Interpreter Memory Safety via Differential Fuzzing with a CHERI on Top

Kai Feng
University of Glasgow
Glasgow, United Kingdom
kai.feng@glasgow.ac.uk

Huanting Wang
University of Leeds
Leeds, United Kingdom
H.Wang7@leeds.ac.uk

Jeremy Singer
University of Glasgow
Glasgow, United Kingdom
Jeremy.Singer@glasgow.ac.uk

Zheng Wang
University of Leeds
Leeds, United Kingdom
z.wang5@leeds.ac.uk

Abstract

Memory safety is a critical issue in embedded systems. Although high-level languages like MicroPython simplify IoT development, their C-based runtimes remain vulnerable to memory errors triggered by Python code or native extensions. The CHERI (Capability Hardware Enhanced RISC Instructions) architecture offers hardware-enforced memory safety, but its effectiveness for exposing latent bugs in real-world interpreters has not yet been fully explored. We present DIFFCHERI:FRUITFLY, a novel differential testing framework for systematically uncovering memory defects in MicroPython across conventional (x86/ARM) and CHERI-enabled (Arm Morello) platforms. We mine historic vulnerabilities from diverse Python runtimes to extract recurring stress patterns, then use a large language model to generate new test programs, and apply Concrete Syntax Tree (CST) mutation to diversify inputs. On 24-hour automated testing, our framework executed 8,189 generated programs on MicroPython v1.20 and the development branch, identifying 40 distinct defects in the conventional build and 51 in the CHERI port. Memory errors that caused silent corruption or weak symptoms on conventional hardware were converted into precise capability faults on CHERI. These results show that CHERI not only shrinks the attack surface but also serves as an effective memory safety oracle for revealing latent vulnerabilities in embedded interpreters.

CCS Concepts: • **Software and its engineering → Runtime environments**.

Keywords: Differential Testing, MicroPython, CHERI

ISMM '26, Boulder, CO, USA

ACM ISBN 979-8-4007-2720-7/2026/06
https://doi.org/10.1145/3814942.3816133

ACM Reference Format:
Kai Feng, Huanting Wang, Jeremy Singer, and Zheng Wang. 2026. Interpreter Memory Safety via Differential Fuzzing with a CHERI on Top. In *Proceedings of the 2026 ACM SIGPLAN International Symposium on Memory Management (ISMM '26), June 16, 2026, Boulder, CO, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3814942.3816133

1 Introduction

Embedded system development increasingly relies on high-level languages like MicroPython [6, 13] to improve developer productivity. However, their runtime systems and interpreters are largely implemented in low-level languages like C, inheriting long-standing memory-safety risks [5, 26], including out-of-bounds access, use-after-free, integer overflow, and related errors [12, 29].

Beyond vulnerabilities in the runtime itself, MicroPython exposes low-level interfaces (e.g., FFI and native modules) that allow user programs to interact with C libraries. While powerful, these interfaces also enable client code to misuse unsafe libraries, for example, by passing invalid pointers, violating expected memory ownership, or triggering undefined behavior in underlying C components. Publicly disclosed vulnerabilities confirm that such defects persist in practice, including heap buffer overflows and use-after-free bugs[1] .

Emerging hardware-based memory protection mechanisms like CHERI (Capability Hardware Enhanced RISC Instructions) provide hardware-enforced memory safety by replacing conventional pointers with unforgeable, bounded capabilities [44, 46]. Although CHERI was originally designed for high-performance processors, recent work has extended the model to microcontroller-scale devices [1, 2].

While CHERI offers a promising path toward stronger memory safety, its practical effect on complex, real-world software stacks is still under active research [36, 43]. Early evidence of CHERI's effectiveness for programming language interpreters has mainly relied on small benchmarks or manual analysis [17]. A larger empirical study is needed to assess

[1] https://www.cve.org/CVERecord/SearchResults?query=micropython

how CHERI's architectural protections change the security properties of widely deployed language runtime systems such as the MicroPython interpreter [8].

Evaluating this effect raises a significant testing challenge. Traditional coverage-based fuzzing techniques struggle in this domain. Random mutations often generate programs that are syntactically invalid and rejected by the interpreter [30]. General-purpose fuzzers also lack the semantic guidance needed to reach deep internal components of the interpreter, such as the garbage collector or C-based modules, where memory-safety errors are most likely to appear [16]. To test the interpreter's C core systematically, we need a method to generate a large, diverse set of valid, semantically plausible programs that target these high-risk features.

We present diffCHERI:FruitFly, a differential testing framework that combines structure-aware test generation with CHERI-aware execution. Differential testing serves as a practical oracle by comparing the behaviour of two implementations of the same specification. In our framework, each generated program is executed on two MicroPython builds: a conventional (non-CHERI) build running on a standard x86 architecture and a CHERI-enabled build running on CHERI hardware (Arm Morello [15]).

Our key insight is that the CHERI architecture will act as a sensitive and deterministic oracle for memory errors. A latent bug, whether in the interpreter or triggered by client misuse of unsafe libraries, may cause silent memory corruption on the conventional build, but is expected to trigger a precise capability fault on the CHERI build. This difference in outcome provides strong evidence of a memory-safety violation that CHERI detects and stops. In this way, diffCHERI:FruitFly not only uncovers runtime defects but also detects incorrect or unsafe usage patterns in user programs that interact with low-level libraries.

To drive this differential harness, we develop a multi-stage test generation pipeline. The process begins by seeding a corpus with test cases derived from historic bug reports and security advisories for two open-source runtimes: i.e. MicroPython and CPython. This seed set is then expanded using a large language model (LLM), guided by risk-focused prompts to synthesize new, valid test cases that target known problematic patterns. Finally, we apply a syntax-directed mutation tool (based on the LibCST library[2]) that performs syntactically valid transformations on the concrete syntax tree to broaden semantic coverage.

We applied our framework to test MicroPython v1.20 and the latest development release, v1.27, executing 8189 generated programs on 24-hour automated test runs. Our experiments revealed 40 unique defects in the baseline non-CHERI build (a total of 52 bugs, including libffi) and 51 in the CHERI build, of which 38 are bugs of MicroPython runtime and 13 are bugs of illegal use of unsafe libraries in the client code. These results show that CHERI can expose latent memory-safety violations that cannot be detected on conventional architectures, while diffCHERI:FruitFly further reveals unsafe behaviours arising from client code that misuses low-level libraries.

This paper makes the following contributions:

- A structure-aware test generation pipeline that combines LLM-guided seeding with a LibCST-based mutator to create a corpus of 8,189 valid MicroPython programs stressing memory-sensitive features.
- A dual-lane execution harness that collects matched telemetry from non-CHERI and CHERI builds, enabling capability faults to act as a differential oracle.
- An empirical evaluation of MicroPython v1.20 and the current development branch (v1.27), identifying 51 unique bugs in the CHERI port, 40 in the baseline, and 39 in the latest release, with CHERI revealing latent defects missed by conventional builds.

2 Technical Background

Systems software is riddled with memory safety bugs such as buffer overflows and use-after-free vulnerabilities, which remain a dominant source of vulnerabilities and failures. For example, recent analysis from Microsoft reveals roughly 70% of CVEs are due to memory corruption [25]. Such spatial and temporal memory errors in C or C++ code can be exploited or cause unpredictable behavior, limiting the use of low-level languages in safety-critical systems. Traditional software defenses include runtime checks (e.g. AddressSanitizer) or safe languages (e.g. Rust), but these often incur high overhead or lack full coverage. In contrast, hardware mechanisms have been proposed to enforce memory safety with lower impact on performance [8].

CHERI memory protection: Capability Hardware Enhanced RISC Instructions (CHERI) is an instruction-set architecture (ISA) extension that augments conventional processors with *architectural capabilities* to provide fine-grained memory safety and scalable compartmentalisation [45]. Instead of layering additional checks around unsafe code, CHERI changes the basic contract between software and memory by replacing raw integer pointers with hardware-enforced, unforgeable *capabilities*. As we show later, this design is the main driver of the divergent behaviour observed in our differential testing experiments. A CHERI capability is an atomic token that combines (i) an integer address, (ii) metadata describing authority, and (iii) a 1-bit validity tag [2]. For example, on a 64-bit CHERI architecture, a conventional 64-bit pointer is replaced by a token that holds a 64-bit address, roughly 64 bits of metadata, and a 1-bit tag.

[2] https://libcst.readthedocs.io/en/latest/index.html

MicroPython: The choice of MicroPython as the software under test is deliberate and strategic [37]. As a high-level, dynamically typed language, Python is designed to be memory-safe from the programmer's perspective. However, the MicroPython interpreter that executes Python code is itself a complex program written primarily in C. Subjecting this interpreter to differential testing provides a clear view of how CHERI's protections can strengthen critical runtime systems [7].

MicroPython has been ported to CHERI [21, 22], which makes it an excellent candidate for our differential testing framework. The interpreter's complexity and reliance on C for core functionality mean that it likely contains hidden memory-safety issues that CHERI can help expose and mitigate. By running the same Python scripts on both the CHERI-enabled and non-CHERI builds of MicroPython, we can directly observe how CHERI's architectural features shape the interpreter's behaviour in the presence of memory errors. This setup allows us not only to identify potential vulnerabilities in the interpreter but also to demonstrate how CHERI can turn these vulnerabilities from silent, exploitable flaws into well-defined, manageable exceptions.

Differential testing, also called back-to-back testing, detects semantic or logical bugs by running the same inputs on two or more implementations of the same specification and comparing their outputs [23]. Its main strength is that it can expose subtle differences that do not cause obvious failures such as crashes or assertion violations [9]. This is especially useful when a single correct result is hard or impossible to define, a challenge known as the test oracle problem [4]. Differential testing addresses this by treating the implementations as oracles for each other. The key assumption is that, despite internal differences, they should show equivalent external behaviour for the same inputs [10].

This technique has been widely used for complex software. For example, Csmith [47] generates random C programs to find compiler miscompilations by comparing the outputs of GCC and Clang. More recent tools such as YARGen generate well-defined C and C++ programs and have found hundreds of compiler bugs [20]. RustSmith extends this idea to Rust, generating random programs to test `rustc` and other compilers [35, 40]. Similar differential frameworks have also been applied to interpreters and virtual machines, including deep differential testing of JVM implementations [9]. These approaches usually combine random or generative test case generation with pruning or coverage guidance to produce valid and interesting tests.

We apply differential testing to MicroPython on CHERI with a different objective from the conventional one. The non-CHERI system is the baseline, representing the behaviour of a C interpreter on hardware without architectural memory protection. Its outcomes, including crashes, silent data corruption, and other unstable behaviour, provide the reference point. The CHERI system, running the same MicroPython interpreter compiled for the CHERI ABI, is then compared against this baseline.

3 Differential Testing Overview

Figure 1 provides a high-level overview of our differential testing framework, which is organised into three layers. The first layer constructs a corpus of valid MicroPython programs. We start from curated scripts that correspond to known CVEs in CPython and MicroPython and to public bug reports. On top of these seeds, an LLM-based generator, guided by prompts that encode patterns known to be risky in MicroPython, synthesises new programs while staying within the subset of Python features supported by our target platform. A LibCST-based mutator then applies structure-preserving transformations to broaden the input space and increase syntactic and semantic diversity. Each candidate input is validated, de-duplicated, and added to a unified corpus that is shared by both execution lanes in later layers.

The second layer runs each test program on two builds of MicroPython. The left lane uses a normal (non-CHERI) build of MicroPython, executed inside a sandbox on a conventional system with strict time and memory limits. The right lane uses a CHERI-enabled build running on hardware that supports capability enforcement, the ARM Morello prototype board. The harness on each side captures the program's output, exit code, and any signals or crashes; on the CHERI side, it also logs any capability fault details (such as bounds, tag, or permission violations). To enable a fair comparison, the harness normalizes non-deterministic aspects of outputs (e.g., memory addresses in error logs or ephemeral file names in tracebacks).

The third layer compares the two runs. It classifies the pair into one of several categories. The category that matters most for memory safety is the one where the baseline crashes or exhibits undefined behaviour while the CHERI build reports a capability violation or exits normally. The comparator assigns a stable signature to each discrepancy using a small set of fields drawn from the termination state and from a normalized summary of the top frames. A reducer then shrinks the input. The result feeds back into the corpus and into the prompt context for the generator so that future runs start from richer seeds.

4 Methodology

The implementation of our framework follows the three-layer structure described above. In particular, the methodology is divided into input generation (layer 1), execution and monitoring (layer 2), and differential analysis plus bug triage (layer 3).

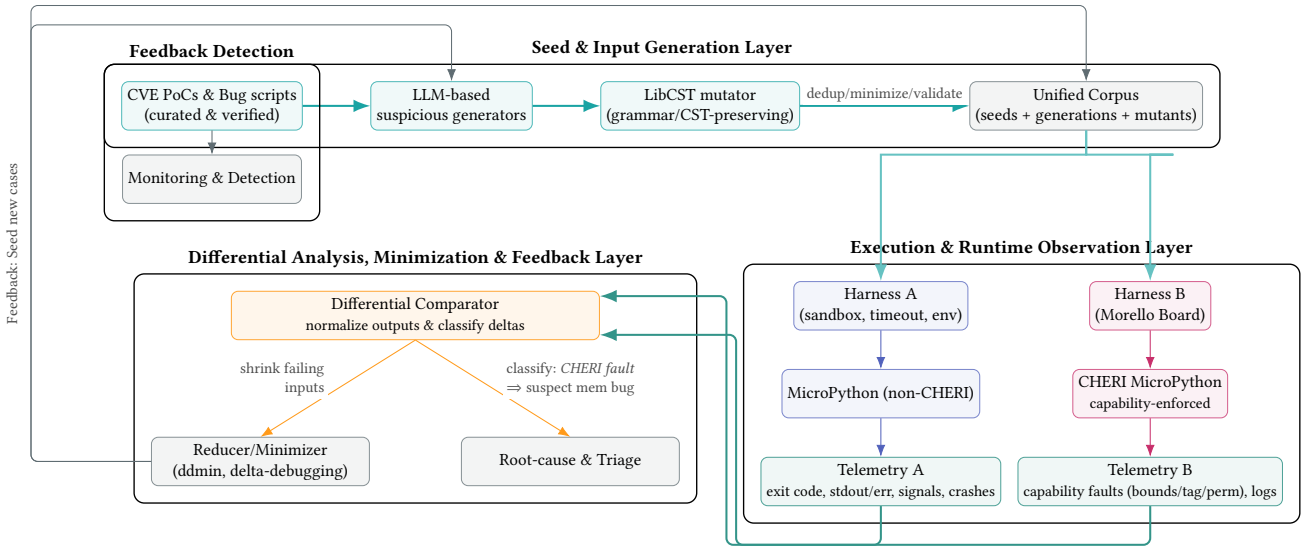

Figure 1. Differential testing framework for MicroPython with CHERI.

Table 1. Vulnerability and Bug Report Classification

Category	Verified Reports	CVEs
Raw memory, buffer protocol & view lifetime	16	19
Binary conversions & bigint corners	4	14
FFI / native emitters	1	5
Parsers, codecs & compressors in C	68	15
Filesystem, VFS & Race Conditions	12	19
MMIO & peripherals (embedded targets)	0	10
Interpreter internals, exceptions & GC	38	18
Total	**139**	**100**

4.1 Input Generation and Corpus Management

Our approach to generating test inputs consists of three stages: **(A)** seeding the process with a collection of real-world bug scripts and CVE proofs-of-concept, **(B)** using an LLM-based generator to produce new candidate programs, and **(C)** applying a LibCST-based mutator to systematically introduce variations.

4.1.1 A: The Collection of CVE PoCs and Bug Reports.

Our methodology begins with a curated corpus of proof-of-concept (PoC) scripts derived from known CVEs and bug reports affecting both CPython and MicroPython. This initial set provides a foundation grounded in real-world vulnerabilities. Through the analysis of these reports, we identified common bug patterns and high-risk programming constructs, particularly those relevant to the resource-constrained environment of the MicroPython interpreter. These seed scripts serve as the basis for further generation and mutation, allowing us to explore novel security issues beyond the known vulnerability landscape.

The classification results are summarised in Table 1. The analysis reveals several critical areas. These components must handle complex, untrusted data formats, making them frequent sources of crashes. Listing 1 demonstrates a typical edge case in the Parsers, codecs & compressors category, where the parser is stressed by a combination of named arguments and the walrus operator within a function call, targeting the tokeniser and compiler logic.

Listing 1. Parser edge case targeting syntax handling

```
def f(a, b):
    pass
f(None, x=b"", y:=True)
```

The Raw memory, buffer protocol & view lifetime category is particularly critical, accounting for 19 CVEs. This category covers direct memory manipulation, where logical errors can lead to buffer overflows or use-after-free scenarios. Listing 2 illustrates a self-referential buffer assignment. This pattern forces the interpreter to resize the bytearray while simultaneously reading from it, a complex operation that often exposes heap corruption vulnerabilities.

Listing 2. Self-referential buffer expansion

```
b = bytearray(b'1234567')
for i in range(3):
     b[-1:] = b
```

The interpreter internals, exceptions, and GC category covers some of the most tightly coupled parts of the system, where faults can destabilise the runtime. Listing 3 shows a seed script that stresses the garbage collector by creating

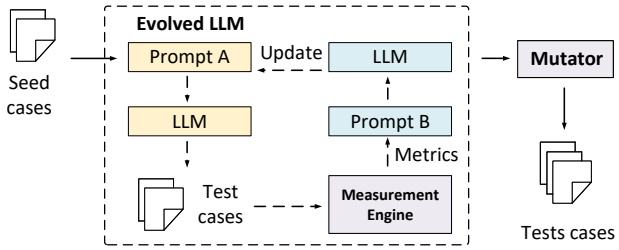

Figure 2. Overview of the test case generation pipeline.

many short-lived string objects. This drives frequent allocation and deallocation, increasing the chance of exposing reference-management errors and memory leaks.

Listing 3. Garbage Collection stress test

```
def f():
    for i in range(10000):
        s = str(i)
        print(s)
```

Other notable categories include:

- **Binary conversions & bigint corners:** Focuses on vulnerabilities in converting Python integers to byte sequences, often leading to size calculation errors or integer overflows.
- **FFI / native emitters:** Covers interactions between Python and C libraries, where mismatches in function prototypes or unsafe pointer arithmetic can cause memory corruption.
- **Filesystem, VFS & Race Conditions:** Includes errors in file I/O, such as length mismatches in buffer operations and use-after-close errors.
- **MMIO & peripherals:** Specific to embedded targets, dealing with direct hardware access where misalignment or arbitrary memory access can cause system instability.

4.1.2 B: LLM-Based Generators.

As shown in Figure 2, our test case generator consists of an evolved LLM-based component and a structural mutator, which together produce test cases for our testing engine. Starting from the seed cases described in Section 4.1.1, we use these seeds as input and apply in-context prompting to generate test cases that may trigger bugs or vulnerabilities. Specifically, we employ an adaptive evolutionary loop [28] that analyses execution metrics provided by the measurement engine and iteratively refines the prompt used for generation. This enables the prompt to evolve over time, producing test cases that are increasingly likely to expose bugs. Finally, a LibCST-based structural mutator is applied to further augment the generated test cases.

Evolved Prompts. The supplementary material provides the prompts used in this work. As shown in Figure 2, Prompt A is used to generate MicroPython test cases, while Prompt B is used to evolve Prompt A so that it produces cases more likely to trigger bugs. In this work, we perform 100 evolutionary iterations to update Prompt A, after which the final evolved prompt is used to generate test cases.

The raw outputs are then sanitised to ensure executability. This process includes stripping Markdown formatting, removing language tags, trimming trailing syntax errors, and automatically invoking zero-argument functions.

Execution and Reward Function. Execution metrics drive the evolution strategy. Specifically, generated test cases are executed in an isolated environment (e.g., Docker). The runner captures execution states, distinguishing between standard return codes ($RC \in 0, 1$), timeouts ($RC = 124$), and runner errors ($RC = 125$). When a crash or timeout is detected, the system attaches GDB to extract stack traces and enables per-test coverage collection (GCOV). These execution results are then fed back to the LLM to update the prompt for the next evolutionary iteration.

We employ a reward function to quantify the likelihood that a test case will help the testing engine detect vulnerabilities. The higher-is-better reward score is derived from two primary components:

`Coverage depth`, which measures the extent of code coverage. Higher coverage indicates that a test case explores more execution paths and is therefore more likely to expose bugs. Specifically, coverage depth is computed as the mean of line and function coverage, augmented by the normalised depth of the GDB stack trace.

`Crash severity`, which captures whether execution terminates abnormally. Test cases that crash, time out, or exit with a non-standard return code are considered more valuable for bug discovery and are assigned higher rewards.

The scoring function maps *bug-inducing* behaviours (crashes, timeouts, or non-standard return codes) to the interval $(0.6, 1.0)$, while non-buggy executions are mapped to $(0.0, 0.4)$. The geometric mean of per-seed rewards is used as the fitness metric passed to the evolutionary engine.

Data Augmentation. To improve diversity, we generate variants of high-performing test cases using LibCST-based mutation [24]. This process targets specific intents (e.g., boundary testing) and prioritises semantics-preserving transformations. We evaluate these variants using a *family score*, which extends the standard seed reward by incorporating the following components:

- **Bug ratio:** The frequency of faults triggered within the variant family.
- **Diversity metric:** The number of unique signature tuples (comprising status, return code, top stack frame, and coverage flags).
- **Novelty:** The degree of deviation from a rolling historical coverage baseline (e.g., the 75th percentile).

A weighted combination of the seed score and the family score determines the final selection preference, with higher scores corresponding to higher-quality test cases.

Adaptive Loop and Refinement. The system operates as a closed-loop feedback mechanism. High-reward seeds and their variants are archived and reintroduced in subsequent iterations as few-shot exemplars or used to identify risky code patterns. Negative feedback is also incorporated to discourage code structures that are computationally expensive but yield no faults. Low-reward or redundant cases are discarded.

The aggregate fitness drives a MAP-Elites search over the prompt space, enabling the harness to dynamically optimise prompt content based on coverage gains and crash diversity until convergence.

4.1.3 C: LibCST Mutator.

After generating new test cases using the LLM, we further augment them by designing a *context-aware, type-aware Python code mutator* based on LibCST. LibCST is a Python parsing and rewriting toolkit that preserves formatting details such as whitespace and comments, ensuring that mutated code remains syntactically valid and stylistically consistent with the original source [18]. We define a series of mutation rules to introduce subtle variations into the programs while respecting contextual constraints (e.g., scope and syntax) and optional type constraints [11]. The goal is to generate a diverse set of program variants for testing without breaking syntax or introducing manifest type inconsistencies.

Key design goals of the mutator include:

(i) **Preserve syntactic validity and formatting.** By leveraging LibCST's CST, all transformations maintain valid Python syntax and preserve layout, comments, and formatting of unmodified parts of the code. This prevents trivial syntax errors and keeps mutations semantically readable.

(ii) **Context-aware replacements.** The mutator uses scope-sensitive context pools of CST nodes to guide replacements. Any code fragment selected for replacement is substituted with another fragment of a compatible category (expression, statement, etc.), drawn either from elsewhere in the program or from a template library. This ensures that replacements respect the surrounding context (for example, an expression is only replaced with another valid expression of an appropriate type or structure).

(iii) **Multi-pass mutation pipeline.** Instead of applying a single mutation in isolation, the mutator supports executing multiple passes of different mutation operator families (structural, peephole, chaotic, etc.) sequentially in one run. This multi-pass approach allows both coarse-grained structural changes and fine-grained modifications to be introduced, increasing the chance of complex interactions. Undesired mutation steps can be rolled back or skipped (with type-check gating, as described below) to maintain overall correctness.

(iv) **Type-budget enforcement.** To keep mutations semantically plausible, an optional type-checking gate (using MyPy[3]) enforces a "type budget." Before any mutations, we record the original program's type errors as a baseline. Each candidate's mutated program is only accepted if it does not increase the number of type errors beyond that baseline. This prevents the mutator from introducing obvious type violations (such as arity mismatches in function calls or incompatible assignments), thereby preserving a baseline level of semantic consistency.

(v) **Runtime anomaly scoring.** The mutator can optionally execute the final mutated program under the MicroPython interpreter with a timeout, in order to detect crashes, assertion failures, or infinite loops (hangs). A scoring mechanism assigns higher scores to mutations that trigger abnormal behaviour (non-zero exit status, crashes, timeouts), which is useful for prioritising interesting or bug-inducing mutants in a fuzzing campaign.

(vi) **Deterministic, reproducible generation.** All randomness in the mutator is driven by a single master seed. Given the same seed and configuration, the mutator will produce the same sequence of mutations every time. This determinism greatly aids debugging and evaluation by ensuring that experiments are repeatable.

The mutation workflow is formally defined in Algorithm 1. At a high level, the mutator accepts an input source code S and a configuration cfg that specifies mutation parameters, including the operator selection, mutation count, and type safety constraints. The process executes in four primary phases:

(i) **Parsing and Indexing:** The source code S is first parsed into a LibCST Concrete Syntax Tree (CST), denoted as T. Simultaneously, the system generates wrapper metadata W (utilising LibCST's `ScopeProvider` to track variable scope) and constructs a `ContextIndex` I. The index I categorises nodes by syntactic type, such as expressions, control-flow blocks (`if`/`while`), and definitions. It creates pools of candidate nodes. This separation allows subsequent mutation passes to access context-aware replacement targets efficiently.

(ii) **Pipeline Assembly:** Based on the specified mutation profile, the system constructs a sequence of transformation passes Π (BuildPipeline). Each pass $p \in \Pi$ corresponds to a specific family of mutation operators, such as structural changes or peephole optimisations.

(iii) **Multi-pass Mutation with Type Gating:** The mutator iterates through each pass $p \in \Pi$, applying it to the

[3] https://github.com/python/mypy

Algorithm 1: Context- and Type-Aware Mutation Workflow

Input: Source code S, Configuration cfg
Output: Mutated source code S_{final}
// 1. Parse code and build pipeline
$T, W, I \leftarrow$ ParseAndIndex(S)
$\Pi \leftarrow$ BuildPipeline(cfg)
// 2. Get baseline type errors
$B \leftarrow$ cfg.type_safe ? RunMyPy(S) : ∞
// 3. Apply mutation passes
for $p \in \Pi$ **do**
  $T_{cand} \leftarrow p$.Apply(T, W, I)
  $S_{cand} \leftarrow$ CodeGen(T_{cand})
  // Reject if it exceeds the error budget
  if cfg.type_safe ***and*** *RunMyPy*(S_{cand}) $> B$ **then**
    continue
  $T \leftarrow T_{cand}$ // Accept mutation
// 4. Finalize and emit
$S_{final} \leftarrow$ CodeGen(T)
if cfg.score_runtime **then**
  RuntimeScore(S_{final}, cfg)
Emit(S_{final}, cfg)
return S_{final}

current CST state $T^{\star}$ (initialized as T). Unlike simple text replacement, the transformation function p.Apply (Line 6) operates on the tree structure, utilising the metadata W and context index I to perform valid substitutions. This produces a candidate tree T_{cand}.
To enforce the "type budget," the system must validate T_{cand}. Since static analysis tools require textual input, T_{cand} is serialised back into source code S_{cand} via the CodeGen function. If type safety is enabled (cfg.type_safe), the system runs MyPy on S_{cand}. If the resulting error count E_{cand} exceeds the baseline B, the candidate T_{cand} is rejected, and the loop continues with the previous state $T^{\star}$. This rollback mechanism ensures that the mutation chain does not accumulate type inconsistencies. If accepted, $T^{\star}$ is updated to T_{cand}.

(iv) **Runtime Evaluation and Output:** Upon completion of all passes, the final CST $T^{\star}$ is materialized into the final source code S_{final}. This code is optionally executed in a sandbox (e.g., MicroPython) via RuntimeScore to detect crashes or timeouts. Finally, S_{final} is emitted for downstream testing.

As summarised in Algorithm 1, each transformation pass targets a specific *operator family*. If a candidate mutation violates the type budget, the system gracefully rejects that specific variant and proceeds, maintaining robustness during automated fuzzing.

The mutator provides four families of operators that manipulate the CST while preserving syntactic validity (Table 2). Structural/Block operators rewrite statements or blocks to explore control and data-flow modifications. Peephole/Op operators perform local edits to operators, literals, and expressions. Semantic/Path operators rewrite boolean logic and path conditions using equivalence rules to alter execution flow. Aggressive/Inflation operators stress parser and runtime limits without violating syntax. Certain operators (e.g., `WALRUS_INSERT`) are conditionally enabled based on target language version support.

Structural operators facilitate coarse edits via `REPLACE`, `ADD`, `DELETE`, and `SWAP`. Controlled code reuse is enabled by `REUSE`, `INJECT`, and `COMBINE`, while `TRY_WRAP` introduces exception contexts. Simple refactorings are performed by `INLINE_TEMP` and `EXTRACT_TEMP`. The Peephole family performs local adjustments: `ARITHMETIC_FLIP` and `LOGICAL_NEGATE` modify operators, whereas `BOUNDARY_OFF_BY_ONE` adjusts integer comparisons. Additionally, these operators toggle decorators and perturb parameter defaults (`PARAM_DEFAULT_MUTATE`). Semantic operators rewrite standard idioms (`SEMANTIC_AWARE`), invert conditions (`PATH_CONDITION`, `DEMORGAN`), and conditionally insert assignment expressions (`WALRUS_INSERT`). Finally, the `CHAOS` operator inflates literals, collections, and expression depth to probe resource limits while maintaining program validity.

Overall, the context- and type-aware mutator provides a robust methodology for generating program variants. It balances exploration (through aggressive and chaotic mutations that can uncover edge cases) with soundness (through context awareness and optional type checking to keep mutants valid and interpretable).

4.2 Layer 2: Execution and Runtime Observation

The second layer executes each program from the corpus on both the baseline and CHERI-enabled builds of MicroPython and records detailed, normalised telemetry.

Each interpreter process is launched in a strict sandbox that enforces resource limits using `setrlimit` on CPU time, address space, and file writes. This prevents non-terminating programs from stalling the framework and contains side effects. The harness captures the process exit code, any terminating signal number, and the complete `stdout` and `stderr` streams. In addition, the CHERI harness logs detailed information about any capability faults (such as bounds, tag, or permission violations) that occur during execution.

4.3 Layer 3: Differential Analysis and Triage

The final layer of our framework performs a differential analysis between the baseline and CHERI-enabled builds. This layer aggregates paired execution logs into crash clusters and generates comparative reports, distinguishing between

Table 2. Mutation operators implemented in the mutator.

Family	Operator	Effect
Structural / Block	REPLACE	Replace node using template or context pool.
	ADD	Insert simple statements to blocks (diversity).
	DELETE	Remove node; keep `pass` if block empties.
	SWAP	Swap sibling statements (module or nested blocks).
	REUSE	Replace with deep-clone from in-file pool.
	INJECT	Replace with deep-clone from secondary file.
	COMBINE	Combine templates/subtrees (conservative).
	TRY_WRAP	Wrap stmt/CF in `try/except Exception: pass`.
	INLINE_TEMP/EXTRACT_TEMP	Basic temp var rewrites (local refactoring).
Peephole / Op	ARITHMETIC_FLIP	Swap `+` ↔ `-`, `*` ↔ `//`, etc.
	LOGICAL_NEGATE	Insert/remove `not` around boolean exprs.
	BOUNDARY_OFF_BY_ONE	±1 tweaks of integer comparators.
	DECORATOR_TOGGLE	Cycle `staticmethod` → `classmethod` → `property`.
	PARAM_DEFAULT_MUTATE	Modify parameter default values (int/string).
Semantic / Path	SEMANTIC_AWARE	`len(x)==0` ↔ `not x`; `is None` ↔ `== None`; identity arith.
	PATH_CONDITION	In tests only: flip `and`/`or`, flip relations, ±1 thresholds.
	DEMORGAN	Apply De Morgan's laws with safe parentheses.
	WALRUS_INSERT	Insert `:=` in if/while/boolean sub-exprs (CPython).
Aggressive / Inflation	CHAOS	Oversized integers/strings/bytes, grow collections, deepen expr, add blocks.

platform-agnostic logic errors and CHERI-specific memory safety violations.

4.3.1 Crash Categorisation and Signature Generation. The system parses execution logs (JSON artifacts) to classify run outcomes. We define a crash event as any execution yielding a return code $RC \notin \{0, 1\}$. When a crash occurs, the system extracts stack traces from the GDB output. To handle the variability of raw logs, we employ a normalisation pipeline that:

1. **Extracts Frames:** Stack frames are parsed from the structured logs, collapsing recursive frame sequences and removing duplicates.
2. **Prioritises Relevance:** Domain-specific filters (e.g., regex matching) are applied to prioritise frames associated with the target application (e.g., MicroPython internals) over generic library code.
3. **Generates Signatures:** A unique crash signature is constructed using the top-most relevant frame (formatted as `function @ file:line`)

Test cases lacking stack traces are segregated into a generic "Missing Backtrace" cluster, indexed by return code and signal.

4.3.2 Differential Comparison. To isolate CHERI-specific faults, we compare the aggregated crash signatures of the baseline build (B) against those of the CHERI build (C). The comparator normalises signature text by stripping non-deterministic elements, such as memory offsets (e.g., `+0x...`). We then calculate the frequency of test cases falling into each signature bucket for both builds. The analysis identifies three distinct categories:

- **CHERI-Exclusive ($S_C \setminus S_B$):** Signatures appearing only in the CHERI build.
- **Baseline-Exclusive ($S_B \setminus S_C$):** Signatures appearing only in the baseline build.
- **Shared ($S_C \cap S_B$):** Signatures present in both, sorted by the magnitude of the frequency difference $|\Delta|$.

4.3.3 Semantic Interpretation and Feedback. The categorisation enables automated triage of the findings.

1. CHERI-Exclusive signatures are interpreted as latent memory-safety violations (e.g., spatial safety bounds checks) that do not corrupt the state sufficiently to crash the baseline build but are trapped by architectural capabilities.
2. Shared signatures with a higher frequency in the CHERI build suggest that CHERI exposes more code paths triggering the same underlying defect.
3. Baseline-Exclusive crashes often reflect harness noise or environment mismatches rather than genuine vulnerabilities. Timeouts and harness errors keep separate codes, avoiding confusion with behavioural divergence.

This differential signal closes the feedback loop, guiding the LLM generator to prioritise prompt strategies that yield CHERI-specific faults.

5 Evaluation

5.1 Experiment Setup

Our experimental design uses differential testing to evaluate the impact of the CHERI architecture on the MicroPython interpreter. To achieve this, we established two parallel execution environments. The first is a control environment, which runs a standard, non-CHERI build of MicroPython on a conventional Linux system (x86-64, Ubuntu 24.04 LTS, GCC 15.2.0 and ARMv8-32, Raspberry Pi 3B). This setup provides a baseline for the interpreter's expected behaviour without hardware-based memory safety enhancements. The second is the experimental environment, where MicroPython is compiled for and executed on the CHERI-enabled Morello platform (AArch64 with CHERI, CheriBSD 24.05, CHERI LLVM 15.0.0). This allows us to assess how CHERI's hardware-enforced memory safety influences the interpreter's behaviour under identical test conditions.

Our testing methodology accounts for several key variables. Since the CHERI build of MicroPython does not support the libffi module, we conduct separate test runs on the baseline system both with and without this module. This ensures that our comparisons accurately isolate the effects of the CHERI architecture. We also run all test cases against the latest official version of MicroPython (1.27-preview) to help distinguish between pre-existing bugs in the core project and unique issues discovered during our analysis.

A custom harness is deployed in both environments to automate test execution and data collection. Each harness is responsible for running the same unified set of test scripts, enforcing resource limits like timeouts and memory usage, and capturing detailed telemetry. The data collected includes standard output and error streams, exit codes, and operating system signals. For the CHERI environment, the harness also records specific information about any capability violations, offering direct insight into hardware-level memory protection events.

5.2 Testcase Generation

Test case generation is central to our evaluation framework. We begin with a curated seed corpus derived from known vulnerabilities, bug reports, and representative programming patterns likely to expose memory-safety faults in MicroPython.

We then expand this corpus with a multi-stage pipeline. First, we use OpenEvolve[4] to iteratively optimise generation prompts using feedback from the execution oracle. This feedback loop improves prompt quality and increases the rate of valid test generation. Once the prompts stabilise, we use an LLM-based code agent to generate new syntactically valid scripts that follow the structure of the seed corpus. We prefix-tune the model to favour constructs relevant to memory safety, including pointer manipulation, buffer operations, and dynamic allocation.

In parallel, we apply the context-aware LibCST mutator described in Section 4.1.3. The mutator performs transformations that preserve syntactic and, where required, semantic validity, so that the resulting programs remain executable Python scripts. It supports both type-safe and non-type-safe modes, allowing us to explore faults arising from both conventional memory misuse and type inconsistencies.

The final corpus combines the original seeds with the generated variants. We execute each script in both the baseline and CHERI-enabled environments, enabling direct comparison of behaviour. Diversity in the corpus is important because it improves coverage of subtle memory-safety faults that simpler tests may miss.

Using this pipeline, we generated 8,189 test cases from the initial seed corpus through rule-based transformations and context-aware mutation. We classify these test cases in Table 3 using the same taxonomy as the CVE PoCs and bug reports in Table 1.

Table 3. Distribution of testcases across different categories.

Category	Generated testcases
Raw memory, buffer protocol & view lifetime	702
Binary conversions & bigint corners	783
FFI / native emitters	401
Parsers, codecs & compressors in C	638
Filesystem, VFS & Race Conditions	682
MMIO & peripherals (embedded targets)	170
Interpreter internals, exceptions & GC	590
Total	**8189**

5.3 Results

After generating the testcases, we executed the testcases on these three distinct targets: the baseline non-CHERI build of MicroPython v1.20, the CHERI-enabled build of MicroPython v1.20, and the latest development version of MicroPython (1.27-preview). This section details the findings, beginning with the raw crash counts, followed by our de-duplication methodology, and concluding with an analysis of the unique bugs identified.

5.3.1 Initial Crash Analysis. The initial execution of the test suite produced a large number of crashes across all targets. As summarized in Table 4, we initially recorded 673 crashes on the non-CHERI build, 536 on the CHERI-enabled build, and 491 on MicroPython v1.27 preview. The "FFI / native emitters" category highlights a key difference, as the CHERI build lacks libffi support and thus produced no

[4] https://github.com/codelion/openevolve

crashes in this area. Conversely, the CHERI build detected a significant number of bugs in categories sensitive to memory layout, such as "Binary conversions & bigint corners" (202 vs. 133) and "MMIO & peripherals" (36 vs. 0).

5.3.2 Crash De-duplication Analysis. These initial crash counts are inflated by duplicate testcases. The generative and mutational nature of our fuzzer often produces many slight variations of a test case that all trigger the same underlying bug. To obtain an accurate count of unique vulnerabilities, we implemented an automated crash de-duplication pipeline.

The pipeline processes each crash as follows:

1. Filter for Crashes: Only test runs resulting in a crash (defined by non-zero and non-one exit codes) are retained for analysis.
2. Extract Stack Trace: The GDB debugger is used to generate a backtrace for each crash.
3. Generate Crash Signature: We parse the backtrace to create a unique 'crash signature'. By default, this signature is defined by the top-most function call within the MicroPython source code (e.g., mp_obj_subscr @ obj.c:538). Frames from external libraries, such as libc, are ignored to ensure the signature is specific to the project's code.

All test cases that produce the same crash signature are grouped into a single crash cluster. We reserve the term defect for manually triaged root causes; one defect may yield multiple clusters, and one cluster may still conflate multiple causes.

5.3.3 Deduplicated Crash Clusters and Manual Triage. Applying this de-duplication process reveals a more precise landscape of the discovered bugs, as shown in Table 5. In total, we identified 47 bugs (39 non-libffi, 8 libffi-related) on the ARM Cortex architecture, 52 bugs (40 non-libffi, 12 libffi-related) on the x86 architecture, and 51 unique bugs on the CHERI build. Additionally, MicroPython v1.27 preview contained 39 unique bugs (26 non-libffi, 13 libffi-related). When comparing the core interpreter (excluding libffi), the CHERI-enabled build exposed more unique memory safety bugs than the baseline non-CHERI build. This result is consistent with CHERI's design, as its hardware-enforced capability bounds and permissions transform latent memory errors, which may not cause a crash on a conventional architecture, into observable faults.

Additionally, there are also differences in the number of test cases that trigger each unique bug, which also proves CHERI's capability to catch more memory safety bugs. Figure 3 illustrates this by comparing trigger counts for the top 10 bugs identified on the CHERI build. For several bugs related to memory access, the detection gap is stark. For instance, a bug in *array_subscr()* was triggered by 64 testcases on CHERI but was never detected on the non-CHERI build.

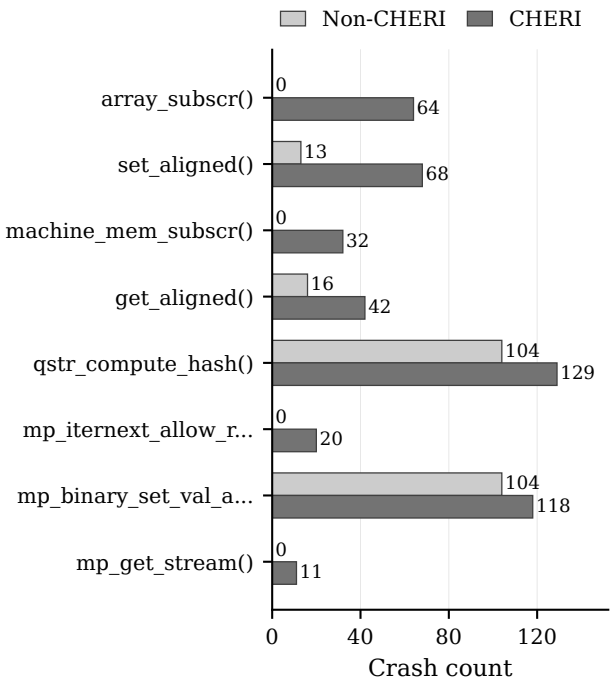

Figure 3. Crashes found on CHERI-build and non-CHERI build.

Similarly, a bug in *machine_mem_subscr()* was found by 32 testcases on CHERI and none on the baseline.

The differences between the CHERI and non-CHERI builds are shown in Table 5. They show that the CHERI build can expose more unique bugs. Even when the same bug appears on both builds, more test cases trigger it on the CHERI build and are missed on the non-CHERI build, as shown in Figure 3. We select the top 10 bugs found on CHERI-enabled MicroPython. The light grey bars show the number of test cases that trigger the bugs on the non-CHERI build, and the dark grey bars show the number of test cases that trigger the bugs on the CHERI build.

In these examples, the CHERI build produces almost twice as many triggering test cases overall. The largest gaps occur in paths that perform bounds, alignment, or pointer-derived access (e.g., array subscripts and aligned loads/stores). On CHERI, these operations fail fast at the exact misuse site (capability bounds or alignment checks), while the non-CHERI build only crashes if the corruption later propagates into illegal access.

Even when a bug is detectable on both platforms, the CHERI build consistently identifies it more frequently. For example, a bug in `set_aligned()` was triggered by 68 testcases on CHERI compared to only 13 on the non-CHERI build. This increased detection rate occurs because CHERI fails fast at the precise point of misuse, like a capability bounds or alignment check. In contrast, the non-CHERI build

Table 4. Raw crash triggers by category before deduplication. Counts are failing test executions, not unique defects.

Category	MicroPython v1.20 on x86	MicroPython v1.20 on ARM	MicroPython v1.20 on CHERI	MicroPython v1.27 preview
Raw memory, buffer protocol & view lifetime	222	201	238	230
Binary conversions & bigint corners	133	110	238	109
FFI / native emitters	111	0	0	81
Parsers, codecs & compressors in C	15	11	11	4
Filesystem, VFS & Race Conditions	175	166	17	38
MMIO & peripherals	0	0	0	0
Interpreter internals, exceptions & GC	17	17	32	29
Total	**673**	505	**536**	**491**

Table 5. The unique bug distribution after cleaning up the duplicates.

Target	bug-non libffi	bug-on libffi	Total
MicroPython v1.20 on x86	40	12	52
MicroPython v1.20 on ARMv8-32	39	8	47
MicroPython v1.20 on CHERI	50	0	50
MicroPython v1.27 preview	26	13	39

only crashes if and when the initial memory corruption later propagates to cause a segmentation fault.

5.4 Selected Bug Examples

MicroPython provides low-level primitives such as `uctypes`, the buffer protocol, and VFS block drivers that allow direct memory access in Python. In a conventional non-CHERI build, common bugs like out-of-bounds (OOB) accesses or the use of stale pointers often lead to silent memory corruption. The program may continue to execute, only to fail unpredictably at a later time, or in some cases, not fail at all, producing incorrect results. In contrast, CHERI enforces pointer bounds and permissions in hardware, providing robust spatial and temporal memory safety. When code attempts to use a pointer outside its designated range or after the memory it references has been deallocated (a stale pointer), CHERI immediately raises a precise fault at the point of the illegal access. This section presents concrete examples in MicroPython where CHERI reports bugs deterministically, while the non-CHERI Unix port typically does not.

5.4.1 Bug Example on CHERI-Enabled MicroPython.

The following examples demonstrate memory safety violations that are caught by the CHERI architecture.

Listing 4. Writing beyond the buffer via an overlong alias

```
owner = bytearray(b"hello")
base = uctypes.addressof(owner)
raw = uctypes.bytearray_at(base, len(owner)+8)
raw[len(owner):] = b"X"*8
```

In Listing 4, a raw memory alias is created that extends beyond the true boundary of the owner buffer. On the CHERI build, the first attempt to write past the end of the owner buffer raises a capability bounds fault, immediately terminating the illegal operation. On the non-CHERI build, these writes proceed silently, corrupting adjacent heap memory and allowing the program to continue in an undefined state.

Listing 5. Writing before the buffer via an underflow alias

```
owner = bytearray(16)
base  = u.addressof(owner)
a = u.bytearray_at(base - 1, 8)
a[0] = 1
```

Listing 5 demonstrates a spatial memory violation where an alias is created starting just before the allocated buffer. The CHERI build faults on the write to a[0], as it is outside the valid bounds of the owner capability. Conversely, the non-CHERI build executes this write, corrupting memory preceding the buffer, and the program continues, ignoring the error.

5.4.2 Bug Examples on Latest MicroPython.

This section shows four crash examples reproduced on the latest standard MicroPython Unix port. The first three reveal runtime defects in how MicroPython manages the lifetime of memoryview objects and handles reentrancy in list.sort(). The fourth example is a negative control, a Viper-compiled store to an unmapped memory address, which serves to validate our fault analysis methodology. All results in this section were obtained on a non-CHERI build.

Listing 6. Persistent writable crash after owner growth

```
ba = bytearray(b"abcdefghij")          # 10 bytes
old = uctypes.addressof(ba)
views = [memoryview(ba) for _ in range(4)]
```

```
ba[:] = ba + b"X"*256
new = uctypes.addressof(ba)
print("MOVED?", old != new, hex(old), "->", hex(new))
gc.collect()
for mv in views:
  mv[0:1] = b"Y"
```

The program typically prints MOVED? True, indicating the buffer was reallocated, and then crashes at a later, unrelated point in its execution. The backtrace often shows a fault during an indirect call within the virtual machine (e.g., in `mp_load_method_maybe`), which is consistent with the corruption of an object's type information or method table. MicroPython does not invalidate or prevent the use of existing writable *memoryview* objects when their underlying buffer is reallocated. When the *bytearray* grows, its storage moves, but the views retain a stale pointer to the old, now-freed memory location. Subsequent writes through these stale views corrupt whichever object has since been allocated in that memory space.

Listing 7. Typed array view kept across growth

```
path = "t.bin"
with open(path, "wb") as f:
  f.write(bytes(range(16)))
a = array('I', [0]*4)       # 4 * 4B = 16 bytes
old = uctypes.addressof(a)
mv  = memoryview(a)         # Writable view
a.extend([1]*2048)          # Likely moves storage
new = uctypes.addressof(a)
with open(path, "rb") as f:
  n = f.readinto(mv)      # OS writes into stale address
```

A crash typically occurs when the program exits the with block. The backtrace points to a fault within *mp_stream_close()* while trying to read the type field of what it assumes is a valid stream object, indicating that the object's metadata has been corrupted. A writable *memoryview* of a typed array persists after the array's storage is moved. This stale view is then used as the destination buffer for *readinto*, which writes data to the deallocated memory region, corrupting heap objects that now occupy that space.

Listing 8. Comparator clears the list during sort operation

```
def lt(self, other):
  self.arr_ref.clear()  # Mutates the container being
     sorted
  gc.collect()
  return True
a = []
a.extend(Evil(a) for _ in range(5))
a.sort()
```

The virtual machine crashes with a segmentation fault inside mp_binary_op() while executing the quicksort algorithm. The GDB backtrace reveals that the program attempts to dereference a NULL pointer (rdi == 0) when trying to retrieve an object's type, indicating that the sort algorithm is operating on an invalid list element. The list.sort() implementation is not reentrant-safe. It does not protect against scenarios where the comparison function (__lt__) modifies the list being sorted. In this example, the comparator clears the list, deallocating its elements. The sort function, unaware of this change, proceeds to use its now-invalidated pointers to these elements, resulting in a NULL pointer dereference.

According to the MicroPython documentation [14], the `uctypes`, `ffi`, and `_thread` modules are explicitly described as low-level, unsafe interfaces intended for expert use. Many of the behaviours we demonstrate, therefore, fall within their documented risk model rather than constituting security bugs. For this reason, we only reported three issues to the upstream MicroPython repository: those that impact core semantics or safety guarantees beyond these explicitly unsafe APIs[5][6][7].

6 Related Work

Capability-based hardware and CHERI provide the architectural foundation for treating memory safety as a property enforced by the machine rather than only by software instrumentation. The foundational CHERI paper introduced tagged, bounded, provenance-preserving capabilities as a scalable basis for fine-grained memory protection and software compartmentalisation [42], while a recent survey emphasizes that CHERI is aimed at memory safety for large existing C/C++ codebases and explicitly notes that language runtimes remain important attack surfaces because the runtimes themselves are typically implemented in unsafe languages [41]. CheriABI and the wider CheriBSD effort then show that these ideas scale into a POSIX/FreeBSD-style runtime environment, making CHERI relevant not only as an ISA design but as a practical systems platform.

Morello MicroPython reports one of the first concrete experiences of adapting a scripting-language runtime to CHERI, focusing on source changes, runtime costs, and pragmatic bug detection in a MicroPython port [21]. Secure Scripting with CHERIoT MicroPython extends this direction by considering secure scripting atop a CHERI-derived embedded platform [22]. These works are primarily focuse on porting and evaluation, they do not centre on a differential fuzzing methodology, nor do they make a systematic comparison between CHERI-backed enforcement and software-only hardening as bug-finding oracles.

McKeeman's classic formulation of differential testing motivates the use of multiple executions as an oracle when conventional specifications are weak or incomplete [23]. In virtual machines, Chen et al. introduced coverage-directed differential testing of JVM implementations, using mutation

[5] https://github.com/micropython/micropython/issues/18171
[6] https://github.com/micropython/micropython/issues/18170
[7] https://github.com/micropython/micropython/issues/17941

and coverage uniqueness to generate representative class-files [10], and later extended that line with deep differential testing via live bytecode mutation [32]. These are strong evidence for interpreter and VM testing, but they mainly seek semantic inconsistencies across implementations, they do not exploit hardware-enforced capability violations as the differential signal.

AFL [49] and libFuzzer [33] remain the practical baseline coverage-guided evolutionary engines in widespread use, and both embody the coverage-driven mutation loop that motivates modern fuzzing practice. Interpreter-specific work then argues that generic byte-level mutation is insufficient: IFuzzer applies genetic programming to generate syntactically valid, uncommon programs for JavaScript interpreters [38], Superion and NAUTILUS combine grammar awareness with coverage guidance to reach deeper states in programs that consume structured inputs [3, 39]. These works establish why grammar-aware and evolutionary generation matters for interpreter targets, but they remain software-only and generally rely on crashes, semantic divergence, or JIT-specific behaviour rather than CHERI faults.

Finally, software-only hardening and sandboxing provide the most natural comparison baseline. SoftBound/CETS and AddressSanitizer are representative instrumentation-based approaches for spatial, temporal, or general memory-error detection [27, 34], while Native Client exemplifies sandboxing through software fault isolation rather than capability-aware pointer semantics [48]. Our work is not simply fuzzing interpreters or porting an interpreter to CHERI, but integrating CHERI-backed enforcement with differential, structured-input fuzzing so that hardware capability checks become part of the testing oracle, and then using that oracle to compare what CHERI exposes against software-only approaches.

7 Threats to Validity

Following best-practice guidelines for fuzzing studies [31], we consider several threats to validity.

For *internal validity*, the limited diversity of the seed corpus may introduce bias, since it is derived from publicly disclosed bugs in CPython and MicroPython. We are not aware of clear evidence of such bias. To reduce this threat, we collected bugs over a long time span, covering MicroPython issues since its public release in 2015 and CPython issues since v3.0.0 in 2008. This gives the seed corpus a broad historical base, but it does not guarantee that all important classes of MicroPython bugs are represented.

Another internal threat is root-cause ambiguity. Our technique detects observable failures, but a failure does not by itself establish whether the root cause lies in the MicroPython runtime, in an underlying C library, in the CHERI port, or in client code that misuses an explicitly unsafe interface. This issue is particularly important for modules such as `uctypes`, `ffi`, and other low-level APIs, where the boundary between an implementation defect and invalid client behaviour can be subtle. We therefore treat each failure as evidence of a memory-safety-relevant behaviour, and separate automatic crash clustering from manual root-cause classification.

For *construct validity*, our main oracle is based on crashes, timeouts, and CHERI capability faults. Crashes and timeouts are standard signals in fuzzing studies, but they are incomplete proxies for correctness. Some bugs may only appear as silent data corruption, wrong output, resource leaks, or semantic divergence between interpreter variants. A more fine-grained oracle would compare program outputs across interpreter builds and classify behavioural differences even when neither build crashes. We leave this as future work.

There is also a construct validity threat in treating CHERI faults as a memory safety oracle. CHERI can expose violations involving capability bounds, permissions, tags, pointer provenance, and some alignment errors. These signals are valuable because they often turn latent memory corruption into a precise fault at the point of misuse. However, CHERI is not a complete detector for all memory errors. It may miss intra-object corruption, uninitialised reads, semantic misuse of an API, logical bugs, and some temporal errors when allocator support or capability revocation is absent. Therefore, our results should be interpreted as measuring the class of memory-safety failures that become observable under CHERI enforcement, rather than as a complete account of all possible bugs in MicroPython.

The use of an LLM introduces an additional evaluation threat. The LLM is not part of the tight fuzzing loop. Instead, it is used in a slower outer loop to score and select prompts that are more likely to generate useful test programs. The selected tests are then expanded by cheaper LibCST-based mutation. This design is important for performance and reproducibility: LLM-based generation has much lower throughput than standard mutation-based fuzzing, producing on the order of hundreds of generated tests per hour rather than thousands of cheap mutants per hour. Its benefit is not raw execution speed, but a higher rate of syntactically valid and semantically plausible programs. We therefore report LLM generation cost separately from mutation and execution cost, and do not claim that LLM generation replaces high-throughput fuzzing.

For *external validity*, we expect our approach to extend to other C-based language interpreters ported to CHERI platforms, because the framework does not rely on Python-specific memory-safety assumptions. One possible target is CRuby [19]. Nevertheless, the concrete bug patterns, unsafe APIs, and runtime behaviours studied here are specific to MicroPython. Other interpreters may expose different memory-management policies, extension interfaces, object layouts, and garbage-collection designs.

8 Conclusion

We have presented diffCHERI:FruitFly, a differential testing framework for evaluating memory safety in the MicroPython interpreter. diffCHERI:FruitFly combines targeted test generation with hardware-assisted checking to move beyond blind mutation. It uses an LLM-based generator and a LibCST mutator to produce semantically valid, high-risk Python programs that naive fuzzers often fail to generate. Each program is then executed on both a standard build and a CHERI-enabled build under a unified harness that captures and normalises behaviour. Treating the CHERI-enabled interpreter as a variant implementation turns behavioural divergence into an oracle for memory-safety faults. In particular, CHERI traps make out-of-bounds accesses, use-after-free errors, and related violations visible as deterministic failures. Our evaluation shows that this approach exposes memory-safety bugs that remain undetected in the non-CHERI build. It also uncovered 39 unique bugs in MicroPython v1.27 preview. We have responsibly disclosed these bugs to the MicroPython developers and release the framework and bug database through the project repository at https://github.com/MaksimFeng/ML4Secure.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable feedback. This work was funded by EPSRC under grant numbers EP/X037304/1 and EP/X037525/1.

References

[1] Saar Amar, Tony Chen, David Chisnall, Nathaniel Wesley Filardo, Ben Laurie, Hugo Lefeuvre, Kunyan Liu, Simon W Moore, Robert Norton-Wright, Margo Seltzer, and Robert N. M. Watson. 2025. CHERIoT RTOS: An OS for Fine-Grained Memory-Safe Compartments on Low-Cost Embedded Devices. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, 67–84. doi:10.1145/3731569.3764844

[2] Saar Amar, David Chisnall, Tony Chen, Nathaniel Wesley Filardo, Ben Laurie, Kunyan Liu, Robert Norton, Simon W Moore, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. 2023. CHERIoT: Complete memory safety for embedded devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, New York, NY, USA, 641–653. doi:10.1145/3613424.3614266

[3] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for deep bugs with grammars. In *NDSS*, Vol. 19. 337. doi:10.14722/ndss.2019.23412

[4] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering* 41, 5 (2014), 507–525. doi:10.1109/TSE.2014.2372785

[5] Michael Barr and Anthony Massa. 2006. *Programming embedded systems: with C and GNU development tools*. O'Reilly Media, Sebastopol, CA, USA. doi:doi/10.5555/1211083

[6] Charles Bell. 2017. *MicroPython for the Internet of Things*. Apress, Berkeley, CA, USA.

[7] Andrew R Bernat and Barton P Miller. 2011. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*. Association for Computing Machinery, New York, NY, USA, 9–16. doi:10.1145/2024569.2024572

[8] Marco Brohet and Francesco Regazzoni. 2023. A survey on thwarting memory corruption in RISC-V. *Comput. Surveys* 56, 2 (2023), 1–29. doi:10.1145/3604906

[9] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, IEEE, Piscataway, NJ, USA, 1257–1268. doi:10.1109/ICSE.2019.00127

[10] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 85–99. doi:10.1145/2908080.2908095

[11] Andrew G Clark, Neil Walkinshaw, and Robert M Hierons. 2021. Test case generation for agent-based models: A systematic literature review. *Information and Software Technology* 135 (2021), 106567. doi:10.1016/j.infsof.2021.106567

[12] Roland Croft, Yongzheng Xie, Mansooreh Zahedi, M Ali Babar, and Christoph Treude. 2022. An empirical study of developers' discussions about security challenges of different programming languages. *Empirical Software Engineering* 27, 1 (2022), 27. doi:10.1007/s10664-021-10054-w

[13] Gabriel Gaspar, Peter Fabo, Michal Kuba, Jana Flochova, Juraj Dudak, and Zuzana Florkova. 2020. Development of IoT applications based on the MicroPython platform for Industry 4.0 implementation. In *2020 19th International conference on mechatronics-mechatronika (ME)*. IEEE, IEEE, Piscataway, NJ, USA, 1–7. doi:10.1109/ME49197.2020.9286455

[14] Damien P George and Paul Sokolovsky. 2025. MicroPython documentation. https://docs.micropython.org/.

[15] Richard Grisenthwaite, Graeme Barnes, Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Jonathan Woodruff. 2023. The Arm Morello Evaluation Platform—Validating CHERI-Based Security in a High-Performance System. *IEEE Micro* 43, 3 (2023), 50–57. doi:10.1109/MM.2023.3264676

[16] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. 2023. FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities. Network and Distributed System Security Symposium (NDSS). https://www.ndss-symposium.org/wp-content/uploads/2023-290-paper.pdf.

[17] Brett Gutstein. 2022. *Memory safety with CHERI capabilities: security analysis, language interpreters, and heap temporal safety*. Technical Report UCAM-CL-TR-975. University of Cambridge, Computer Laboratory. doi:10.48456/tr-975

[18] Lu Liang, Yong Li, Ming Wen, and Ying Liu. 2022. KG4Py: A toolkit for generating Python knowledge graph and code semantic search. *Connection Science* 34, 1 (2022), 1384–1400. doi:10.1080/09540091.2022.2072471

[19] Hanhaotian Liu, Tetsuro Yamazaki, and Tomoharu Ugawa. 2026. Pitfalls in VM Implementation on CHERI: Lessons from Porting CRuby. *The Art, Science, and Engineering of Programming* 11, 1 (2026). doi:10.22152/programming-journal.org/2026/11/2

[20] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–25. doi:10.1145/3428264

[21] Duncan Lowther, Dejice Jacob, and Jeremy Singer. 2023. Morello MicroPython: a Python interpreter for CHERI. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. Association for Computing Machinery, New York, NY, USA, 62–69. doi:10.1145/3617651.3622991

[22] Duncan Lowther, Dejice Jacob, Jacob Trevor, and Jeremy Singer. 2025. Secure Scripting with CHERIoT MicroPython. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*.

Association for Computing Machinery, New York, NY, USA, 180–191. doi:10.1145/3708493.3712694

[23] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[24] Meta Platforms, Inc. 2025. LibCST. https://github.com/Instagram/LibCST.

[25] Matt Miller. 2019. Trends and challenges in the vulnerability mitigation landscape. 13th USENIX Workshop on Offensive Technologies (WOOT 19).

[26] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. 2018. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. Network and Distributed System Security Symposium (NDSS). https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_01A-4_Muench_paper.pdf.

[27] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 245–258. doi:10.1145/1542476.1542504

[28] Alexander Novikov, Ngân V u, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. 2025. AlphaEvolve: A coding agent for scientific and algorithmic discovery. arXiv:2506.13131 [cs.AI] https://arxiv.org/abs/2506.13131

[29] Ignas Plauska, Agnius Liutkevičius, and Audronė Janavičiūtė. 2022. Performance evaluation of C/C++, MicroPython, Rust and TinyGo programming languages on ESP32 microcontroller. *Electronics* 12, 1 (2022), 143. doi:10.3390/electronics12010143

[30] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. 2021. Token-Level Fuzzing. In *30th USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 2795–2809. https://www.usenix.org/system/files/sec21-salls.pdf.

[31] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. 2024. SoK: Prudent evaluation practices for fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, IEEE, Piscataway, NJ, USA, 1974–1993. doi:10.1109/SP54263.2024.00137

[32] Arshdeep Sekhon, Ritambhara Singh, and Yanjun Qi. 2018. DeepDiff: DEEP-learning for predicting DIFFerential gene expression from histone modifications. *Bioinformatics* 34, 17 (2018), i891–i900. doi:10.1093/bioinformatics/bty612

[33] Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 157–157. doi:10.1109/SecDev.2016.043

[34] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*. 309–318. doi:doi/10.1145/3719027.3744861

[35] Mayank Sharma, Pingshi Yu, and Alastair F Donaldson. 2023. Rustsmith: Random differential compiler testing for Rust. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Association for Computing Machinery, New York, NY, USA, 1483–1486. doi:10.1145/3597926.3604919

[36] Xiaoyang Sun, Jeremy Singer, and Zheng Wang. 2025. Sweet or Sour CHERI: Performance Characterization of the Arm Morello Platform. In *2025 IEEE International Symposium on Workload Characterization*. IEEE, Piscataway, NJ, USA, 423–438. doi:10.1109/IISWC66894.2025.00042

[37] Nicholas H Tollervey. 2017. *Programming with MicroPython: embedded programming with microcontrollers and Python*. O'Reilly Media, Sebastopol, CA, USA.

[38] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security*. Springer, 581–601. doi:10.1007/978-3-319-45744-4_29

[39] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 724–735. doi:10.1109/ICSE.2019.00081

[40] Qian Wang and Ralf Jung. 2024. Rustlantis: Randomized differential testing of the Rust compiler. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 1955–1981. doi:10.1145/3689780

[41] Robert NM Watson, David Chisnall, Jessica Clarke, Brooks Davis, Nathaniel Wesley Filardo, Ben Laurie, Simon W Moore, Peter G Neumann, Alexander Richardson, Peter Sewell, et al. 2024. CHERI: Hardware-enabled C/C++ memory protection at scale. *IEEE Security & Privacy* 22, 4 (2024), 50–61. doi:10.17863/CAM.107230

[42] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. 2015. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*. IEEE, Piscataway, NJ, USA, 20–37. doi:10.1109/SP.2015.9

[43] Robert N. M. Watson, Jessica Clarke, Peter Sewell, Jonathan Woodruff, Simon W. Moore, Graeme Barnes, Richard Grisenthwaite, Kathryn Stacer, Silviu Baranga, and Alexander Richardson. 2023. *Early performance results from the prototype Morello microarchitecture*. Technical Report UCAM-CL-TR-986. University of Cambridge, Computer Laboratory. doi:10.48456/tr-986

[44] Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Peter G. Neumann. 2019. *An Introduction to CHERI*. Technical Report UCAM-CL-TR-941. University of Cambridge, Computer Laboratory. doi:10.48456/tr-941

[45] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Nathaniel Wesley Filardo, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. 2019. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7)*. Technical Report UCAM-CL-TR-927. University of Cambridge, Computer Laboratory. doi:10.48456/tr-927

[46] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: revisiting RISC in an age of risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecture*. Association for Computing Machinery, New York, NY, USA, 457–468. doi:10.1145/2678373.2665740

[47] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. Association for Computing Machinery, New York, NY, USA, 283–294. doi:10.1145/1993498.1993532

[48] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2010. Native client: A sandbox for portable, untrusted x86 native code. *Commun. ACM* 53, 1 (2010), 91–99. doi:10.1109/CGO68049.2026.11395235

[49] Michal Zalewski. 2017. American Fuzzy Lop. [Online]. Available: https://lcamtuf.coredump.cx/afl/.

Received 2026-04-03; accepted 2026-05-04