2026.03.04

Toward Practical Vulnerability Repair

Zheng Yu

Committee: Xinyu Xing, Peter Dinda, Yan Chen, Kexin Pei

Northwestern University

40,303
CVE reported in 2024

6x
Increase since 2015

3,500+
Vulnerabilities Still Open

Thousands of Vulnerabilities Awaiting Repair

PHP

php/php-src

260 open verified issues

CPython

python/cpython

135 open crash issues

LLVM

llvm/llvm-project

1925 open crash issues

Linux

syzbot/upstream

1447 open vulnerabilities

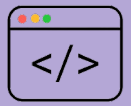
[1] https://github.com/php/php-src/issues?q=is%3Aissue%20state%3Aopen%20label%3A%22Status%3A%20Verified%22
[2] https://github.com/python/cpython/issues?q=is%3Aissue%20state%3Aopen%20label%3Atype-crash
[3] https://github.com/llvm/llvm-project/issues?q=is%3Aissue%20state%3Aopen%20label%3Acrash
[4] https://syzkaller.appspot.com/upstream

Table of contents

Automated Vulnerability Repair (AVR)

Fault Localization

FL aims to identify the root cause and to provide code locations to apply patches.



Patch Generation

Takes both the buggy code snippet and bug description as input, then produces a patch.

Patch Validation

Verify that a patch addresses vulnerabilities while maintaining functional integrity.

Perfect Fault Localization Input

Use Perfect FL As Input

```
if (tp_len(ctx, p1) ⩾ cnt &&
  tp_len(ctx, p) ⩾ pos + cnt)
{
  memcpy(p1→ptr,
         p→ptr + (pos << oft),
         cnt << oft);
} else {
  for (n = 0; n < cnt; n++) {
```

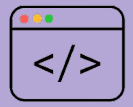
Vulnerable Function

```
if (tp_len(ctx, p1) ⩾ cnt &&
  tp_len(ctx, p) ⩾ pos + cnt)
{
  memcpy(p1→ptr,
         p→ptr + (pos << oft),
         cnt << oft);
} else {
  for (n = 0; n < cnt; n++) {
```

Mask Vulnerable Line

```
if (tp_len(ctx, p1) ⩾ cnt &&
  tp_len(ctx, p) ⩾ pos + cnt)
{
  memcpy(p1→ptr,
         p→ptr + (pos << oft),
         cnt << oft);
} else {
  for (n = 0; n < cnt; n++) {
```

Mask Vulnerable Token



Patch Generation

Takes both the buggy code snippet and bug description as input, then produces a patch.

Patch Validation

Verify that a patch addresses vulnerabilities while maintaining functional integrity.

PatchAgent

A Practical Program Repair Agent Mimicking Human Expertise

(USENIX Security 2025)

Global Buffer Overflow

The bug occurs because GetOpInfo performs pointer arithmetic on a global operation table without validating the opcode. An out-of-range opcode leads to out-of-bounds access on the global array, resulting in a global buffer overflow.

```
==35==ERROR: AddressSanitizer: global-buffer-overflow
READ of size 8 at 0x55bc18 thread T0
    #0 0x55969b in Compile_BlockStat /source/m3_compile.c:22
    #1 0x55c4c6 in Compile_Block /source/m3_compile.c:277
    #2 0x55cbc3 in Compile_If /source/m3_compile.c:1648
    #3 0x5596ec in Compile_BlockStatement /source/m3_compile.c:207
    #4 0x55ca29 in Parse_InitExpr /source/m3_parse.c:282
    ......
    #8 0x55d715 in LLVMFuzzerTestOneInput /app_fuzz/fuzzer.c:30
    #9 0x552e14 in fuzzer::Fuzzer::ExecuteCallback
    ......
    0x55c18 is located 8 bytes after global variable c_operations

    SUMMARY: AddressSanitizer: global-buffer-overflow
```

```c
const M3OpInfo c_operations[] = { /* ... */ };
const M3OpInfo c_operationsFC[] = { /* ... */ };

static inline const M3OpInfo*
GetOpInfo(m3opcode_t opcode) {
    switch (opcode >> 8) {
    case 0x00:
        return &c_operations[opcode];
    case 0xFC:
        return &c_operationsFC[opcode & 0xFF];
    default:
        return NULL;
    }
}

M3Result
Compile_BlockStat(IM3Compilation o) {
    m3opcode_t opcode;
    Read_opcode (&opcode, &o);
    IM3OpInfo opinfo = GetOpInfo(opcode);
    _throwif(unknownOpcode, opinfo == NULL);
    if (opinfo->compiler) { // global overflow
        (*opinfo->compiler) (o, opcode)
    } else {
        Compile_Operator (o, opcode);
    }
}
```

Human Expert *vs* Vanilla Agent

Repair Comparison

We provide the same three capabilities to both humans and a vanilla agent, and compare their repair processes to evaluate differences in performance.

View Code

Retrieve Code by Specifying Ranges

Find Definition

Find definition location by providing reference location

Validate

Replay PoC & Run Functional Test

Human Expert

```
==35==ERROR: AddressSanitizer: global-buffer-overflow
READ of size 8 at 0x55bc18 thread T0
    #0 0x55969b in Compile_BlockStat /source/m3_compile.c:22
    #1 0x55c4c6 in Compile_Block /source/m3_compile.c:277
    #2 0x55cbc3 in Compile_If /source/m3_compile.c:1648
    #3 0x5596ec in Compile_BlockStatement /source/m3_compile.c:207
    #4 0x55ca29 in Parse_InitExpr /source/m3_parse.c:282
    ......
    #8 0x55d715 in LLVMFuzzerTestOneInput /app_fuzz/fuzzer.c:30
    #9 0x552e14 in fuzzer::Fuzzer::ExecuteCallback
    ......
    0x55c18 is located 8 bytes after global variable c_operations

    SUMMARY: AddressSanitizer: global-buffer-overflow
```

```
 GetOpInfo(m3opcode_t opcode) {
    switch (opcode >> 8) {
    case 0x00:
-       return &c_operations[opcode];
+       if (opcode < sizeof(c_operations)/sizeof(c_operations[0]))
+           return &c_operations[opcode];
+       break;
    case 0xFC:
-       return &c_operationsFC[opcode & 0xFF];
+       opcode &= 0xFF;
+       if (opcode < sizeof(c_operationsFC)/sizeof(c_operationsFC[0]))
+           return &c_operationsFC[opcode];
+       break;
    default:
        return NULL;
    }
```

According to the address sanitizer report, a global overflow occurred at line 22 in the /source/m3_compile.c. Let me view lines 17 to 27 in /source/m3_compile.c.

....
IM3OpInfo opinfo = GetOpInfo(opcode);
_throwif(unknownOpcode, opinfo == NULL);
if (opinfo->compiler) {
  (*opinfo->compiler) (o, opcode)
....

`opinfo` is an overflow pointer generated by GetOpInfo, I would like to locate its definition.

The variable GetOpInfo is defined at line 5 in the /source/m3_compile.c.

Great, now I want to view lines 5 to 16 in /source/m3_compile.c

GetOpInfo(m3opcode_t opcode) {
  switch (opcode >> 8) {
  case 0x00:
    return &c_operations[opcode]; ....

Vanilla Agent

Too small code range

```
==35==ERROR: AddressSanitizer: global-buffer-overflow
READ of size 8 at 0x55bc18 thread T0
    #0 0x55969b in Compile_BlockStat /source/m3_compile.c:22
    #1 0x55c4c6 in Compile_Block /source/m3_compile.c:277
    #2 0x55cbc3 in Compile_If /source/m3_compile.c:1648
    #3 0x5596ec in Compile_BlockStatement /source/m3_compile.c:207
    #4 0x55ca29 in Parse_InitExpr /source/m3_parse.c:282
    ......
    #8 0x55d715 in LLVMFuzzerTestOneInput /app_fuzz/fuzzer.c:30
    #9 0x552e14 in fuzzer::Fuzzer::ExecuteCallback
    ......
    0x55c18 is located 8 bytes after global variable c_operations

    SUMMARY: AddressSanitizer: global-buffer-overflow
```

```
M3Result Compile_BlockStat(IM3Compilation o) {
    m3opcode_t opcode;
    Read_opcode (&opcode, &o);
+   if (opcode < 0 || opcode > MAX_OPCODE)
+       return unnkownOpcode;
    IM3OpInfo opinfo = GetOpInfo(opcode);
    _throwif(unknownOpcode, opinfo == NULL);
    if (opinfo→compiler) { // global overflow
```

According to the address sanitizer report, a global overflow occurred at line 22 in the /source/m3_compile.c. So I want to view lines 21 to 23 in /source/m3_compile.c.

```
_throwif(unknownOpcode, opinfo == NULL);
if (opinfo->compiler) {
  (*opinfo->compiler) (o, opcode)
```

It appears that the overflow occurs in opinfo. I would like to locate the definition of opinfo.

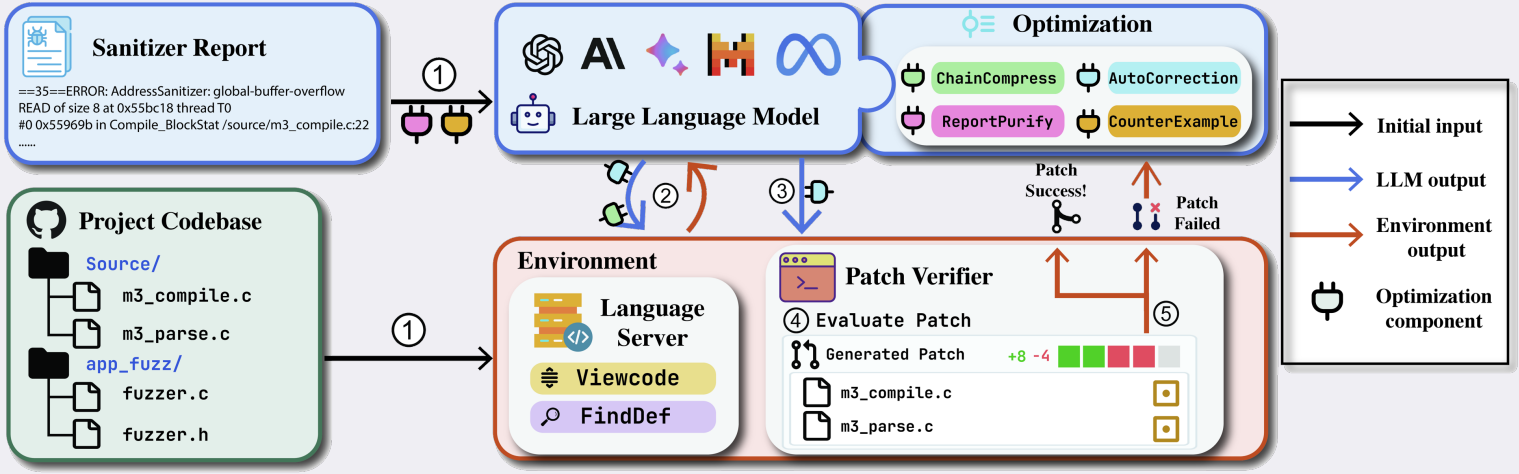
The variable opinfo is defined at line 20 in the /source/m3_compile.c.

Great, now I want to view lines 17 to 20 in /source/m3_compile.c

m3opcode_t opcode;
read_opcode(&opcode,&o);
IM3OpInfo opinfo = GetOpInfo(opcode);

Why not continue exploring related function?

PatchAgent Overview



The process begins with the **sanitizer report** and the **project codebase** (1). The LLM retrieves the code context using the **viewcode** and **find_definition** APIs (2) and then **generates a patch** (3). The patch is subsequently validated by the patch verifier (4). If the patch is incorrect, the agent will refine the patch or **gather additional context** (5), iterating until a correct patch is generated or the budget is exhausted.

Auto Correction (Disabled)

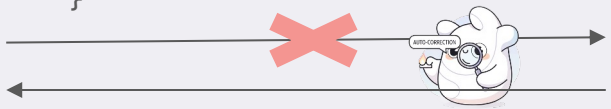
```
{
    "action": "viewcode",
    "path": "src/m3_compile.c",
    "start_line": 21,
    "end_line": 23
}
```

Language Server

```
21| _throwif(unknownOpcode, opinfo == NULL);
22| if (opinfo->compiler) {
23|     (*opinfo->compiler) (o, opcode)
```

Auto Correction (Enabled)

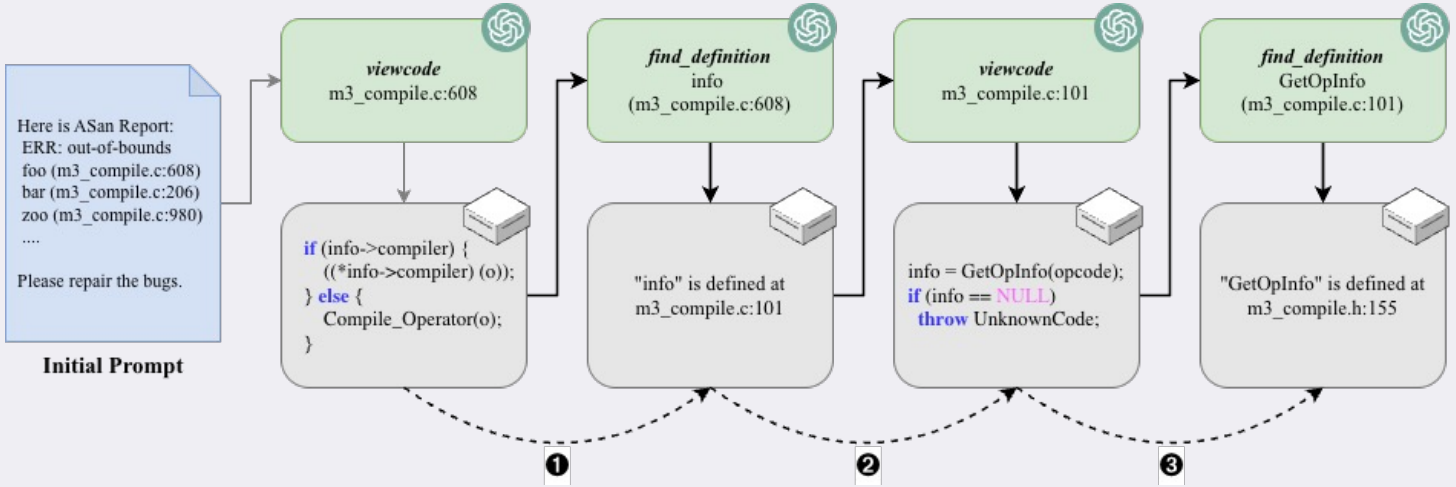
```
{
    “action”: “viewcode”,
    “path”: “src/m3_compile.c”,
    “start_line”: 17,
    “end_line”: 27
}
```



Language Server

```
17| Compile_BlockStat(IM3Compilation o) {
18|     m3opcode_t opcode;
19|     Read_opcode (&opcode, &o);
20|     IM3OpInfo opinfo = GetOpInfo(opcode);
21|     _throwif(unknownOpcode, opinfo == NULL);
22|     if (opinfo->compiler) {
23|         (*opinfo->compiler) (o, opcode)
24|     } else {
25|         Compile_Operator (o, opcode);
26|     }
27| }
```

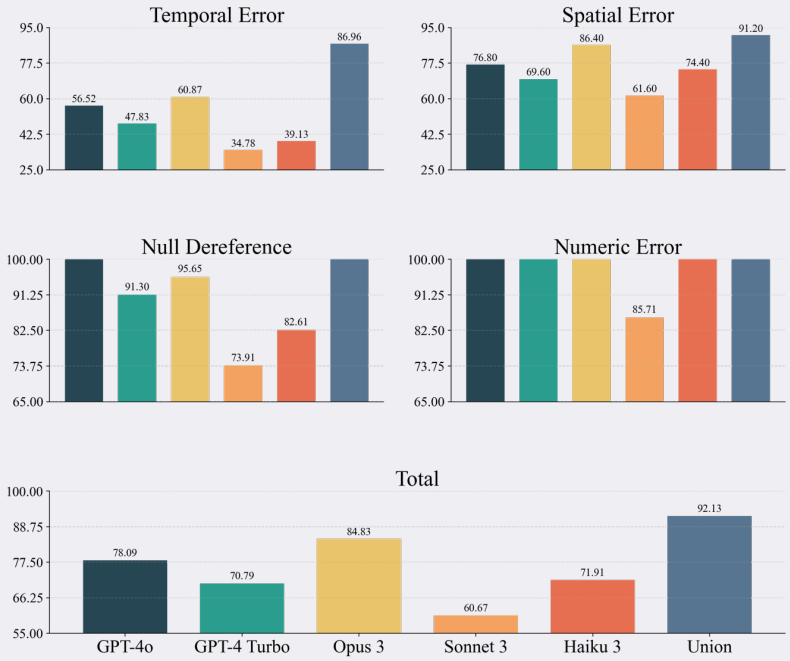
Chain Compression



The LLM initially requires four sequential LSP actions (viewcode → find_definition → viewcode → find_definition) to trace the crash-causing symbol **info** and its dependency **GetOpInfo**. Our mechanism identifies crash-relevant symbols, expands their definitions, and recursively resolves dependent symbols in a single semantic step, compressing the interaction chain from length 4 to 1 while reducing latency and token overhead.

Evaluation

We created a dataset of 178 vulnerabilities, covering 9 bug types including stack/heap overflow, UAF, and null dereference. Evaluating PatchAgent with five different LLMs, individual models achieved 60.67%–84.83% repair accuracy, while the Union approach combining all models reached 92.13%, demonstrating that collaborative multi-model usage significantly improves AVR.



32 Merged

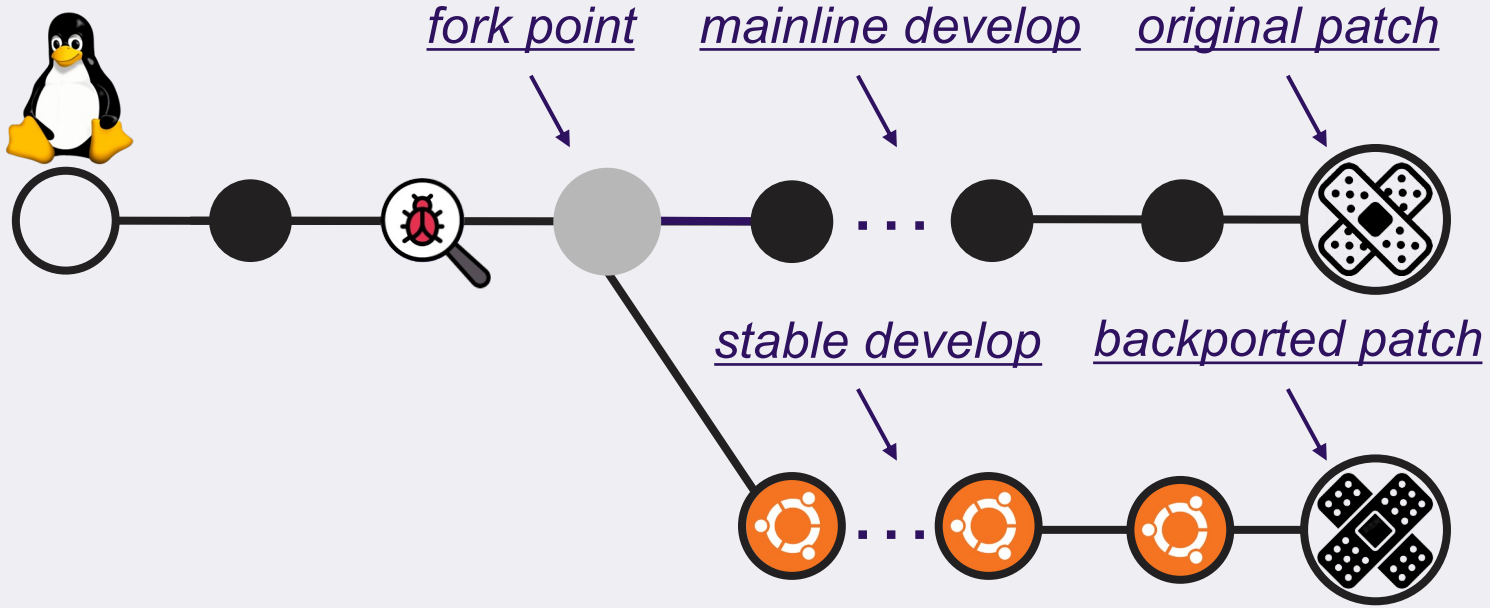
Including projects like assimp, libssh2, hdf5, libredwg, pcapplusplus, yasm, linux

PortGPT

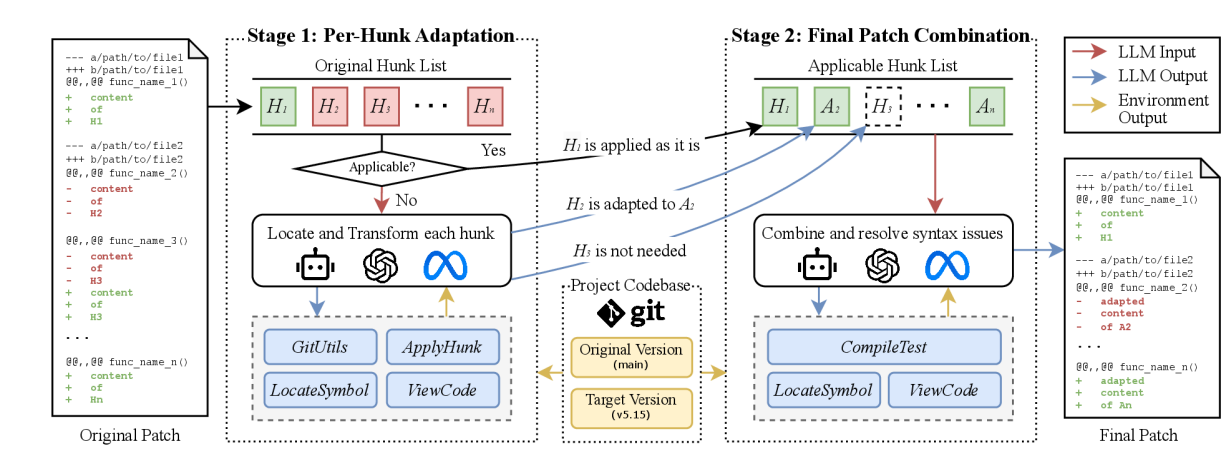
Towards Automated Backporting Using Large Language Models

(IEEE Security & Privacy 2026)

Backporting Background



PortGPT Workflow

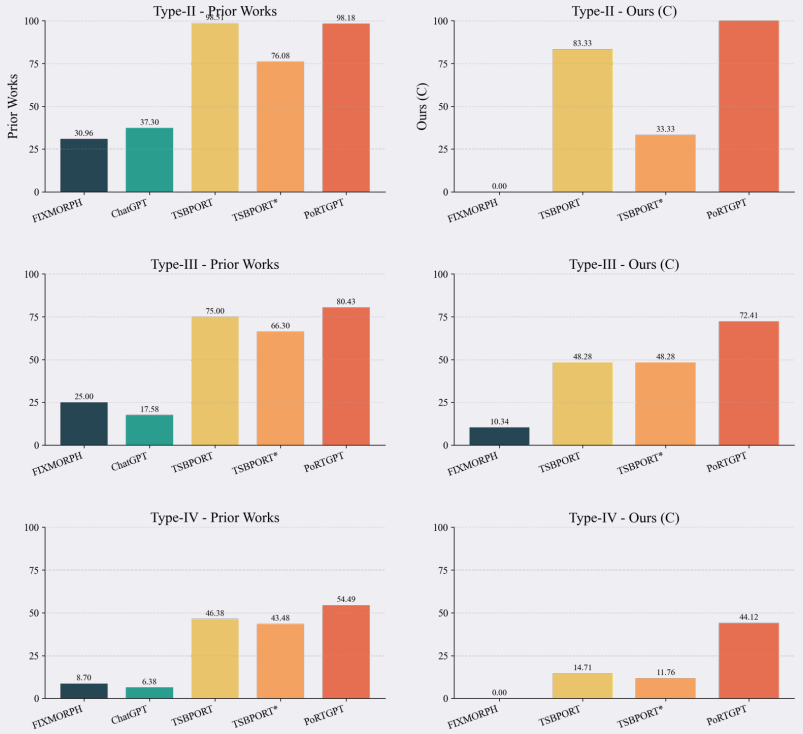


In **Stage 1 (Per-Hunk Adaptation)**, the system extracts code hunks from the original patch and processes each individually, determining whether backporting is required and leveraging LLMs with target codebase context to perform transformations for structural correctness.

In **Stage 2 (Final Patch Combination)**, transformed hunks are merged into a complete patch and applied to the target version for compilation validation. Upon failure, PortGPT iteratively resolves issues by adding necessary definitions or adjusting code context to produce a syntactically compilable backported patch.

Evaluation

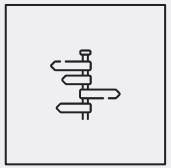
- **Performance Highlights:** PortGPT consistently outperforms all baselines across datasets. For simple scenarios (Type-I/II), PortGPT achieves perfect success rates. For complex scenarios (Type-III/IV), PortGPT outperforms prior approaches.
- **Cross-Language:** PortGPT attains the highest overall success rate on standard benchmarks and demonstrates strong generalization to cross-language settings (C, C++, and Go), significantly exceeding existing tools in all categories.
- **Real World Impact:** 9 backported patches have been merged into Linux 6.1



Patch Validation

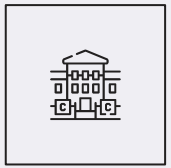
Patch Validation in Automated Vulnerability Repair

Findings from GitHub Pull Request



Functional Issues

Fixes focus on making the program work correctly under specific conditions, but do not necessarily address deeper problems.



Security Issues

Fixes mitigate a specific vulnerability instance,
yet may fail to ensure complete memory-safety guarantees.

Incorrect Root Cause

Patches treat visible symptoms of a crash,
while the underlying cause remains unresolved.

Passing tests does not imply correct repair.

“Golden?” Patch Validation

Test-suite based validation is often treated as a “gold standard”.

Fault Localization

FL aims to identify the root cause and to provide code locations to apply patches.

Patch Generation

Takes both the buggy code snippet and bug description as input, then produces a patch.

Works Using Test Suite-Based Validation

- **CrashRepair** (TOESM 2025)
- **CPR** (PLDI 2021)
- **Fix2Fit** (ISSTA 2019)
- **Zero-Shot** (S&P 2023)
- **San2Patch** (USENIX Sec 2025)
- **VulnFix** (ISSTA 2022)
- ……

Observations from a PHP Case

1) Existing functional tests (e.g., CI) do not capture full functionality.

2) Developers may upgrade functional tests after vulnerability repair.

Patch

```
--- a/ext/standard/array.c
+++ b/ext/standard/array.c
@@ -2924,8 +2924,8 @@ PHP_FUNCTION(range)

   /* If the range is given as strings, generate an array of characters. */
   if (start_type ⩾ IS_STRING || end_type ⩾ IS_STRING) {
-    /* If one of the inputs is NOT a string */
-    if (UNEXPECTED(start_type + end_type < 2*IS_STRING)) {
+    /* If one of the inputs is NOT a string nor single-byte string */
+    if (UNEXPECTED(start_type < IS_STRING || end_type < IS_STRING)) {
       if (start_type < IS_STRING) {
         if (end_type ≠ IS_ARRAY) {
           php_error_docref(NULL, E_WARNING, "Arg #1 ($start) must ……"
```

New Testcase

```
--- /dev/null
+++ b/ext/standard/tests/array/range/gh13094.phpt
@@ -0,0 +1,29 @@
+--TEST--
+GH-13094 (range(9.9, '0') causes segmentation fault)
+--FILE--
+<?php
+var_dump(range(9.9, '0'));
+?>
+--EXPECT--
+array(10) {
+  [0]⇒
+  float(9.9)
+  [1]⇒
+  float(8.9)
+  [2]⇒
+  float(7.9)
+  ……
```

[1] https://github.com/php/php-src/commit/1d6f344bea49ccad82b9a95a80ed9fdc39e260a1

Vulnerability Principle

range(1, 5) ⟶ 1, 2, 3, 4
range(1, 10, 2) ⟶ 1, 3, 5, 7, 9

The code first checks if either argument is string-like using start_type >= IS_STRING || end_type >= IS_STRING. However, the subsequent validation start_type + end_type < 2*IS_STRING fails to properly account for type combinations where the sum equals exactly 12. In these cases, the condition evaluates to false despite neither argument being a valid string, causing execution to skip the numeric handling path and leads the code to invoke Z_STRVAL_P on non-string zvals, resulting in invalid memory access.

```
PHP_FUNCTION(range) {
  struct zval *user_start = /* Extract start argument */;
  struct zval *user_end = /* Extract end argument */;
  // Extract type from arguments
  uint8_t start_type = Z_TYPE_P(user_start);
  uint8_t end_type = Z_TYPE_P(user_end);
  /* If the range is given as strings,
     generate an array of characters. */
  if (start_type ⩾ IS_STRING || end_type ⩾ IS_STRING) {
    // VULNERABLE: condition fails when start_type=5 (IS_DOUBLE)
    // and end_type=7 (IS_ARRAY) because 5+7 = 2*6 (IS_STRING)
    if (start_type + end_type < 2*IS_STRING) {
      // ... handle mixed type inputs and convert to numeric
      goto handle_numeric_inputs;
    }
    // TYPE CONFUSION OCCURS HERE:
    //   When the vulnerable condition fails, we reach this point
    //   with non-string types, but try to access them as strings

    unsigned char low = Z_STRVAL_P(user_start)[0];
    unsigned char high = Z_STRVAL_P(user_end)[0];
    // ... character range generation logic
    return;
  }
handle_numeric_inputs:
  if (start_type == IS_DOUBLE || end_type == IS_DOUBLE) {
    // ... process numeric ranges (floats)
```

[1] https://github.com/php/php-src/commit/1d6f344bea49ccad82b9a95a80ed9fdc39e260a1

Developer Thinkings

Develop Patch & New Test & Spec

```
@@ -2924,8 +2924,8 @@ PHP_FUNCTION(range)
   /* If the range is given as strings, generate an array of characters. */
   if (start_type ⩾ IS_STRING || end_type ⩾ IS_STRING) {
-    if (UNEXPECTED(start_type + end_type < 2*IS_STRING)) {
+    if (UNEXPECTED(start_type < IS_STRING || end_type < IS_STRING)) {
       if (start_type < IS_STRING) {
         if (end_type ≠ IS_ARRAY) {
           php_error_docref(NULL, E_WARNING, "Arg #1 ($start) must ……"
```

```
--TEST--
GH-13094 (range(9.9, '0') causes seg fault)
--FILE--
<?php
var_dump(range(9.9, '0'));
?>
--EXPECT--
array(10) {
  [0]⇒ float(9.9)
  [1]⇒ float(8.9)
  [2]⇒ float(7.9)
  [3]⇒ float(6.9)
  [4]⇒ float(5.9)
  [5]⇒ float(4.9)
  ……
```

Return Values

Returns a sequence of elements as an array with the first element being **start** going up to **end**, with each value of the sequence being **step** values apart.

The last element of the returned array is either **end** or the previous element of the sequence, depending on the value of **step**.

If both **start** and **end** are strings, and **step** is int the produced array will be a sequence of bytes, generally latin ASCII characters.

If at least one of **start**, **end**, or **step** is float the produced array will be a sequence of float.

Otherwise, the produced array will be a sequence of int.

Output of LLM Patched Program

The patch mitigates the exploit but contradicts the language specification.

I named the new testcase with “PoC⁺ Test”

```
@@ -2960,6 +2960,14 @@ PHP_FUNCTION(range)
   }

   /* Generate array of characters */
+  if (Z_TYPE_P(user_start) != IS_STRING) {
+    zend_argument_value_error(1, "must be a string");
+    RETURN_THROWS();
+  }
+  if (Z_TYPE_P(user_end) != IS_STRING) {
+    zend_argument_value_error(2, "must be a string");
+    RETURN_THROWS();
+  }

   unsigned char low = Z_STRVAL_P(user_start)[0];
   unsigned char high = Z_STRVAL_P(user_end)[0];
```

```
Fatal error: Uncaught ValueError: range():
Argument #1 ($start) must be a valid string in /test.php:2
Stack trace:
#0 /test.php(2): range(9.9, '0')
#1 {main}
  thrown in /test.php on line 2
```

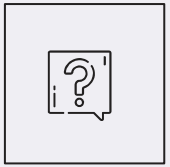
Research Questions

Quantifying Overestimation

To what extent does conventional test-suite-based validation overestimate AVR effectiveness?

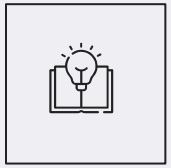
PoC⁺ Test Principle & Production

How do developers construct PoC+ tests, and what additional semantics do they encode?



Reliability of PoC⁺ Test

How reliable are PoC⁺ tests as a validation method for evaluating patch correctness?



False Positive Analysis

What are the underlying causes of false positive patches in modern AVR systems?

PVBench Dataset

- 209 real-world vulnerabilities
- 20 mature open-source projects
- 12 CWE categories
- Primarily memory-safety bugs (C/C++)

Representative projects include:
PHP, CPython, LLVM, V8, libxml2, HDF5, etc.

Project	LoC	#	Test	Project	LoC	#	Test
php	1390.2K	43	18.7K	vim	564.2K	11	5.2K
cpython	745.9K	33	48.6K	hdf5	1334.4K	8	0.6K
llvm	8980.4K	26	128.7K	exiv2	93.5K	7	0.3K
v8	6225.6K	24	53.7K	wabt	514.9K	5	1.1K
libxml2	200.4K	19	3.3K	hermes	590.0K	4	2.3K
icu	1241.5K	15	2.0K	pcap++	160.0K	3	0.3K
quickjs	78.8K	2	79.7K	libtiff	109.0K	1	0.2K
mruby	152.4K	2	1.7K	jasper	5.5K	1	0.2K
jq	4.7K	2	0.9K	simdjson	547.5K	1	0.1K
htslib	108.3K	1	0.4K	wireshark	6088.9K	1	0.1K

Total Vulnerabilities: 209

CWE	#	Description	CWE	#	Description
CWE-476	52	NULL Dereference	CWE-670	3	Incorrect Control Flow
CWE-617	40	Reachable Assertion	CWE-415	3	Double Free
CWE-122	34	Heap Overflow	CWE-704	3	Type Confusion
CWE-416	32	Use After Free	CWE-457	1	Uninitialized Memory
CWE-190	26	Integer Overflow	CWE-362	1	Race Condition
CWE-121	13	Stack Overflow	CWE-369	1	Divide by Zero

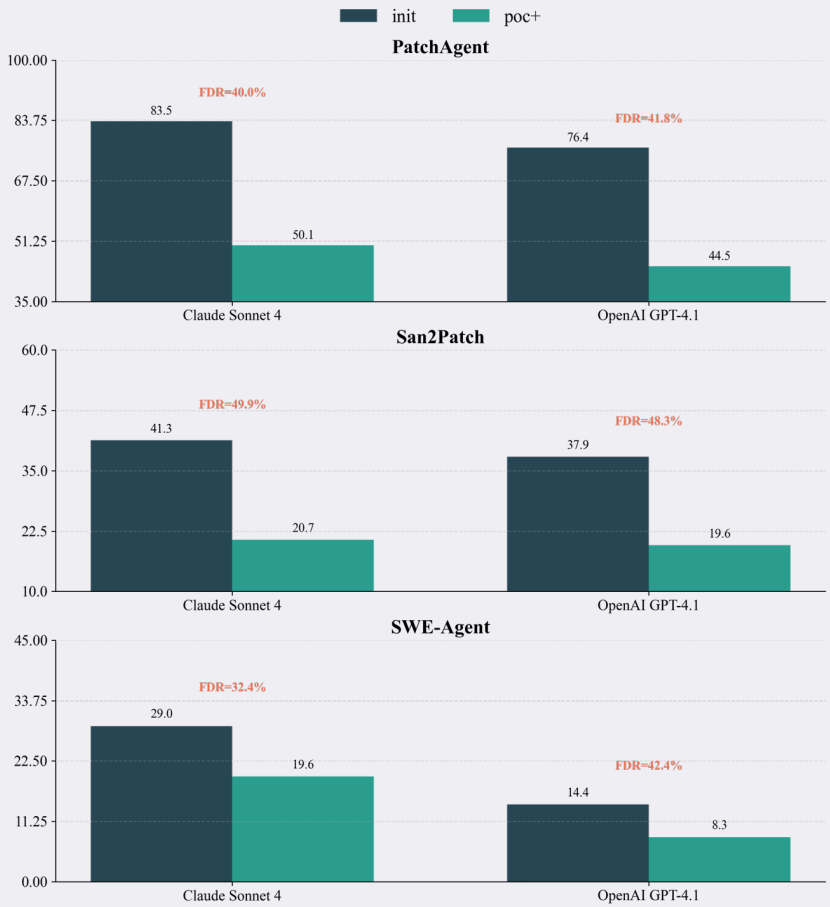
Quantifying Overestimation

We evaluate three LLM-based AVR systems.

Each configuration is executed five times per vulnerability (209 × 5 attempts).

Stage 1: PoC & existing functional tests
Stage 2: PoC⁺ tests (developer-authored)

False Discovery Rate ≈ 42%
Over 40% of patches validated as correct fail under PoC⁺ testing



PoC+ Test Category

Developers typically follow consistent test patterns within the same project.

Across PVBench, we observe that PoC+ tests naturally fall into three recurring categories:

- **Output Checking**: validate observable outputs
- **Intermediate Checking:** assert internal states or return values
- **Self Checking:** embed runtime assertions in interpreted code

These categories reflect project-specific testing conventions.

Category	Projects
Output	exiv2, hermes, htslib jasper, libxml2, php jq, llvm-project, simdjson wabt, wireshark
Intermed.	hdf5, icu pcapplusplus, libtiff
Self	cpython, mruby, quickjs v8, vim

Output Checking

Validation Logic
The patched program must reproduce the expected functional output.

Construction Principle
1. Reuse the original PoC input.
2. Execute the program with the developer patch applied.
3. Capture the expected output as the oracle.
4. Encode this expected output into the test specification.

```
class issue_2377_buffer_overflow(metaclass=CaseMeta):
    filename = "$data_path/issue_2377_poc.mp4"
    commands = ["$exiv2 $filename"]
    retval   = [253]
    stderr   = ["$filename: No Exif data found in the file\\n"]
    stdout   = ["File name: $filename\\nFile size: 225 Bytes\\n..."]
```

```
// CHECK-LABEL: func.func @nested_muli() -> i32 {
// CHECK:         %[[VAL_0:.*]] = "test.constant"() ...
// CHECK:         %[[VAL_1:.*]] = arith.muli %[[VAL_0]], ...
func.func @nested_muli() -> (i32) {
  %0 = "test.constant"() {value = 0x7fffffff} : () -> i32
  %1 = "test.constant"() {value = -2147483648} : () -> i32
  ...
}
```

```
<!-- oss-fuzz-51295_0.xsd (input) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="e" substitutionGroup="e"/>
</xs:schema>

<!-- oss-fuzz-51295_0.err (expected error output) -->
element decl. 'e': The element declaration 'e' defines
a circular substitution group to element declaration 'e'.
```

Intermediate Checking

Validation Logic

The test validates whether the behavior of each intermediate steps are correct.

Construction Principle

1. Start from the original PoC.
2. Insert assertions on return values and internal states.
3. Explicitly verify expected control-flow transitions.

```
// Original PoC for hdf5 bug
space_id = H5Screate_simple(1, dims, NULL);
H5Sselect_hyperslab(space_id, ...);
H5Sset_extent_none(space_id);
H5Sget_select_hyper_blocklist(space_id, ...);
H5Sclose(space_id);
```

```
static void poc_plus_test(void) {
  hsize_t dims[]  = {10}; /* ... initialization ... */
  space_id = H5Screate_simple(1, dims, NULL);
  CHECK(space_id, H5I_INVALID_HID, "H5Screate_simple");
  ret = H5Sselect_hyperslab(space_id, ...);
  CHECK(ret, FAIL, "H5Sselect_hyperslab");
  ret = H5Sset_extent_none(space_id);
  CHECK(ret, FAIL, "H5Sset_extent_none");
  ret = H5Sget_select_hyper_blocklist(space_id, ...);
  VERIFY(ret, FAIL, "H5Sget_select_hyper_blocklist");
  ret = H5Sclose(space_id);
  CHECK(ret, FAIL, "H5Sclose");
}
```

Self Checking

Validation Logic

The patched interpreter must preserve language semantics, raise correct exceptions, return correct values, etc.

Construction Principle

1. Transform the original crashing script.
2. Embed runtime assertions into the interpreted program.
3. Validate expected exception types or side effects.

```python
# Original PoC for CPython sys bug
import sys
sys.remote_exec(0, None)
```

```python
# PoC+ test with self-test
import sys
with self.assertRaises(TypeError):
    sys.remote_exec(0, None)
with self.assertRaises(TypeError):
    sys.remote_exec(0, 123)
```

Are PoC⁺ Tests Reliable?

Basic Tests ↠ Overestimate

PoC⁺ eliminates ~ 40% false positives

PoC⁺ Test ↠ ?

But does passing PoC⁺ imply correct repair?

Reliability Evaluation Framework

Evaluation Pipeline

1702 PoC+ Passing Patches

↓

Manual Semantic Comparison

↓

Categorization by Semantic Distance

↓

Agreement & Consensus Validation

Classification Criteria

Level 1 — Semantic Equivalence

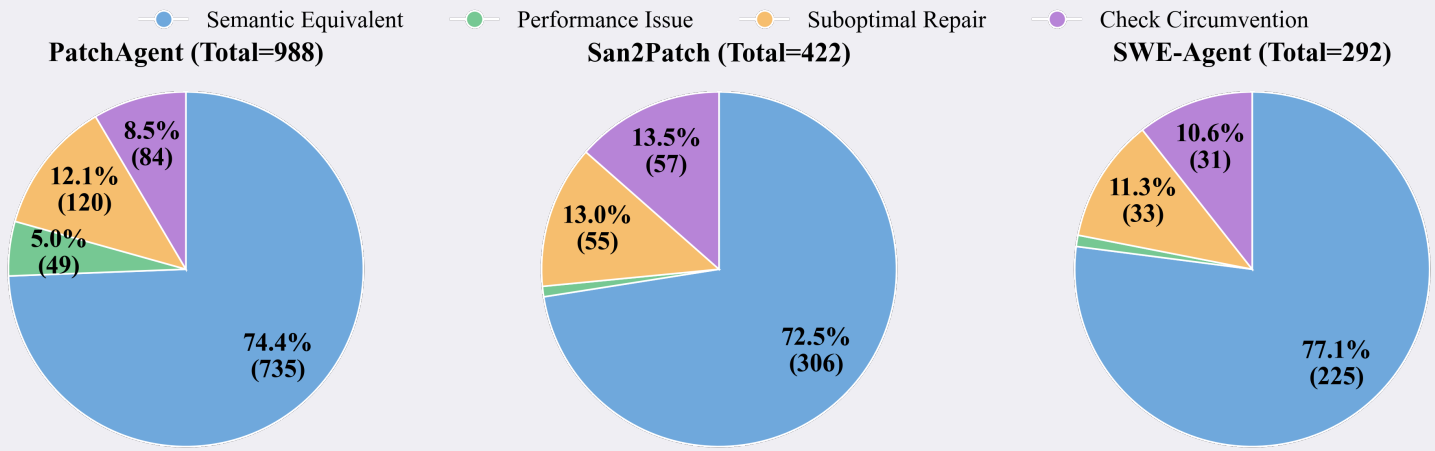
- Same target function
- Same asymptotic complexity
- Same repair logic
- No unintended side effects

Level 2 — Non-Equivalent

- Suboptimal Repair
- Performance Issue
- Check Circumvention

We measure semantic alignment rather than syntactic similarity

PoC⁺ Exhibits Low False Acceptance Rate



Among 1702 PoC⁺ passing patches:
~74% Fully Semantically Aligned

Performance Issue

Performance issues arise when patches are functionally correct but employ repair strategies with suboptimal algorithmic complexity compared to developer solutions.

```c
static int template_clear(TemplateObject *self) {
    Py_CLEAR(self->literal);
    for (Py_ssize_t i = 0, n = Py_SIZE(self); i < n; i++)
        // BUG: items[i].literal may be uninitialized
        Py_CLEAR(self->items[i].literal);
    return 0;
}
static PyObject *sre_template_impl(..., PyObject *template) {
    // template is a list containing interleaved
    // literal strings (str or bytes) and group indices (int)
    // [literal1, group1, literal2, ..., literalN].
    Py_ssize_t n = /* Extract number of groups */;
    // Allocate array but items[].literal not initialized
    TemplateObject *self = PyObject_GC_NewVar(TemplateObject, ..., n);
    self->literal = Py_NewRef(PyList_GET_ITEM(template, 0));
    // FIX OPTION 1: Initialize all literal fields to NULL
    // This prevents crashes since Py_CLEAR safely handles NULL
    // for (Py_ssize_t i = 0; i < n; i++)
    //     self->items[i].literal = NULL;
    for (Py_ssize_t i = 0; i < n; i++) {
        Py_ssize_t index = // Extract (i+1)-th group number;
        if (index < 0) {
            // FIX OPTION 2: Set object size to track
            // how many items were successfully initialized.
            // template_clear will only clean initialized items.
            // Py_SET_SIZE(self, i);
            goto bad_template;
        }
        // Normal case: would initialize items[i].literal here...
    }
    return (PyObject*) self;
bad_template:
    PyErr_SetString(PyExc_TypeError, "invalid template");
    Py_XDECREF(self); // Triggers template_clear() cleanup
    return NULL;
}
```

Suboptimal Repair

Patches exhibit inferior implementation quality that makes their correctness less apparent. With no intuitive justification on why code changes are made at specific locations, this type of patch eventually hurts the maintainability of the overall codebase.

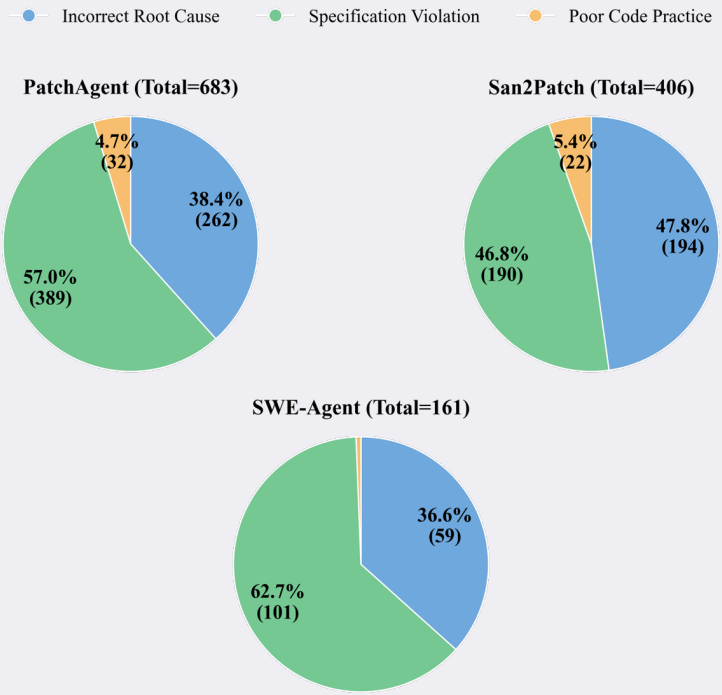
```
@@ -724,7 +724,8 @@ void add_class_vars(...)
     if (((info->flags & ACC_PROTECTED) &&
          !check_protected(info->ce, scope)) ||
          ((info->flags & ACC_PRIVATE) &&
-          info->ce != scope)) {
+          info->ce != scope) ||
+         (info->flags & ACC_VIRTUAL)) {
          continue;
     }
     prop = NULL;
```

```
@@ -729,10 +729,14 @@ void add_class_vars(...)
     }
     prop = NULL;
     if (statics && (info->flags & ACC_STATIC) != 0) {
-        prop = &ce->static_members_table[info->offset];
+        if (ce->static_members_table &&
+            info->offset < ce->static_members_count) {
+            prop = &ce->static_members_table[info->offset];
+        }
     } else if (!statics && (info->flag & ACC_STATIC) == 0) {
-        prop = &property_table[info->offset];
+        if (property_table && info->offset < ce->property_count) {
+            prop = &property_table[info->offset];
+        }
      }
      if (!prop) {
         continue;
```

Categorization of False Positive

I conducted a systematic analysis of all 1250 false positives.

1. Incorrect Root Cause (~41%)
2. Specification Violation (~54%)
3. Poor Code Practice (~4%)



Incorrect Root Cause

Symptom-level fix instead of cause-level repair.

Patch modifies a different function than developer patch

- Fixes crash location, not vulnerability origin
- Violates architectural intent

_fields

Each concrete class has an attribute `_fields` which gives the names of all child nodes.

Each instance of a concrete class has one attribute for each child node, of the type as defined in the grammar. For example, `ast.BinOp` instances have an attribute `left` of type `ast.expr`.

If these attributes are marked as optional in the grammar (using a question mark), the value might be `None`. If the attributes can have zero-or-more values (marked with an asterisk), the values are represented as Python lists. All possible attributes must be present and have valid values when compiling an AST with `compile()`.

```
@@ -2234,6 +2234,10 @@ PySequence_Count(...) {
 PySequence_Contains(PyObject *seq, PyObject *ob)
 {
+    if (seq == NULL) {
+        null_error();
+        return -1;
+    }
     PySeqMethods *sqm = Py_TYPE(seq)->tp_as_sequence;
     if (sqm != NULL && sqm->sq_contains != NULL) {
```

```
@@ -5083,19 +5083,17 @@ ast_type_init(...) {
     PyObject *key, *value, *fields, *attributes = NULL;
-    if (PyObj_GetOptionalAttr((PyObject*)Py_TYPE(self),
-                              state->_fields, &fields) < 0) {
+    fields = PyObj_GetAttr((PyObject*)Py_TYPE(self),
+                           state->_fields);
+    if (fields == NULL)
         goto cleanup;
-    if (fields) {
-        numfields = PySequence_Size(fields);
-        if (numfields == -1)
-            goto cleanup;
-        remaining_fields = PySet_New(fields);
-    }
-    else
-        remaining_fields = PySet_New(NULL);
+    numfields = PySequence_Size(fields);
+    if (numfields == -1)
+        goto cleanup;
+    remaining_fields = PySet_New(fields);
     if (remaining_fields == NULL)
         goto cleanup;
```

Specification Violation

```
@@ -2924,8 +2924,8 @@ PHP_FUNCTION(range)
   /* If the range is given as strings, generate an array of characters. */
   if (start_type ⩾ IS_STRING || end_type ⩾ IS_STRING) {
-    if (UNEXPECTED(start_type + end_type < 2*IS_STRING)) {
+    if (UNEXPECTED(start_type < IS_STRING || end_type < IS_STRING)) {
       if (start_type < IS_STRING) {
         if (end_type ≠ IS_ARRAY) {
           php_error_docref(NULL, E_WARNING, "Arg #1 ($start) must ……"

Fatal error: Uncaught ValueError: range():
Argument #1 ($start) must be a valid string in /test.php:2
Stack trace:
#0 /test.php(2): range(9.9, '0')
#1 {main}
  thrown in /test.php on line 2
```

```
--TEST--
GH-13094 (range(9.9, '0') causes seg fault)
--FILE--
<?php
var_dump(range(9.9, '0'));
?>
--EXPECT--
array(10) {
  [0]⇒ float(9.9)
  [1]⇒ float(8.9)
  [2]⇒ float(7.9)
  [3]⇒ float(6.9)
  [4]⇒ float(5.9)
  [5]⇒ float(4.9)
  ……
```

Return Values

Returns a sequence of elements as an array with the first element being **start** going up to **end**, with each value of the sequence being **step** values apart.

The last element of the returned array is either **end** or the previous element of the sequence, depending on the value of **step**.

If both **start** and **end** are strings, and **step** is int the produced array will be a sequence of bytes, generally latin ASCII characters.

If at least one of **start**, **end**, or **step** is float the produced array will be a sequence of float.

Otherwise, the produced array will be a sequence of int.

Poor Code Practice

Location correct, reasoning correct

- Breaks project coding conventions
- Introduces control-flow shortcuts

Patch compiles — but not robust

```
    GET_NODE(sxe, node);
+   if (!node) {
+     /* avoid null dereference */
+     return &EG(err_zval);
+   }
    php_libxml_invalidate_node_from_doc(node->doc);
    if (node) {
         if (attribs) {
```

Conclusion

LLMs Can Mimic Human Expertise for Vulnerability Repair

By modeling human developer workflows, **PatchAgent** achieves up to 92.13% repair accuracy on 178 real-world vulnerabilities through multi-model collaboration. **PortGPT** extends this to patch backporting, outperforming all baselines across C, C++, and Go, with 9 patches merged into Linux 6.1. Together, they show LLMs can perform end-to-end vulnerability repair at scale.

Validation Is the Core Challenge of Modern AVR

Test-suite based validation substantially overestimates repair correctness. By constructing **PoC+ tests** and building PVBench, I quantified a ~42% false discovery rate, reframing repair reliability as a semantic alignment problem.

Understanding Failure Enables the Next Generation of Repair

Through systematic analysis of 1250 false positives, I identified three dominant failure modes: incorrect root cause (~41%), specification violation (~54%), and poor code practice (~4%). These findings lay the foundation for root-cause-aware, specification-driven repair systems.

Future Work

Rapid Engineering Iteration in a Fast-Evolving LLM Ecosystem

The LLM landscape is evolving at an unprecedented speed. From Codex-style agents to Claude Code and emerging autonomous engineering systems. To keep AVR systems competitive and impactful, we need faster engineering iteration cycles.

Incorporating Domain Knowledge for Aligned Repair

Different projects exhibit vastly different architectural styles, coding conventions, and implicit specifications. To generate truly correct patches, repair systems must go beyond generic reasoning and incorporate project-specific domain knowledge.

Toward More Reliable and Automated Validation

Validation remains the key bottleneck of automated repair. Future work should explore more automated and semantically grounded validation methods. The ultimate goal is to reduce human intervention while increasing trust in automatically generated patches.

Thanks!

Does anyone have any questions?