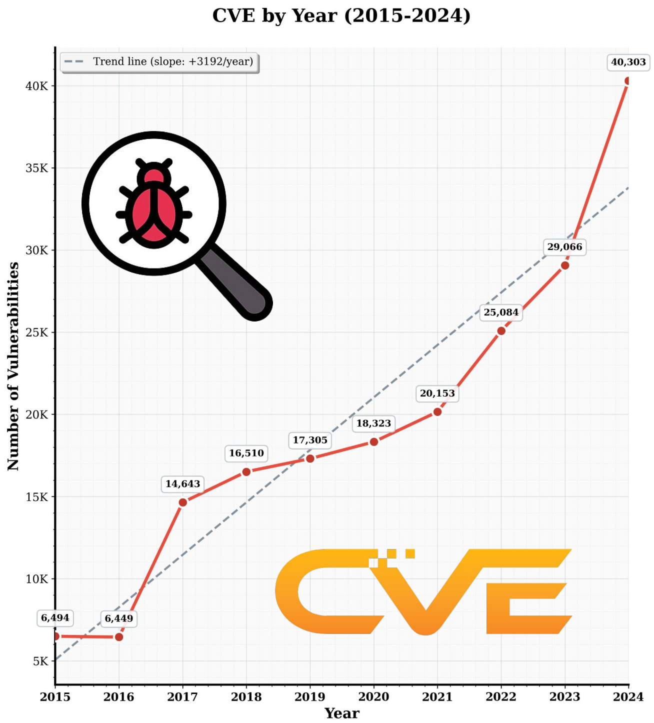
Toward Practical Program Repair

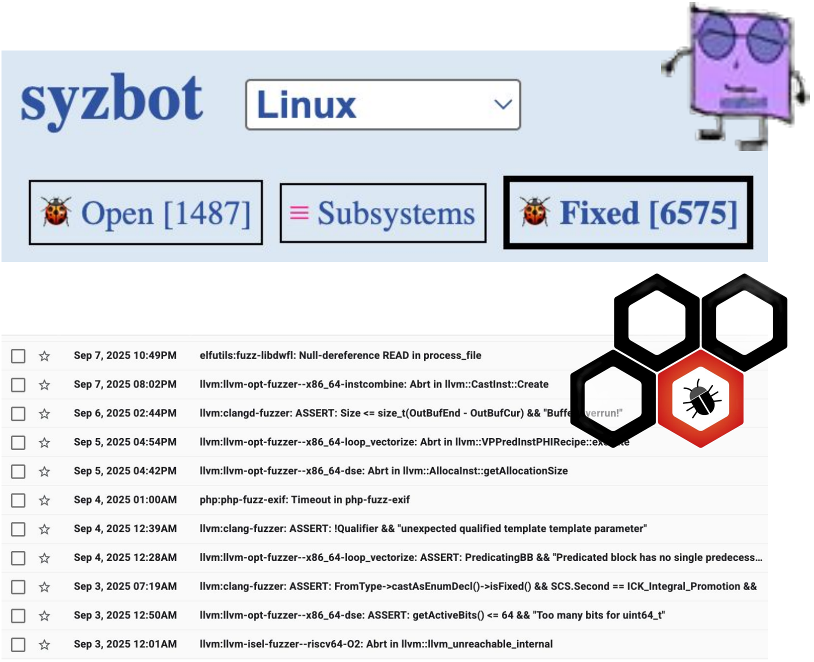
Zheng Yu

Committee: Xinyu Xing (Chair)
Peter Dinda, Yan Chen, Kexin Pei

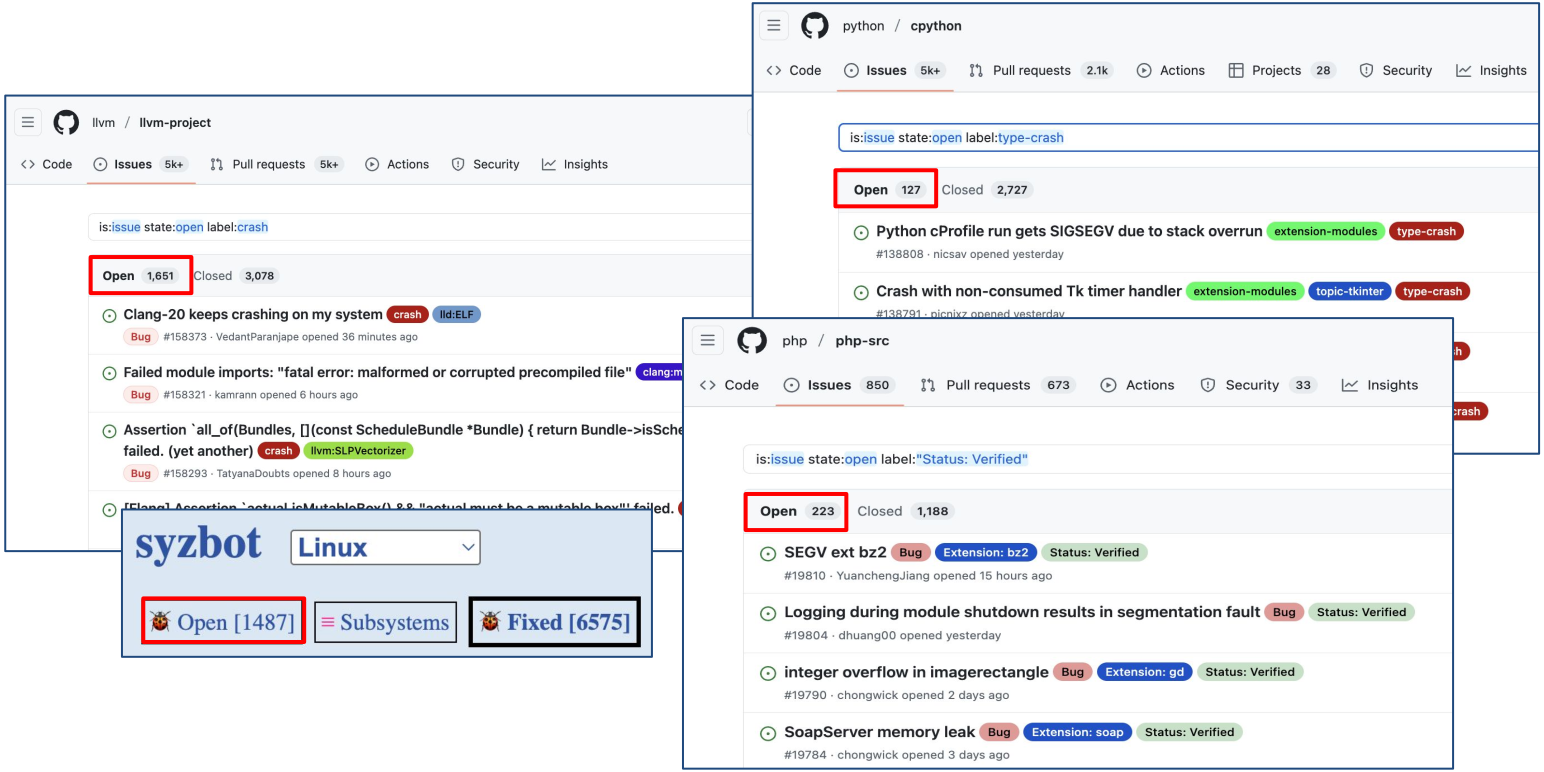
Northwestern

Vulnerability Explosion: Growing Threat Landscape





Thousands of Security Vulnerabilities Remain Unpatched



Typical Automated Program Repair (APR) Workflow

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.*

Unrealistic Assumptions in Previous Studies

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

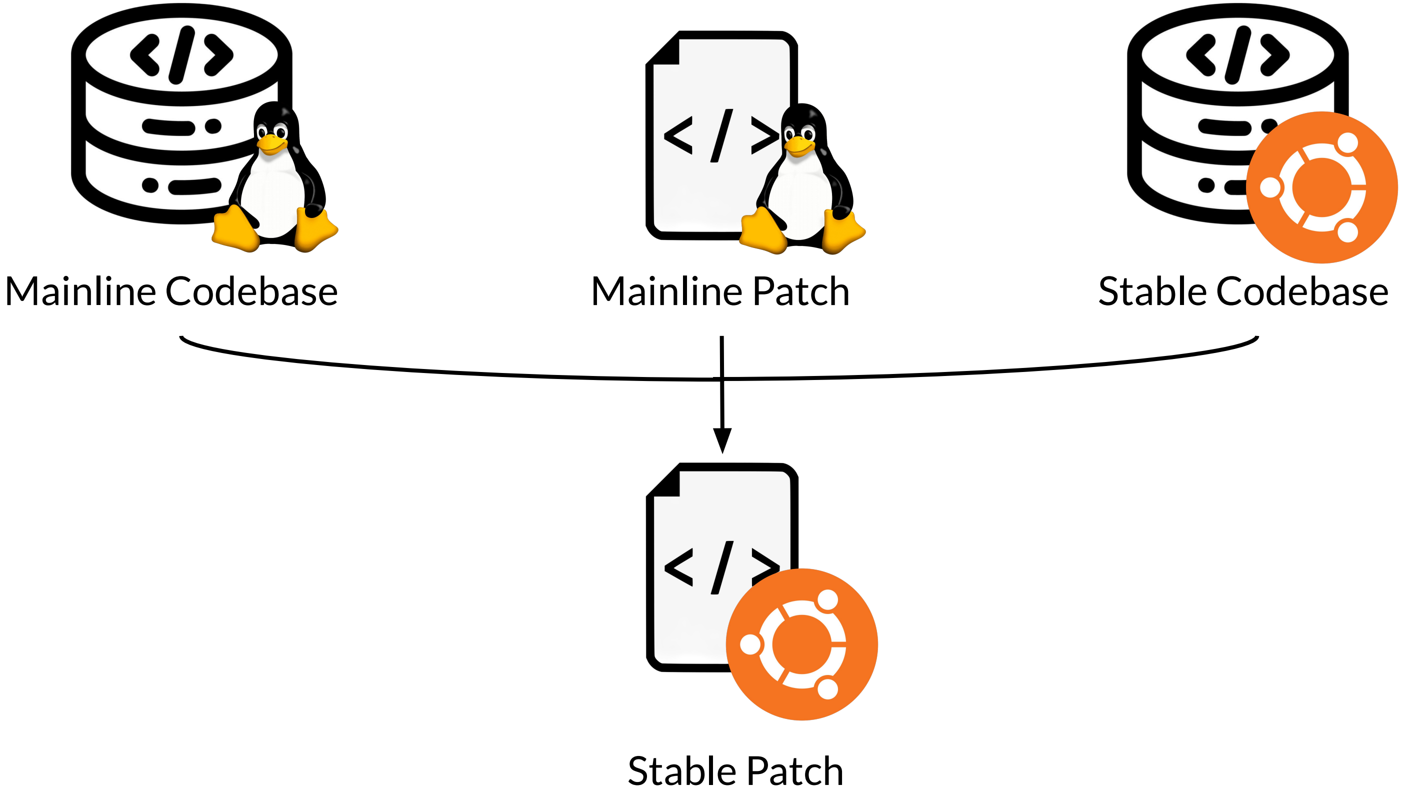
Patch Generation

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.

Scenario 1: Patch Backporting

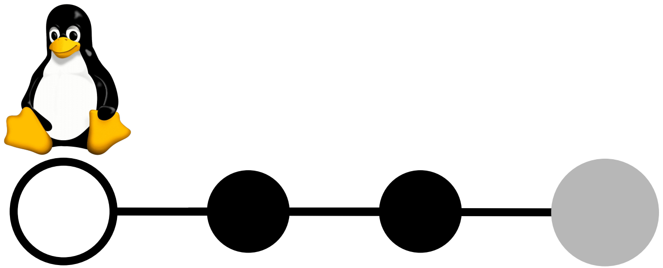


Prior work 1:

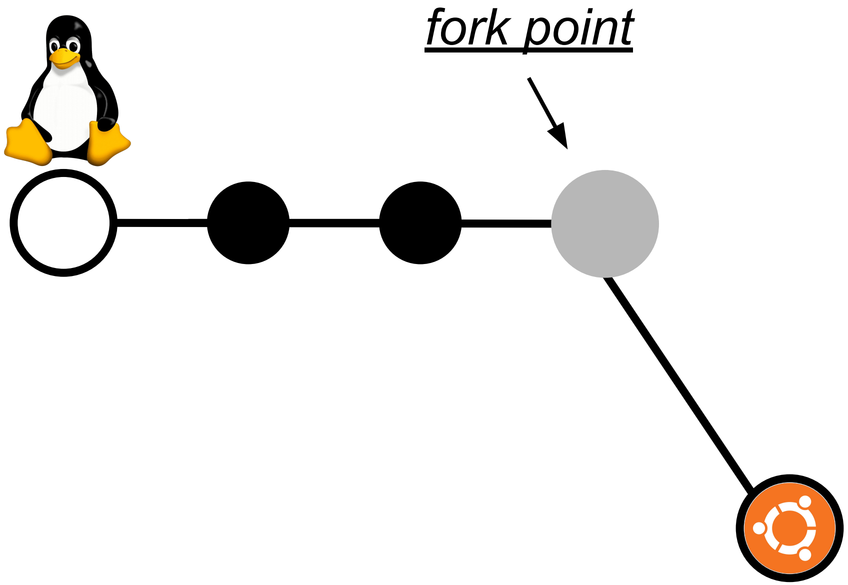
PortGPT: Towards Automated Backporting Using Large Language Models

(IEEE S&P 2026)

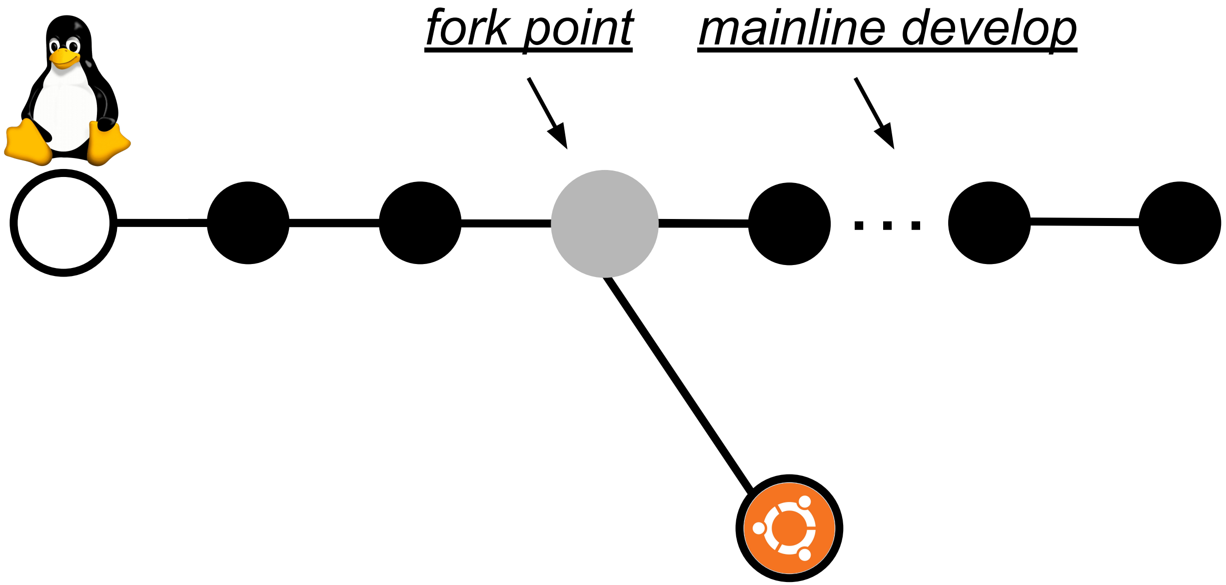
Backporting Background



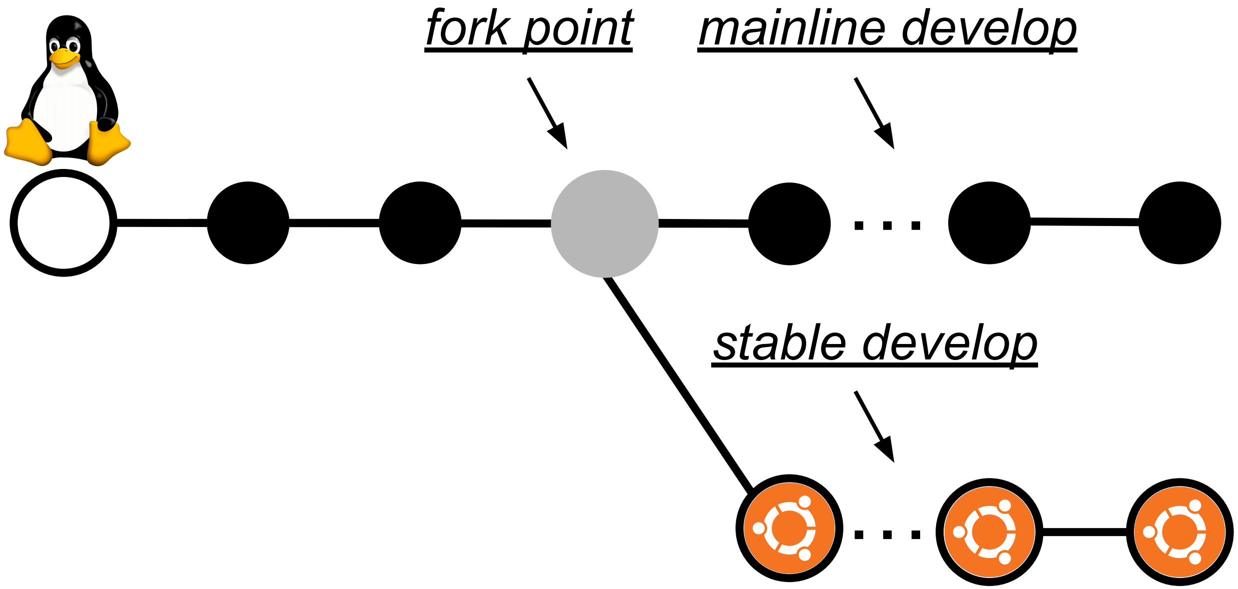
Backporting Background



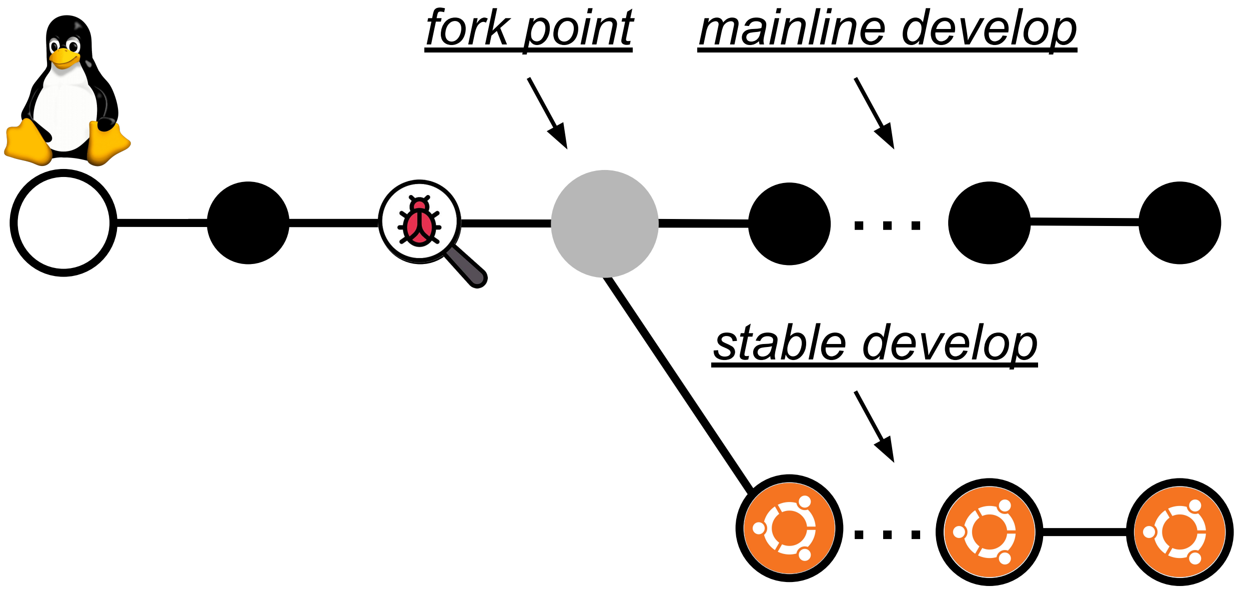
Backporting Background



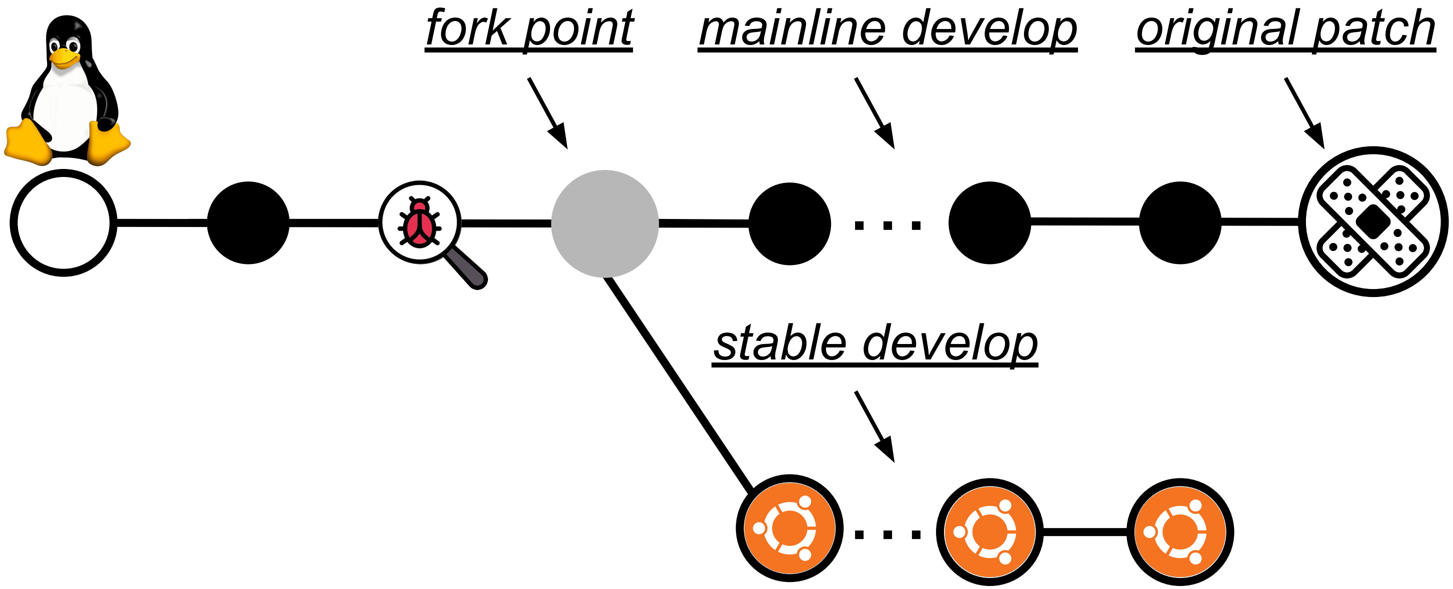
Backporting Background



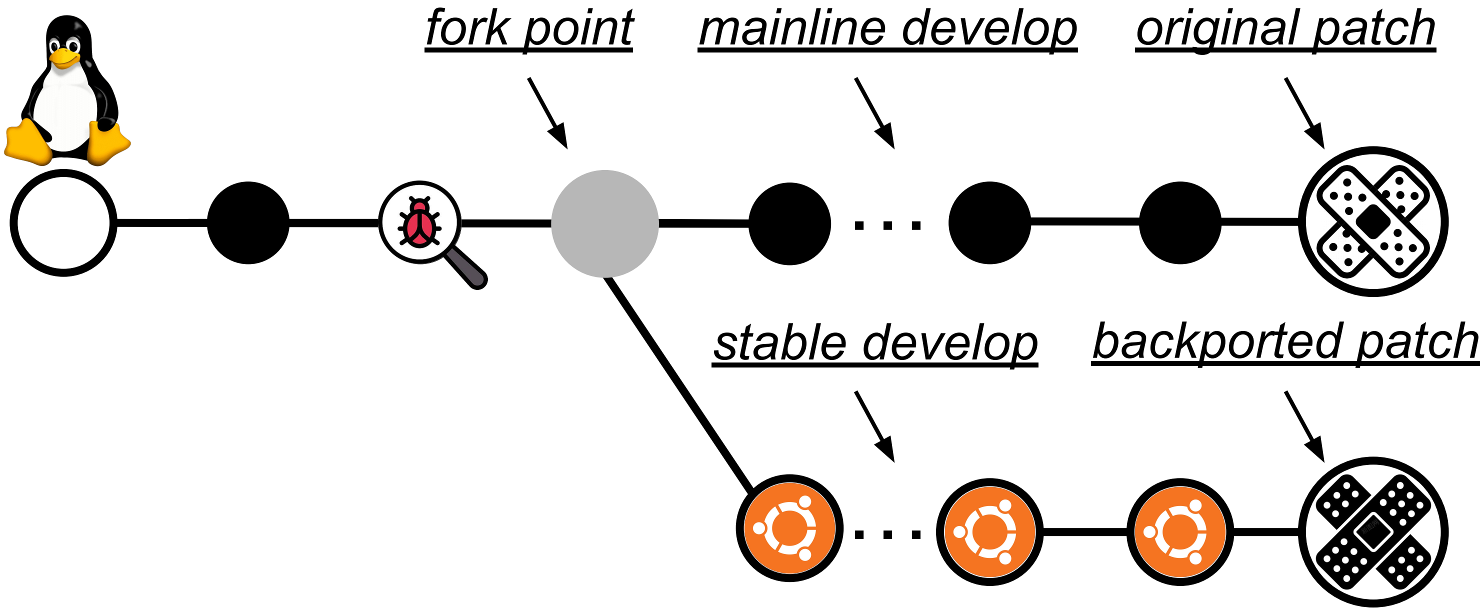
Backporting Background



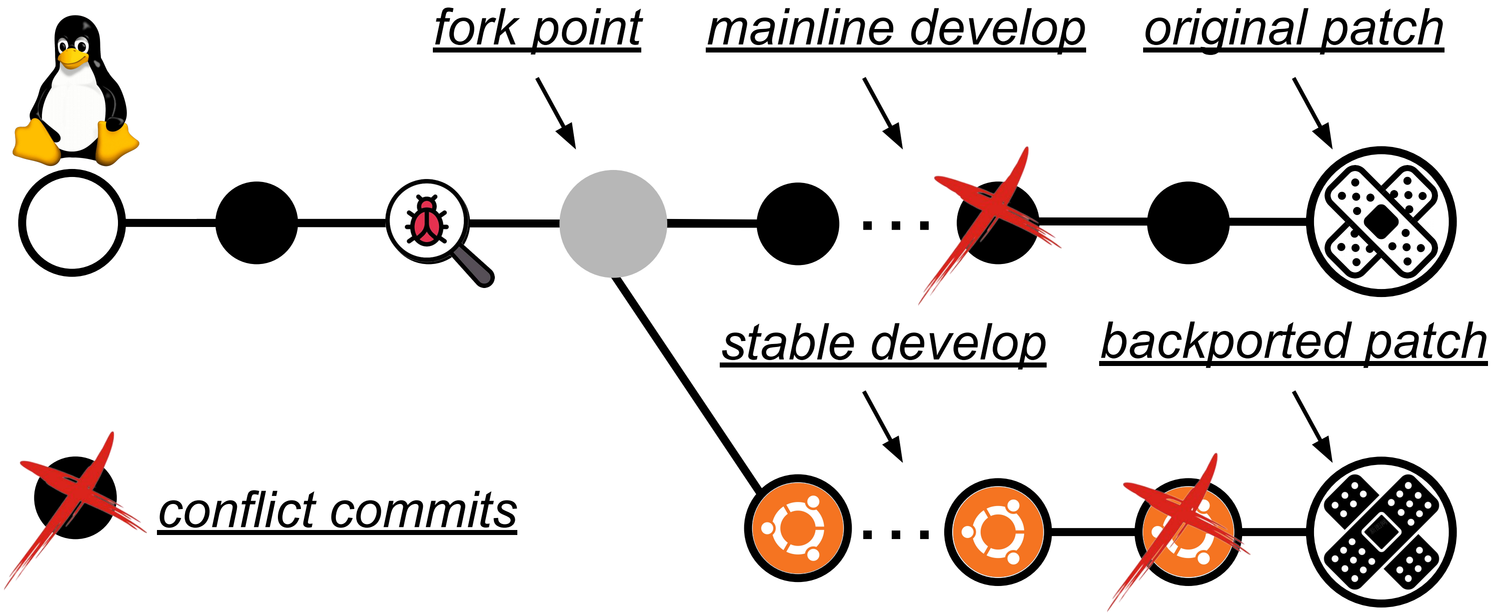
Backporting Background



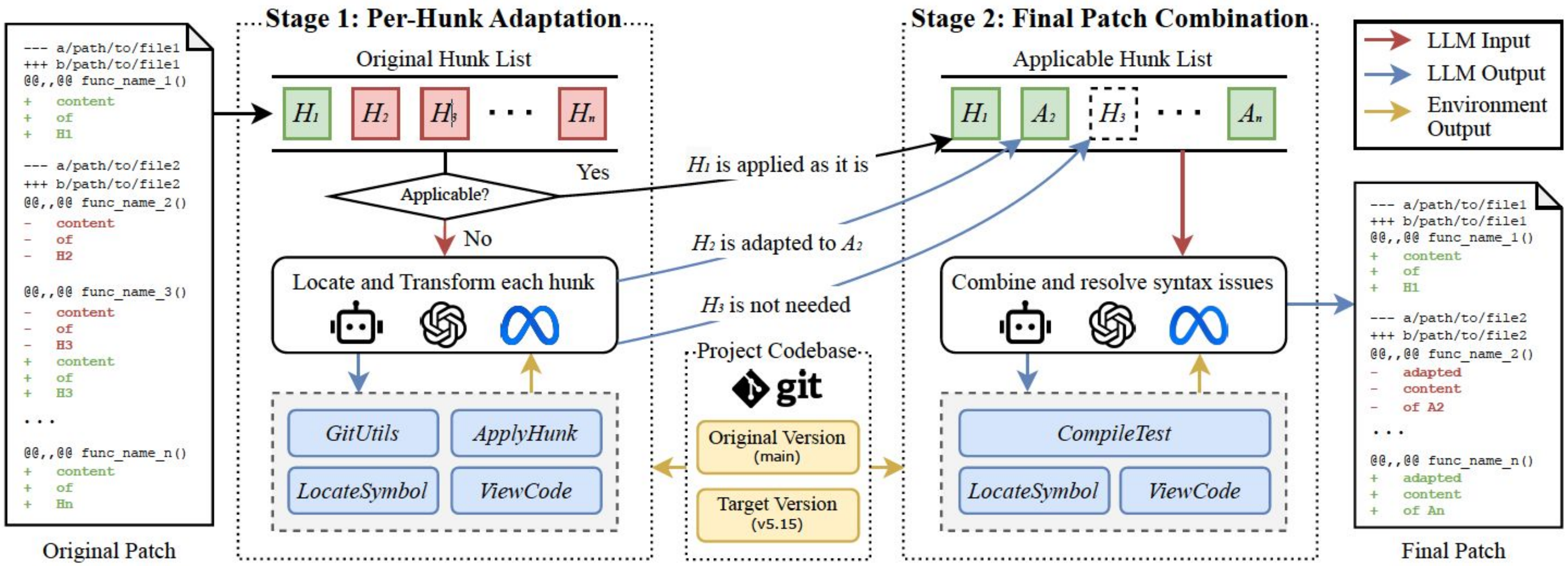
Backporting Background



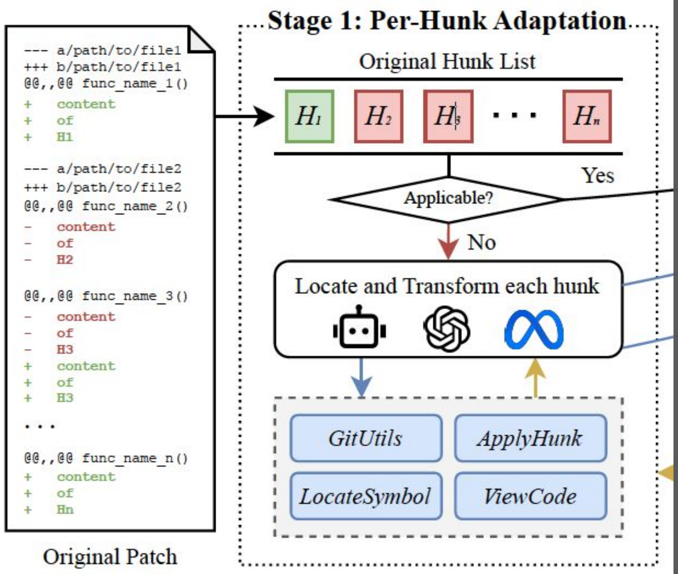
Backporting Background



PortGPT Design



PortGPT Design - Stage 1

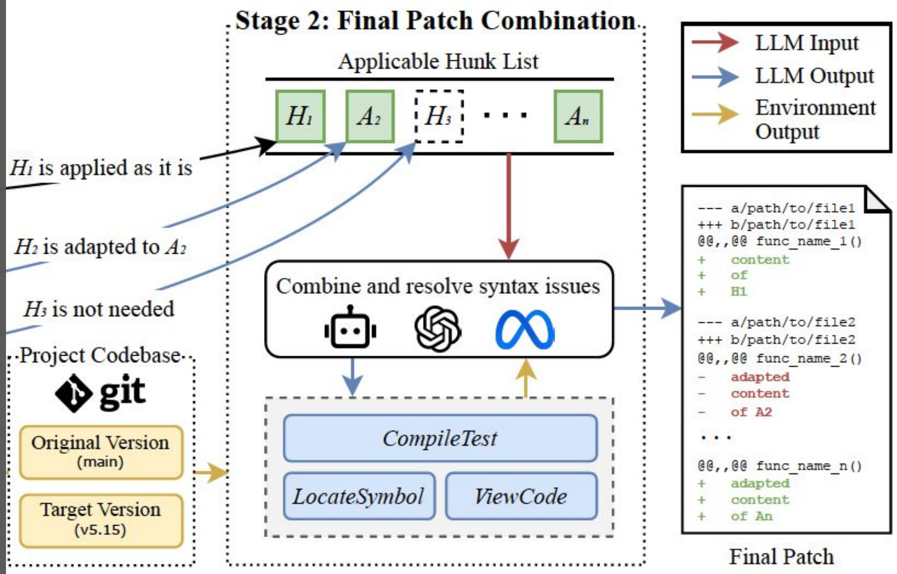


- PortGPT extracts hunks from the original patch and per each hunk.
- PortGPT first determines whether the hunk requires backporting, and if so, transforms the hunk to ensure compatibility with the target version.
- This stage aims to ensure that the generated patch can be successfully applied to the target version.

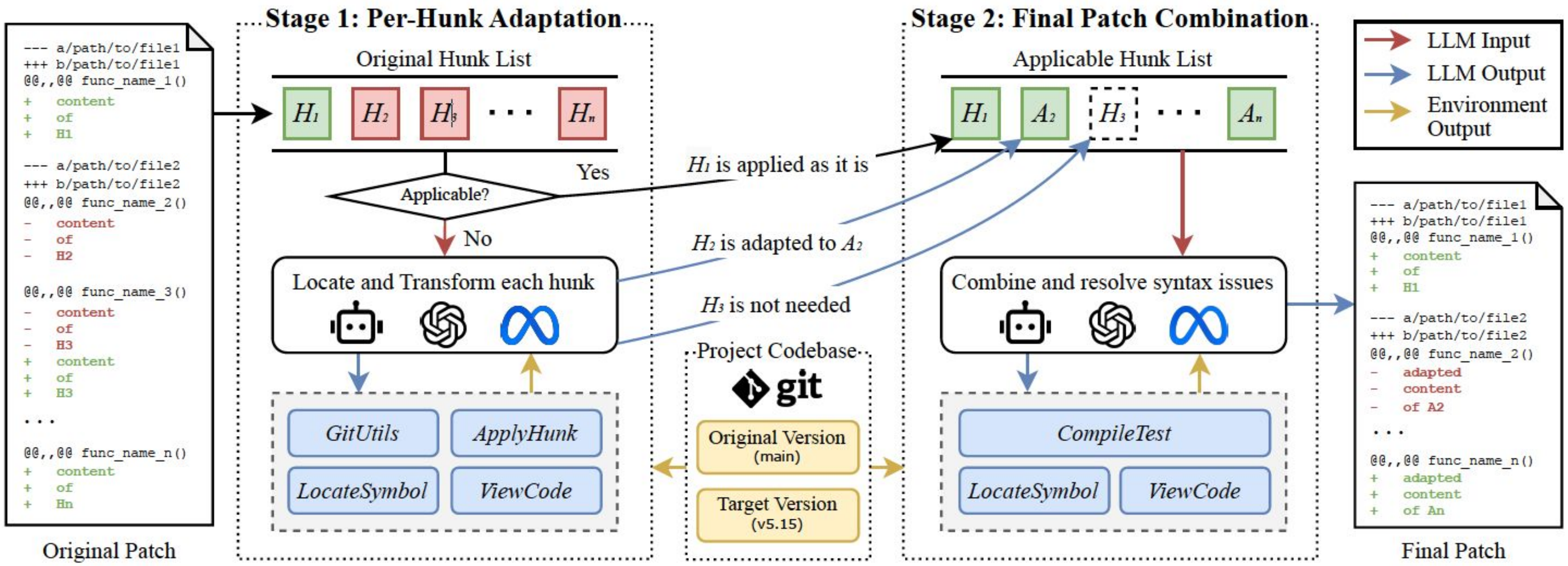
PortGPT Design - Stage 2

- PORTGPT combines the transformed hunks, applies the entire patch to the target version, and sends the <u>backported codebase for compilation.</u> If compilation failed.

- PORTGPT attempts to resolve them by <u>adding necessary definitions </u>or <u>adjusting the code context </u>to finalize the transformation.



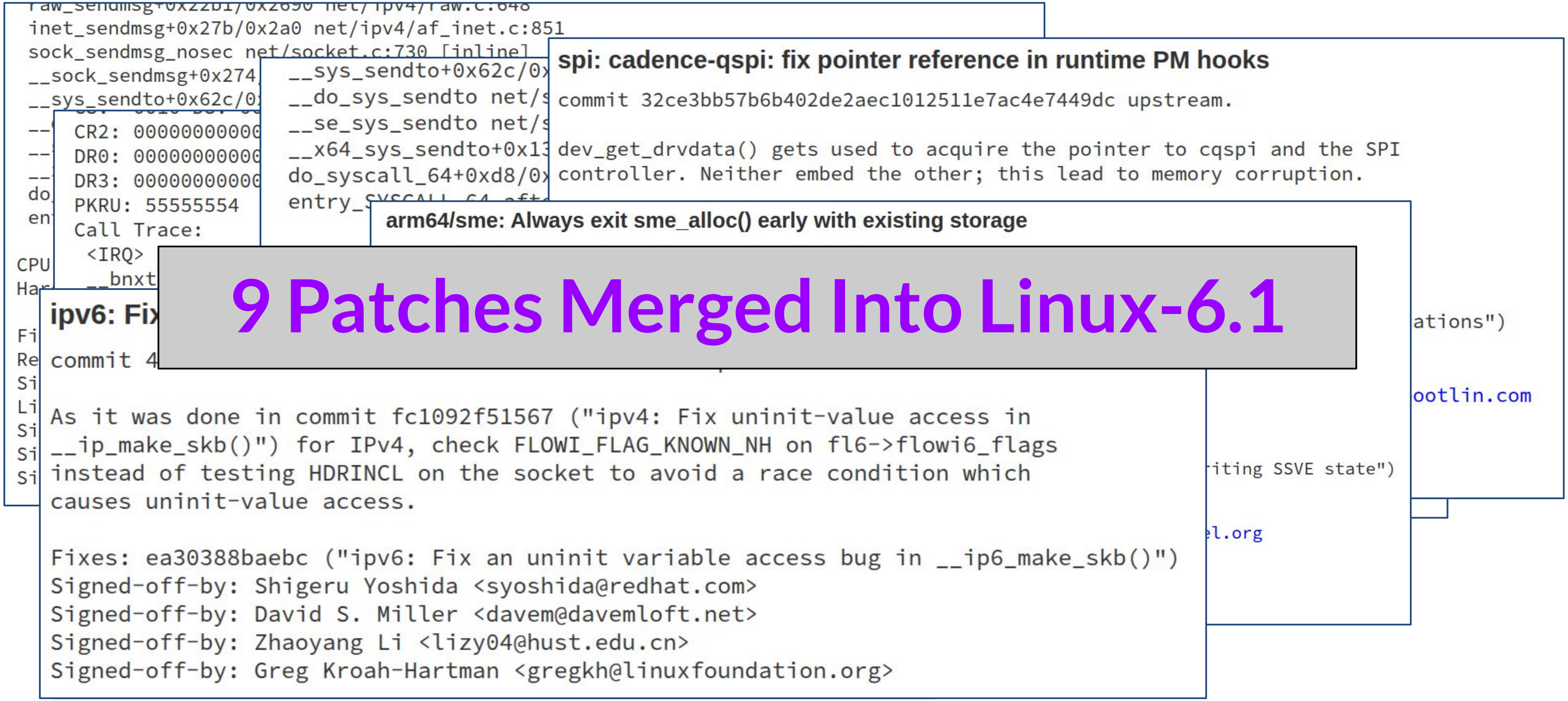
PortGPT Design



Performance Evaluation

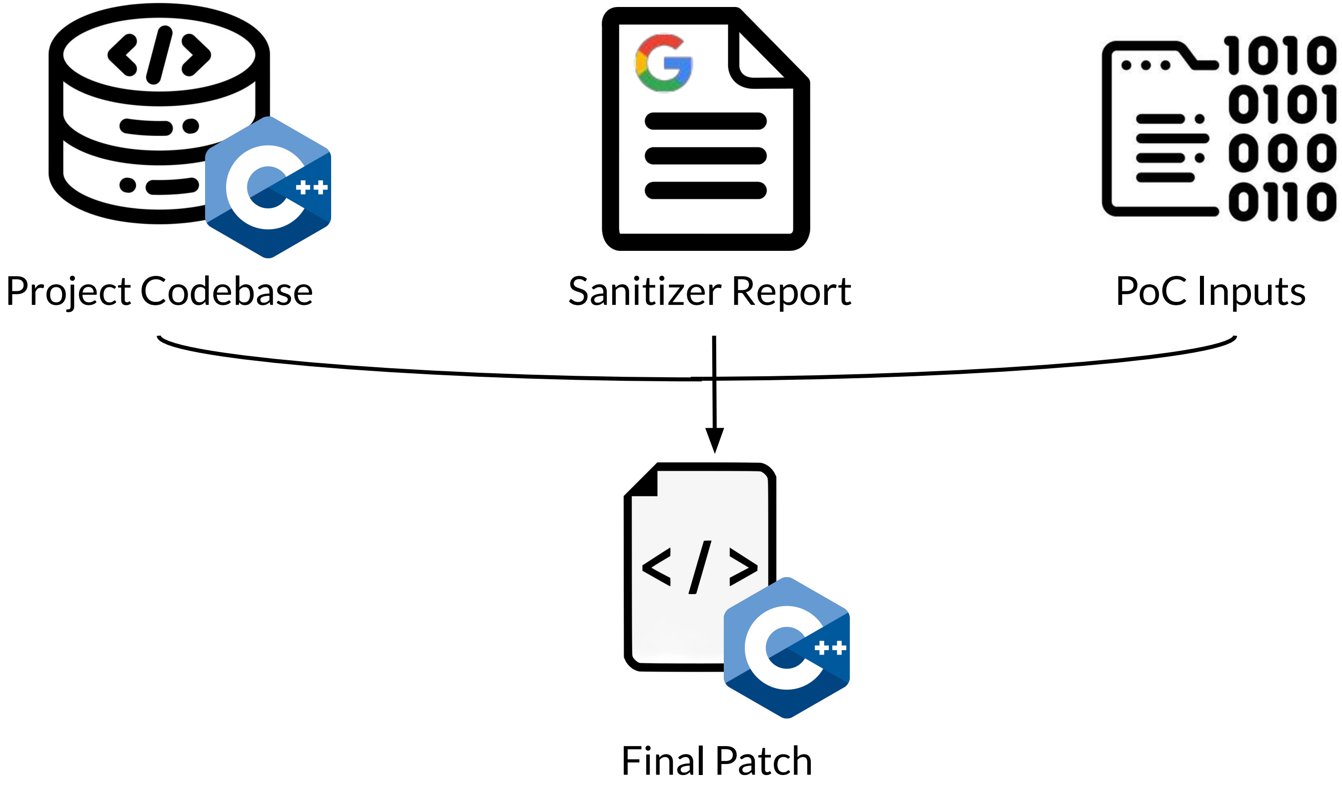
Dataset	System	Type-I	Type-II	Type-III	Type-IV	Total
Prior works [50], [72]	FixMorph	20/170 (11.67%)	374/1208 (30.96%)	23/92 (25.00%)	30/345 (8.70%)	447/1815 (24.63%)
	ChatGPT†	67/170 (39.41%)	451/1208 (37.30%)	16/92 (17.58%)	22/345 (6.38%)	556/1815 (30.63%)
	TSBPORT	170/170 (100.00%)	1190/1208 (98.51%)	69/92 (75.00%)	160/345 (46.38%)	1589/1815 (87.59%)
	TSBPORT*	162/170 (95.29%)	919/1208 (76.08%)	61/92 (66.30%)	150/345 (43.48%)	1292/1815 (71.18%)
	PortGPT	170/170 (100.00%)	1186/1208 (98.18%)	74/92 (80.43%)	188/345 (54.49%)	$\boxed{\text{1618/1815 (89.15\%)}}$
Ours (C)	FixMorph	N/A	0/6 (0.00%)	3/29 (10.34%)	0/34 (0.00%)	3/69 (4.34%)
	TSBPORT	N/A	5/6 (83.33%)	14/29 (48.28%)	5/34 (14.71%)	24/69 (34.78%)
	TSBPORT*	N/A	2/6 (33.33%)	14/29 (48.28%)	4/34 (11.76%)	20/69 (28.99%)
	PortGPT	N/A	6/6 (100%)	21/29 (72.41%)	15/34 (44.12%)	42/69 (60.87%)
Ours (C++)	PortGPT	N/A	N/A	10/11 (90.91%)	5/17 (29.41%)	15/28 (53.57%)
Ours (Go)	PortGPT	N/A	13/13 (100%)	7/9 (77.78%)	14/27 (51.85%)	34/49 (69.39%)

Real World Application



9 Patches Merged Into Linux-6.1

Scenario 2: Repair Fuzzing-Found Vulnerability



Prior work 2:

PatchAgent: A Practical Program Repair Agent Mimicking Human Expertise

(USENIX Sec 2025)

LLM for Program Repair

- ✅ Comprehending bug reports.
- ✅ Comprehending code snippets.
- ✅ Writing a patch.
- ❌ Resolving definitions of symbols.
- ❌ Applying the patch for validation

LLM Agent for Program Repair

- ✅ Comprehending bug reports.
- ✅ Comprehending code snippets.
- ✅ Writing a patch.
- ✅ Resolving definitions of symbols.
- ✅ Applying the patch for validation

Language Server

Patch Verifier

LLM Agent for Program Repair

- ❓ ✅ Comprehending bug reports.
- ❓ ✅ Comprehending code snippets.
- ❓ ✅ Writing a patch.
- ❓ ✅ Resolving definitions of symbols.
- ❓ ✅ Applying the patch for validation

Language Server

Patch Verifier

A Global Buffer Overflow Bug

```
==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:2277
    #2 0x55cbc3 in Compile_If /source/m3_compile.c:1648
    #3 0x5596ec in Compile_BlockStatement /source/m3_compile.c:2207
    #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 (BuildId:
f0fdeb36a)
    ......
    0x555bc18 is located 88 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);
    }
}
```

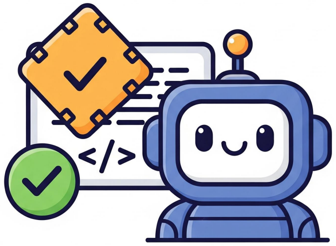
What ability do human/LLM have?

View Code

The viewcode API retrieves the code context by specifying file names and line numbers

Find Definition

The find_definition finds the definition location of symbols by specifying their names and reference locations



Validate

The validate API to check if the patch can mitigate the PoC and pass all functional tests

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:2277
    #2 0x55cbc3 in Compile_If /source/m3_compile.c:1648
    #3 0x5596ec in Compile_BlockStatement /source/m3_compile.c:2207
    #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 (BuildId: f0fdeb36a)
    ......
    0x555bc18 is located 88 bytes after global variable c_operations

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

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.

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:2277
    #2 0x55cbc3 in Compile_If /source/m3_compile.c:1648
    #3 0x5596ec in Compile_BlockStatement /source/m3_compile.c:2207
    #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 (BuildId: f0fdeb36a)
    ......
    0x555bc18 is located 88 bytes after global variable c_operations

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

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)
....
```

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:2277
    #2 0x55cbc3 in Compile_If /source/m3_compile.c:1648
    #3 0x5596ec in Compile_BlockStatement /source/m3_compile.c:2207
    #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 (BuildId: f0fdeb36a)
    ......
    0x555bc18 is located 88 bytes after global variable c_operations

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

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.

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:2277
    #2 0x55cbc3 in Compile_If /source/m3_compile.c:1648
    #3 0x5596ec in Compile_BlockStatement /source/m3_compile.c:2207
    #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 (BuildId: f0fdeb36a)
    ......
    0x555bc18 is located 88 bytes after global variable c_operations

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

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]; ....
```

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:2277
    #2 0x55…
    #3 0x5…                                                   …07
    #4 0x…
    ......
    #8 0x…
    #9 0x5…                                                   …deb36a)
    ......
    0x555bc18 is located 88 bytes after global variable c_operations

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

The report indicates that the overflow occurred in c_operations, so I need to add a boundary check before performing pointer arithmetic.

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]; ....
```

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:2277
    #2 0x55
    #3 0x5                                                          07
    #4 0x5
    .....
    #8 0x5
    #9 0x5                                                             deb36a)
    ......
    0x555bc18 is located 88 bytes after global variable c_operations

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

The report indicates that the overflow occurred in c_operations, so I need to add a boundary check before performing pointer arithmetic.

```
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

```
==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:2277
    #2 0x55cbc3 in Compile_If /source/m3_compile.c:1648
    #3 0x5596ec in Compile_BlockStatement /source/m3_compile.c:2207
    #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 (BuildId: f0fdeb36a)
    ......
    0x555bc18 is located 88 bytes after global variable c_operations

    SUMMARY: AddressSanitizer: global-buffer-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.

Vanilla Agent

```
==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:2277
    #2 0x55cbc3 in Compile_If /source/m3_compile.c:1648
    #3 0x5596ec in Compile_BlockStatement /source/m3_compile.c:2207
    #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 (BuildId: f0fdeb36a)
    ......
    0x555bc18 is located 88 bytes after global variable c_operations

    SUMMARY: AddressSanitizer: global-buffer-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)
```

Vanilla Agent

```
==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:2277
    #2 0x55cbc3 in Compile_If /source/m3_compile.c:1648
    #3 0x5596ec in Compile_BlockStatement /source/m3_compile.c:2207
    #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 (BuildId: f0fdeb36a)
    ......
    0x555bc18 is located 88 bytes after global variable c_operations

    SUMMARY: AddressSanitizer: global-buffer-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.

Vanilla Agent

```
==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:2277
    #2 0x55cbc3 in Compile_If /source/m3_compile.c:1648
    #3 0x5596ec in Compile_BlockStatement /source/m3_compile.c:2207
    #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 (BuildId: f0fdeb36a)
    ......
    0x555bc18 is located 88 bytes after global variable c_operations

    SUMMARY: AddressSanitizer: global-buffer-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);
```

Vanilla Agent

```
==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
    #2
    #3                                                          7
    #4
    ..
    #8
    #9                                                          b36a)
    ......
    0x555bc18 is located 88 bytes after global variable c_operations

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

To address the global overflow vulnerability in the opinfo structure, you can introduce bounds checking before dereferencing.The patch ensure that opinfo points to a valid memory location

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);
```

Vanilla Agent

```
==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
    #2
    #3
    #4
    ..
    #8
    #9                                                     b36a)
    ......
    0x555bc18 is located 88 bytes after global variable c_operations

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

To address the global overflow vulnerability in the opinfo structure, you can introduce bounds checking before dereferencing.The patch ensure that opinfo points to a valid memory location

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

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);
```

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
    #2
    #3
    #4
    ..
    #8
    #9                                                          )36a)
    ......
    0x555bc18 is located 88 bytes after global variable c_operations

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

To address the global overflow vulnerability in the opinfo structure, you can introduce bounds checking before dereferencing.The patch ensure that opinfo points to a valid memory location

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.

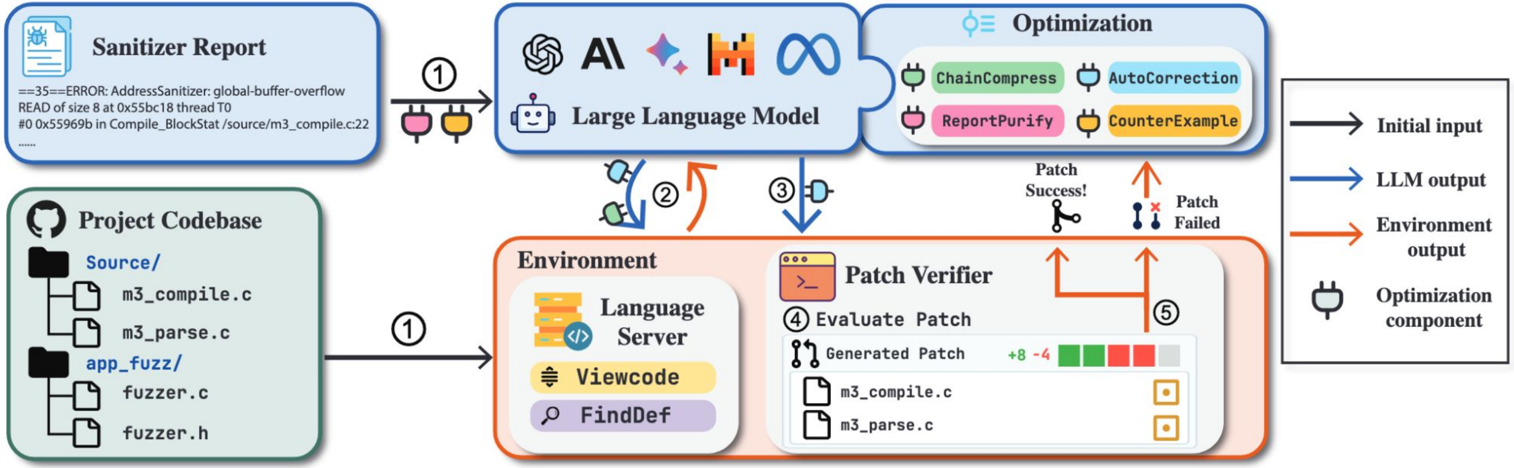
Great, now I want to view lines 17 to 20 in /source/m3_compile.c

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

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

Why not continue exploring related function?

PatchAgent



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

Common Tool Call Process

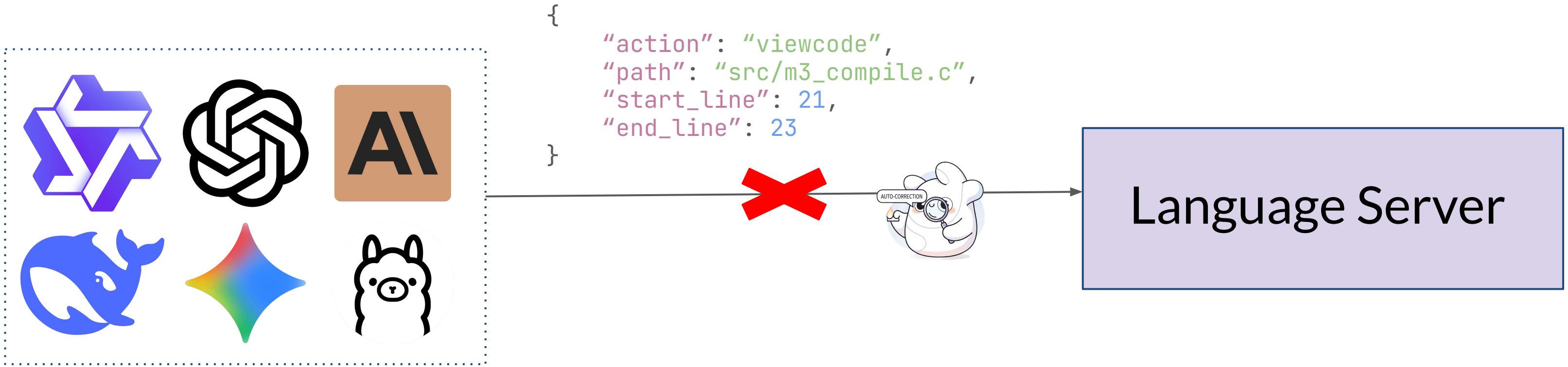
```
{
    "action": "viewcode",
    "path": "src/m3_compile.c",
    "start_line": 21,
    "end_line": 23
}
```

Language Server

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

Auto Correction

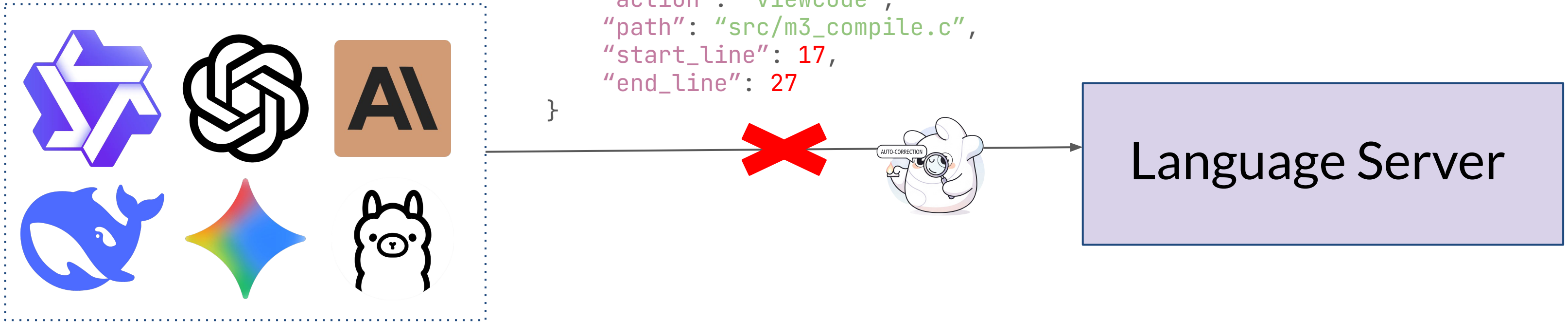
Tool Call Process
(with Auto Correction)



Auto Correction

Tool Call Process
(with Auto Correction)

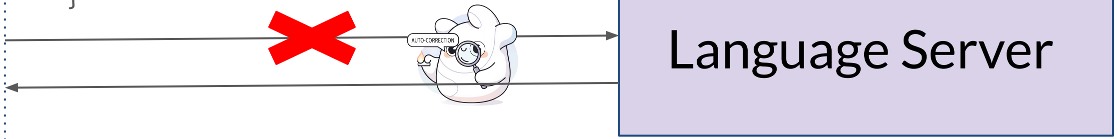
```
{
    "action": "viewcode",
    "path": "src/m3_compile.c",
    "start_line": 17,
    "end_line": 27
}
```



Auto Correction

Tool Call Process
(with Auto Correction)

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




```
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| }
```

Auto Correction

Code Viewing Enhancement

Expanded code range visibility and corrected file path references

- Resolved issues with narrow code display windows that limited readability
- Fixed incorrect file path typos that were breaking navigation links

Symbol Definition Resolution

Enhanced alias symbol detection and lookup

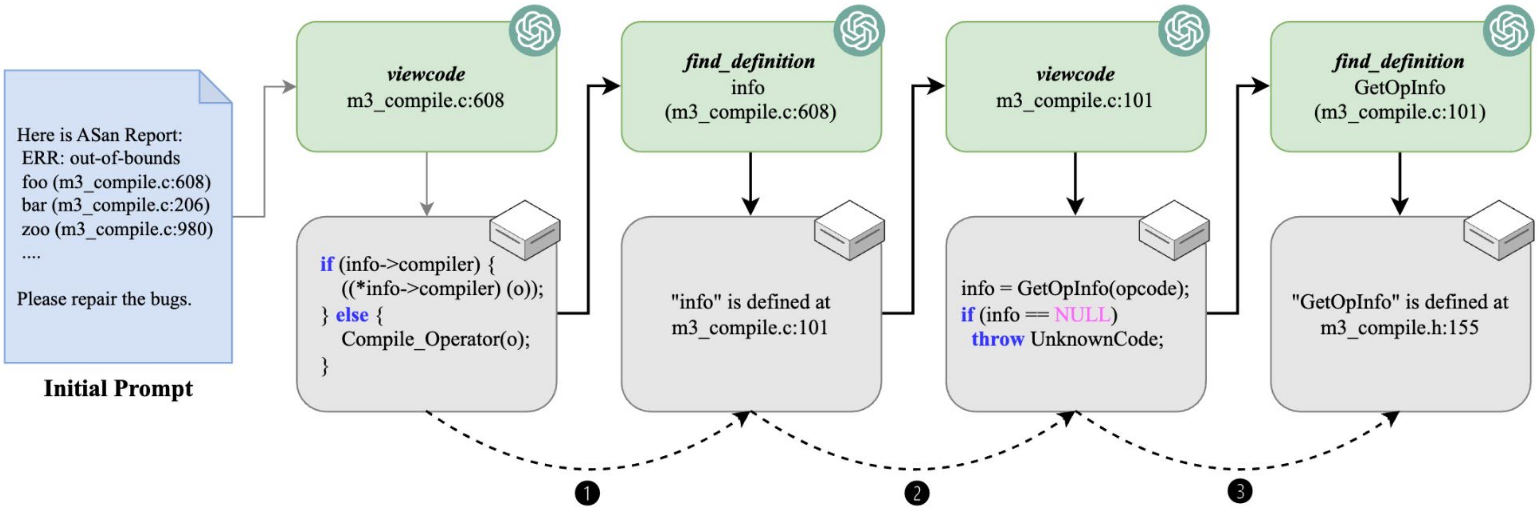
- Improved the "Find Definition" feature to properly handle aliased symbols
- Resolved cases where symbol aliases were not being recognized or linked correctly

Validation System Updates

Standardized patch format processing

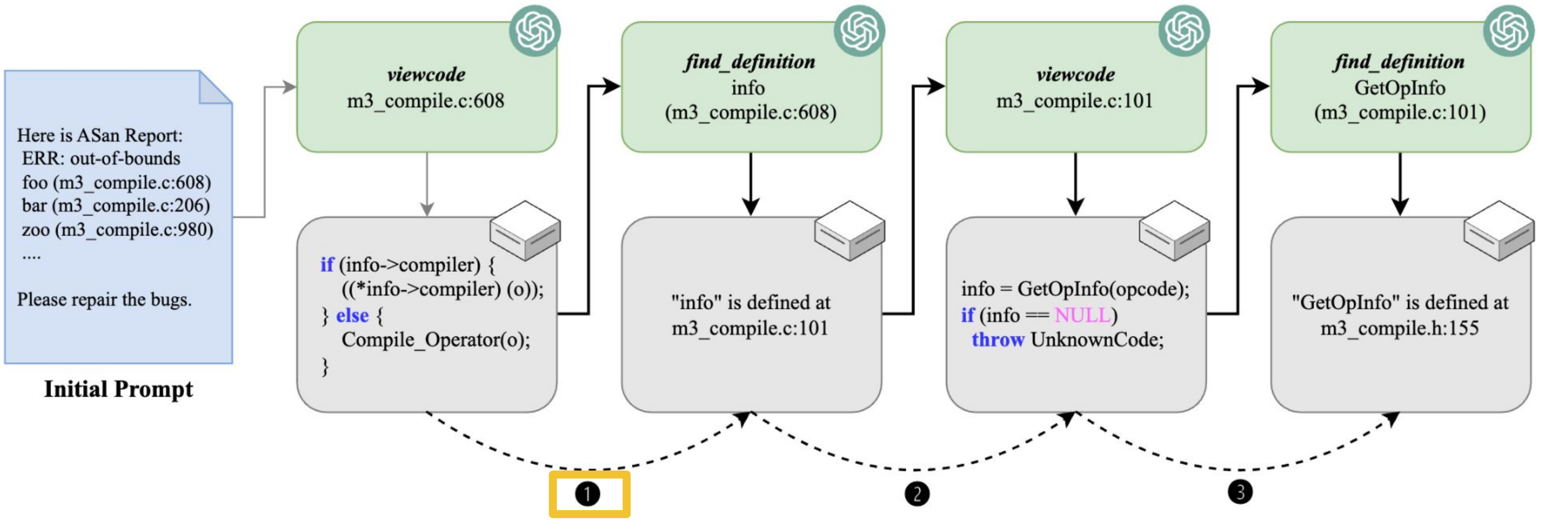
- Fixed validation errors related to inconsistent patch formats
- Ensured all patch submissions now follow proper formatting standards

Chain Compression



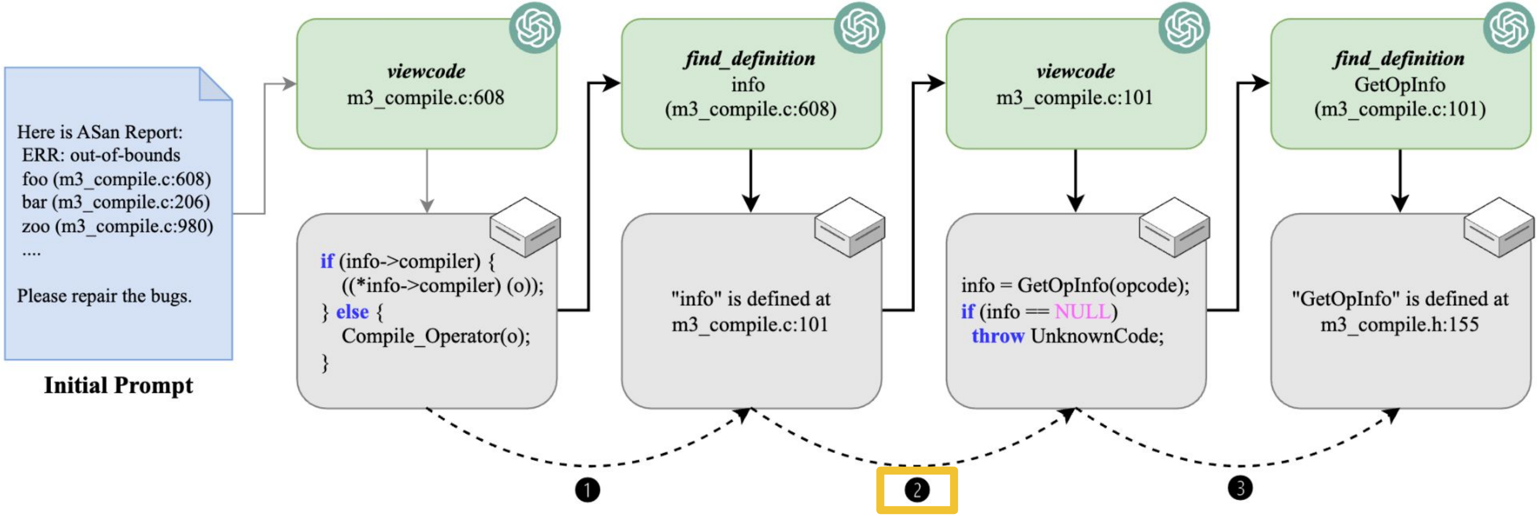
The LLM takes the initial prompt as input and starts interacting with the language server. The black bold arrows illustrate the interaction without chain compression, while the black dashed arrows represent the **compressed interaction process**. The original interaction **chain of length four** was compressed into a **single interaction**.

Chain Compression



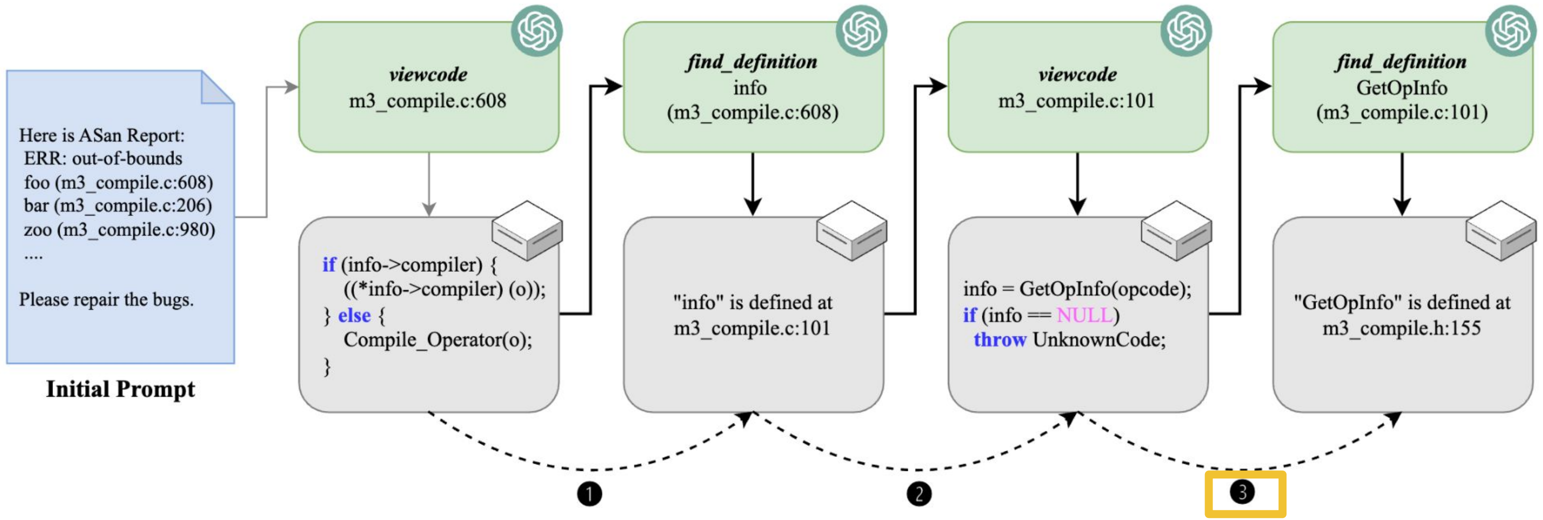
1. After the LLM sends a viewcode action, the mechanism determines that the crash is caused by the dereference of info and the line where info located **appears in both the viewed code snippet and the sanitizer report.** This indicates that it is a valuable symbol to explore.

Chain Compression



2. Using only the find_definition action to locate the definition of info is **insufficient to reveal its complete information.** Therefore, the mechanism first generates another viewcode action to obtain the definition code snippet of info.

Chain Compression



3. Then, it identifies that **the variable relies on another symbol**, GetOpInfo, and **recursively** finds its definition location.

Dataset Overview

We created a dataset comprising 178 cases sourced from OSS-Fuzz , Huntr and ExtractFix on 9 distinct bug types: stack overflow, heap overflow, integer overflow, use-after-free, double free, global overflow, divide by zero, invalid free, and null dereference.

Project	Lang	Source	LoC	#Vulns	#Test
assimp	C++	OSS-Fuzz	347.0K	3	474
c-blosc	C	OSS-Fuzz	88.8K	2	1643
c-blosc2	C++	OSS-Fuzz	117.1K	7	1284
h3	C	OSS-Fuzz	17.2K	1	124
hoextdown	C	OSS-Fuzz	7.1K	1	83
hostap	C	OSS-Fuzz	438.0K	4	19
htslib	C	OSS-Fuzz	66.5K	1	159
hunspell	C++	OSS-Fuzz	83.9K	11	128
irssi	C	OSS-Fuzz	64.4K	3	5
krb5	C	OSS-Fuzz	301.6K	1	125
libplist	C	OSS-Fuzz	12.1K	3	34
libsndfile	C	OSS-Fuzz	56.4K	5	141
libtpms	C	OSS-Fuzz	115.0K	1	6
libxml2	C	OSS-Fuzz	200.4K	10	3272
lz4	C	OSS-Fuzz	18.6K	2	22
md4c	C	OSS-Fuzz	8.0K	4	24
openexr	C++	OSS-Fuzz	227.8K	3	111
sleuthkit	C	OSS-Fuzz	196.2K	5	2
wasm3	C	OSS-Fuzz	22.8K	8	35062
zstd	C	OSS-Fuzz	93.4K	5	28
gpac	C	Huntr	743.7K	32	711
libmobi	C	Huntr	19.1K	5	12
mruby	C	Huntr	62.2K	10	61
radare2	C	Huntr	841.3K	20	858
yasm	C	Huntr	132.1K	3	44
binutils	C	ExtractFix	666.8K	2	20
coreutils	C	ExtractFix	86.1K	4	573
jasper	C	ExtractFix	44.7K	2	16
libjpeg	C	ExtractFix	46.9K	4	530
libtiff	C	ExtractFix	85.9K	11	74
libxml2	C	ExtractFix	200.4K	5	3272

Effectiveness Evaluation

Model	Temporal Error	Spatial Error	Null Dereference	Numeric Error	Total
GPT-4o	13/23 (56.52%)	96/125 (76.80%)	23/23 (100.00%)	7/7 (100.00%)	139/178 (78.09%)
GPT-4 Turbo	11/23 (47.83%)	87/125 (69.60%)	21/23 (91.30%)	7/7 (100.00%)	126/178 (70.79%)
Claude-3 Opus	14/23 (60.87%)	108/125 (86.40%)	22/23 (95.65%)	7/7 (100.00%)	151/178 (84.83%)
Claude-3 Sonnet	8/23 (34.78%)	77/125 (61.60%)	17/23 (73.91%)	6/7 (85.71%)	108/178 (60.67%)
Claude-3 Haiku	9/23 (39.13%)	93/125 (74.40%)	19/23 (82.61%)	7/7 (100.00%)	128/178 (71.91%)
Union	20/23 (86.96%)	114/125 (91.20%)	23/23 (100.00%)	7/7 (100.00%)	164/178 (92.13%)

This table compares the effectiveness of **PatchAgent** when utilizing different LLMs to repair vulnerabilities. The **Union** row represents the combined results of **PatchAgent** across all models, demonstrating the overall improvement in repair accuracy achieved through the collaborative use of multiple models.

Github Pull Requests

Fix vuln OSV-2024-947 #1699

Merged seladb merged 8 commits into `seladb:dev` from `oss-patch:patch-OSV-2024-947` on Mar 29

Fix vuln OSV-2023-77 #5210

Merged lrknox merged 7 commits into `HDFGroup:develop` from `oss-patch:patch-OSV-2023-77` on Jan 15

Fix buffer overflow in MD3Loader #5763

Merged kimkulling merged 2 commits into `assimp:master` from `cla7aye15I4nd:fix-5616` on Sep 10, 2024

kex: fix null pointer dereference in `diffie_hellman_sha_algo()` #1508

Merged willco007 merged 1 commit into `libssh2:master` from `oss-patch:patch-crash-c7c664759c840b3f1c438b7232f713b756ed640f` on Feb 28

Fix Heap-buffer-overflow in __parse_options #1678

Merged tigercosmos merged 5 commits into `seladb:dev` from `oss-patch:patch-crash-7d18f37e1f05e0ff4aa4dfa2f67dd738340ad9cf` on Feb 2

Fix vuln OSV-2024-390 #5201

Merged lrknox merged 1 commit into `HDFGroup:develop` from `oss-patch:patch-OSV-2024-390` on Jan 3

Fix vuln OSV-2024-384 #1061

Merged rurban merged 1 commit into `LibreDWG:master` from `oss-patch:patch-OSV-2024-384` on Dec 24, 2024

Fix vuln OSV-2024-343 #1680

Merged seladb merged 2 commits into `seladb:dev` from `oss-patch:patch-OSV-2024-343` on Jan 11

LIBSSH2

HDF5

LibreDWG

PcapPlusPlus

Proposed work:

Explore the Problem of AI-Generated Patch

Issues with Submitting Patches in the Real World

seladb on Mar 20 — Functional Issue — Owner

The right solution here is to disallow creating this layer if `m_DataLen < sizeof(ppp_pptp_header)`.

I think this layer is quite old, but at some point, we introduced the concept of `isDataValid()` which is implemented per layer, accepts the raw data, and decides if the layer can be created. We should implement this method and use it when the layer is created:

qkoziol left a comment — Incorrect Root Cause — Collaborator

These are not bad, but seem like they are addressing symptoms rather that root causes. Are there test files that trigger them to look at?

jhendersonHDF on Jan 2 • edited — Security Issue — Collaborator

This fixes the specific vulnerability, but a more complete fix may be needed here. Consider the case where both `overlap_size` and `new_accum_size` end up slightly below `accum->alloc_size`. It may make more sense to calculate a pointer to the last valid byte in `accum->buf` and then make use of the `H5_IS_BUFFER_OVERFLOW` macro from H5private.h.

Revisit the Workflow of Program Repair

Fault Localization

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

Patch Generation

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

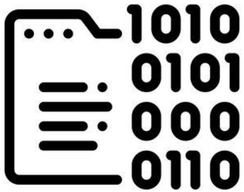
Patch Validation

Verify that a patch addresses vulnerabilities while maintaining functional integrity.

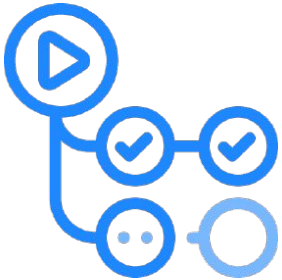
Revisit the Workflow of Program Repair

~~Patch Validation~~

Verify that a patch addresses vulnerabilities while maintaining functional integrity.



Replay the PoC



Rerun Functional Test
(e.g., Github CI)

Works Using Test Suite-Based Validation Method

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

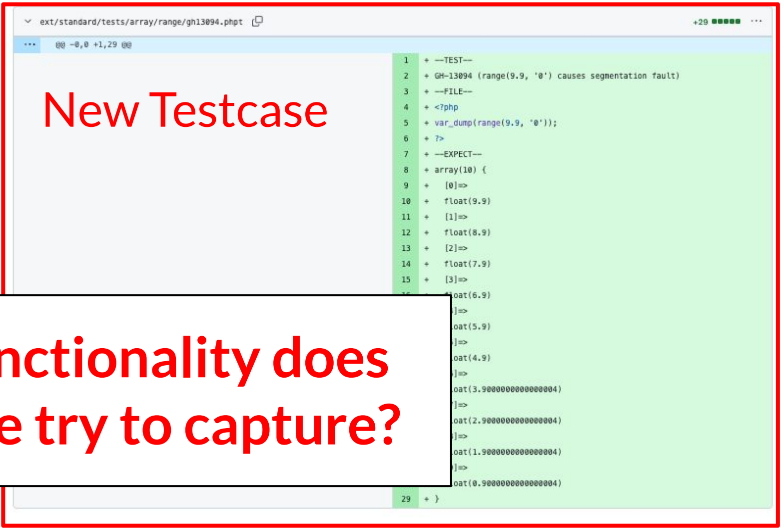
Obervations from a PHP Case

- Existing functional tests (e.g., CI) are not able to capture full functionality.

- Developers may upgrade functional tests during vulnerability repair.

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

Obervations from a PHP Case



- Existing functional tests (e.g., CI) are not able to capture full functionality.

- Developers may upgrade functional tests during vulnerability repair.

What kind of functionality does the new test case try to capture?

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

Vulnerability Principle

```c
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)
```

PHP Sandbox

```php
<?php
var_dump(range(1, 10, 2));
?>
```

```
array(5) {
  [0]=>
  int(1)
  [1]=>
  int(3)
  [2]=>
  int(5)
  [3]=>
  int(7)
  [4]=>
  int(9)
}
```

Vulnerability Principle

```
1  PHP_FUNCTION(range) {
2    struct zval *user_start = /* Extract start argument */;
3    struct zval *user_end = /* Extract end argument */;
4
5    // Extract type from arguments
6    uint8_t start_type = Z_TYPE_P(user_start);
7    uint8_t end_type = Z_TYPE_P(user_end);
8
9    /* If the range is given as strings,
10      generate an array of characters. */
11   if (start_type >= IS_STRING || end_type >= IS_STRING) {
12     // VULNERABLE: condition fails when start_type=5 (IS_DOUBLE)
13     // and end_type=7 (IS_ARRAY) because 5+7 = 2*6 (IS_STRING)
14     if (start_type + end_type < 2*IS_STRING) {
15       // ... handle mixed type inputs and convert to numeric
16       goto handle_numeric_inputs;
17     }
18
19     // TYPE CONFUSION OCCURS HERE:
20     //   When the vulnerable condition fails, we reach this point
21     //   with non-string types, but try to access them as strings
22     unsigned char low = Z_STRVAL_P(user_start)[0];
23     unsigned char high = Z_STRVAL_P(user_end)[0];
24
25     // ... character range generation logic
26     return;
27   }
28
29   handle_numeric_inputs:
30   if (start_type == IS_DOUBLE || end_type == IS_DOUBLE) {
31     // ... process numeric ranges (floats)
```

The bug arises when the function receives arguments of specific type combinations, such as a double (floating-point number) and an array. The first condition (line 11) correctly identifies that at least one argument appears to be string-like since $\texttt{IS\_ARRAY}(7) \geq \texttt{IS\_STRING}(6)$, so the code enters the string-handling branch. However, the inner arithmetic condition (line 14) unexpectedly fails when for example, $\texttt{IS\_DOUBLE}(5) + \texttt{IS\_ARRAY}(7) = 12$. When this condition fails, the code skips the proper type conversion logic and incorrectly attempts to access the non-string data using string accessor macros (line 22), which cause the program crash.

Developer Thinkings (“New TestCase”)

Develop Patch

```
@@ -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 (start_type + end_type < 2*IS_STRING) {
+  if (start_type < IS_STRING || end_type < IS_STRING) {
     if (start_type < IS_STRING) {
       if (end_type != IS_ARRAY) {
         php_error_docref(NULL, E_WARNING, "...");
```

New Testcase

```
--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)
  [3]=>float(6.9)
  [4]=>float(5.9)
  [5]=>float(4.9)
  [6]=>float(3.9000000000000004)
  [7]=>float(2.9000000000000004)
  [8]=>float(1.9000000000000004)
  [9]=>float(0.9000000000000004)
}
```

PHP Specification

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.

Developer Patch vs LLM Patch

```
@@ -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 (start_type + end_type < 2*IS_STRING) {
+  if (start_type < IS_STRING || end_type < IS_STRING) {
     if (start_type < IS_STRING) {
       if (end_type != IS_ARRAY) {
         php_error_docref(NULL, E_WARNING, "...");
```

a. Developer Patch

```
@@ -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];
```

b. LLM Patch

```
--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)
  [3]=>float(6.9)
  [4]=>float(5.9)
  [5]=>float(4.9)
  [6]=>float(3.9000000000000004)
  [7]=>float(2.9000000000000004)
  [8]=>float(1.9000000000000004)
  [9]=>float(0.9000000000000004)
}
```

c. Developer's Testcase

LLM Patch violates PHP specification

```
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
```

d. Output (w. LLM Patch)

Research Question

- **RQ1:** Does test suite-based validation <u>substantially</u> overestimate program repair system performance?

- **RQ2:** How reliable is using <u>“new test cases”</u> to validate patches generated by current program repair systems?

- **RQ3:** How to measure the quality of a testcase for validating a patch?

Research Plan

- **Benchmark**: Develop a benchmark to evaluate whether current test suite-based validation methods significantly overestimate the effectiveness of program repair tools.

- **Positive Case Anaylsis:** Compare patches that successfully pass the new benchmark against actual developer-written patches to assess benchmark reliability and identify potential limitations in evaluation methodology.

- **Testcase Measurement:** Find metrics to measure the quality and reliability of test cases for software functionality.

Why Should It Works?

- **Solid Observation:** I observed a large number of AI-generated patches that differ significantly from developer patches in previous projects, and developers have also provided negative feedback.

- **Preliminary Analysis:** I have obtained some preliminary analysis results on PHP projects, which reflect the assumptions of our proposed work.

- **Rich Experience:** My previous research has provided me with considerable experience and expertise in determining patch correctness.