a generated field accessor. Both require the object expression to carry runtime type metadata.

**Address-of and array access.** `&<expr>` writes the address of the evaluated operand into a pointer-sized cell and yields that cell, which is how pointer and pointer-to-pointer arguments are formed. `<expr>[<expr>]` computes an element view from a base pointer using the element size and the index. Listing A.9 builds a pointer and then a pointer-to-pointer with repeated address-of.

```
alloc a ::uint32_t 3
alloc s ::size_t
s = sizeof(a)            // 12: run-time buffer size
s = sizeof(::uint32_t)   // 4: compile-time type size
```

**Listing A.10:** Compile-time and run-time sizeof

**Integer operators.** The binary operators `+`, `-`, `*`, `/`, and `%`, and the in-place unary operators `++` and `--`, operate on integers. Each operand is read as a signed 64-bit integer, the operation is performed in signed 64-bit arithmetic, and the result is truncated to the destination's size. There are no overflow checks, and the implementation assumes a two's-complement, little-endian target. Unary `++` and `--` modify a 1-, 2-, 4-, or 8-byte integer buffer in place.

**`sizeof`.** `sizeof(<type>)` yields the registered size of a type and is folded to an integer literal at parse time. `sizeof(<expr>)` yields, at run time, the byte size of the expression's buffer, which for an array is the total size of all elements. Listing A.10 contrasts the two forms.

## Built-in Environment

Independently of any target library, the runtime registers a set of primitive and standard types and a few helper functions, so that common values are available without per-library declarations.

**Primitive types** `::char`, `::uchar`, `::uint8_t`, `::uint16_t`, `::uint32_t`, `::uint64_t`, `::int8_t`, `::int16_t`, `::int32_t`, `::int64_t`, `::int`, `::long`, `::unsigned`, `::size_t`, and `::socklen_t`.

**Standard library types** `::std::basic_string<char>`, `::std::basic_string_view<char>`, and `::std::basic_istringstream<char>`, with constructors and selected methods registered.

**Helper functions** `"strlen"` and `"free"`, callable by those bare names, for measuring and releasing C strings and buffers produced by the target.

**Built-in variables** `nullptr` and `empty_cb`, described in the program-structure section.

## Worked Example

Listing A.11 is a complete harness for cJSON that ties the constructs together. It consumes the whole input as a buffer with a length, parses it, exits cleanly when parsing fails, otherwise serializes and frees the result, and finally releases every owned variable.

```
alloc_consume Data Size // whole input + its length
alloc_ptr json

json = "::cJSON_ParseWithLength(const char *,unsigned long)"(Data, Size)

exit zero json // exit cleanly if parsing failed

alloc_ptr json_print
json_print = "::cJSON_Print(const cJSON *)"(json)
"free"(json_print) // release the printed C string

"::cJSON_Delete(cJSON *)"(json) // library-specific teardown

free Data, Size, json, json_print // release owned variables
```

**Listing A.11:** A complete cJSON harness

# Acronyms

**ABI** Application Binary Interface. 2, 6, 11, 15, 23, 33, 43

**API** Application Programming Interface. 1, 2, 5, 6, 9, 11, 14, 15, 21, 22, 28, 29, 33, 36, 37, 40, 43, 48, 50

**ASan** AddressSanitizer. 8, 16, 18

**AST** Abstract Syntax Tree. 22, 28

**DNS** Domain Name System. 32

**DSL** Domain Specific Language. 6, 12, 45

**FFI** Foreign Function Interface. 12, 48, 53

**IR** Intermediate Representation. 1, 2, 11, 41, 43, 45

**JIT** Just-In-Time. 41, 43

**MSan** MemorySanitizer. 8

**OOM** Out of Memory. 33

**UBSan** UndefinedBehaviorSanitizer. 8

**VM** Virtual Machine. 2

# Bibliography


[1] Domagoj Babić et al. "FUDGE: fuzz driver generation at scale". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 975–985. ISBN: 9781450355728. DOI: 10.1145/3338906.3340456. URL: https://doi.org/10.1145/3338906.3340456.

[2] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-based Greybox Fuzzing as Markov Chain". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1032–1043. DOI: 10.1145/2976749.2978428.

[3] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[4] *Callgrind: a call-graph generating cache and branch prediction profiler*. [Online; accessed 17-June-2026]. 2026. URL: https://valgrind.org/docs/manual/cl-manual.html.

[5] Peng Chen et al. "Hopper: Interpretative Fuzzing for Libraries". In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. CCS '23. Copenhagen, Denmark: Association for Computing Machinery, 2023, pp. 1600–1614. ISBN: 9798400700507. DOI: 10.1145/3576915.3616610. URL: https://doi.org/10.1145/3576915.3616610.

[6] *CVE-2014-0160: Heartbleed*. [Online; accessed 17-June-2026]. 2014. URL: https://nvd.nist.gov/vuln/detail/cve-2014-0160.

[7] Andrea Fioraldi et al. "AFL++: combining incremental steps of fuzzing research". In: *Proceedings of the 14th USENIX Conference on Offensive Technologies*. WOOT'20. USA: USENIX Association, 2020.

[8] Andrea Fioraldi et al. "LibAFL: A Framework to Build Modular and Reusable Fuzzers". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 1051–1065. ISBN: 9781450394505. DOI: 10.1145/3548606.3560602. URL: https://doi.org/10.1145/3548606.3560602.

[9] Dave Gamble and contributors. *cJSON: Ultralightweight JSON parser in ANSI C*. [Online; accessed 21-June-2026]. 2026. URL: https://github.com/davegamble/cjson.

[10] Google. *RE2: A fast, safe, thread-friendly regular expression library*. [Online; accessed 21-June-2026]. 2026. URL: https://github.com/google/re2.

[11] Harrison Green and Thanassis Avgerinos. "GraphFuzz: library API fuzzing with lifetime-aware dataflow graphs". In: ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1070–1081. ISBN: 9781450392211. DOI: 10.1145/3510003.3510228. URL: https://doi.org/10.1145/3510003.3510228.

[12] Samuel Groß et al. "FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities". In: Jan. 2023. DOI: 10.14722/ndss.2023.24290. URL: https://doi.org/10.14722/ndss.2023.24290.

[13] Kyriakos Ispoglou et al. "FuzzGen: Automatic Fuzzer Generation". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2271–2287. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou.

[14] Sam Leffler and Silicon Graphics, Inc. *LibTIFF - TIFF Library and Utilities*. [Online; accessed 21-June-2026]. 1997. URL: http://libtiff.gitlab.io/libtiff/.

[15] *libffi: A portable foreign function interface library*. [Online; accessed 20-June-2026]. 2024. URL: https://sourceware.org/libffi/.

[16] Jiayi Lin et al. "Automatic Library Fuzzing through API Relation Evolvement". In: Jan. 2025. DOI: 10.14722/ndss.2025.240750. URL: https://doi.org/10.14722/ndss.2025.240750.

[17] LLVM Project. *libFuzzer - a library for coverage-guided fuzz testing*. [Online; accessed 17-June-2026]. 2026. URL: https://llvm.org/docs/LibFuzzer.html.

[18] LLVM Project. *ORC - Design and Implementation¶*. [Online; accessed 28-June-2026]. 2026. URL: https://llvm.org/docs/ORCv2.html.

[19] Barton P. Miller, Lars Fredriksen, and Bryan So. "An empirical study of the reliability of UNIX utilities". In: *Commun. ACM* 33.12 (Dec. 1990), pp. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279. URL: https://doi.org/10.1145/96267.96279.

[20] *perf: Linux profiling with performance counters*. [Online; accessed 17-June-2026]. 2026. URL: https://perfwiki.github.io/main/.

[21] The OpenSSL Project. *OpenSSL: Cryptography and SSL/TLS Toolkit*. [Online; accessed 21-June-2026]. 2026. URL: https://www.openssl.org/.

[22] Konstantin Serebryany et al. "AddressSanitizer: a fast address sanity checker". In: *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC '12)*. June 2012, pp. 309–318. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[23] Kostya Serebryany. "OSS-Fuzz - Google's continuous fuzzing service for open source software". In: Vancouver, BC: USENIX Association, Aug. 2017.

[24] Gabriel Sherman and Stefan Nagy. "No Harness, No Problem: Oracle-guided Harnessing for Auto-generating C API Fuzzing Harnesses". In: *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 2025, pp. 165–177. DOI: 10.1109/ICSE55347.2025.00239. URL: https://doi.org/10.1109/ICSE55347.2025.00239.

[25] Mitsunari Shigeo and contributors. *Xbyak: A JIT assembler for x86/x64 architectures supporting advanced instruction sets up to AVX10.2*. [Online; accessed 28-June-2026]. 2026. URL: https://github.com/herumi/xbyak.

[26] Prashast Srivastava and Mathias Payer. "Gramatron: effective grammar-aware fuzzing". In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2021. Virtual, Denmark: Association for Computing Machinery, 2021, pp. 244–256. ISBN: 9781450384599. DOI: 10.1145/3460319.3464814. URL: https://doi.org/10.1145/3460319.3464814.

[27] Daniel Stenberg and contributors. *c-ares: A C library for asynchronous DNS requests*. [Online; accessed 21-June-2026]. 2026. URL: https://c-ares.org/.

[28] Evgeniy Stepanov and Konstantin Serebryany. "MemorySanitizer: Fast detector of uninitialized memory use in C++". In: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2015, pp. 46–55. DOI: 10.1109/CGO.2015.7054186.

[29] The Clang Team. *libclang*. [Online; accessed 21-June-2026]. 2026. URL: https://clang.llvm.org/doxygen/group__CINDEX.html.

[30] The Clang Team. *LibTooling*. [Online; accessed 21-June-2026]. 2026. URL: https://clang.llvm.org/docs/LibTooling.html.

[31] The Clang Team. *SanitizerCoverage*. [Online; accessed 17-June-2026]. 2026. URL: https://clang.llvm.org/docs/SanitizerCoverage.html.

[32] The Clang Team. *UndefinedBehaviorSanitizer*. [Online; accessed 17-June-2026]. 2026. URL: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[33] Flavio Toffalini et al. "Liberating Libraries through Automated Fuzz Driver Generation: Striking a Balance without Consumer Code". In: *Proc. ACM Softw. Eng.* 2.FSE (June 2025). DOI: 10.1145/3729365. URL: https://doi.org/10.1145/3729365.

[34] Michał Zalewski. *American Fuzzy Lop — Technical Details*. 2014. URL: https://lcamtuf.coredump.cx/afl/technical_details.txt (visited on 06/30/2026).