What IF Is Not Enough? Fixing Null Pointer Dereference With Contextual Check

Yunlong Xing
George Mason University

Shu Wang
George Mason University

Shiyu Sun
George Mason University

Xu He
George Mason University

Kun Sun
George Mason University

Qi Li
Tsinghua University

Abstract

Null pointer dereference (NPD) errors pose the risk of unexpected behavior and system instability, potentially leading to abrupt program termination due to exceptions or segmentation faults. When generating NPD fixes, all existing solutions are confined to the function level fixes and ignore the valuable intraprocedural and interprocedural contextual information, potentially resulting in incorrect patches. In this paper, we introduce CONCH, a novel approach that addresses the challenges of generating correct fixes for NPD issues by incorporating contextual checks. Our method first constructs an NPD context graph to maintain the semantics related to patch generation. Then we summarize distinct fixing position selection policies based on the distribution of the error positions, ensuring the resolution of bugs without introducing duplicate code. Next, the intraprocedural state retrogression builds the if condition, retrogresses the local resources, and constructs return statements as an initial patch. Finally, we conduct interprocedural state propagation to assess the correctness of the initial patch in the entire call chain. We evaluate the effectiveness of CONCH over two real-world datasets. The experimental results demonstrate that CONCH outperforms the SOTA methods and yields over 85% accurate patches.

1 Introduction

Null Pointer Dereference (NPD) occurs when a program attempts to access memory at a null pointer, and dereferencing a null pointer always causes the program to collapse. If an NPD occurs in a crucial system component, e.g., a kernel module or device driver, it can lead to a system-wide crash [11, 39]. Additionally, NPDs can be exploited by attackers to carry out malicious actions, such as executing unauthorized code or launching a denial-of-service attack [3, 4]. Therefore, NPDs are among the top prevailing security issues in the production environment [8, 9].

Several efforts have been made to patch NPD vulnerabilities [6, 12, 14, 15, 46–48]. Most of them follow the conventional generate-and-validate repair steps. They first traverse the search space using classic search algorithms like genetic programming or random search and then validate the generated patches using test cases. As the SOTA solution for fixing NPDs, VFix [48] localizes suspicious statements by using static value-flow analysis and dynamic test case driven triggering. Then VFix reduces the search space using the predefined fixing patterns, i.e., reallocating a new memory space or adding an if check, to improve the efficiency of the patch process and increase the chances of finding correct patches.

However, all existing solutions are confined to the function level fixes and ignore valuable intraprocedural and interprocedural contextual information when generating NPD fixes. First, the overlook of intraprocedural contextual information may cause overdue occupation of system resources such as memory and locks, denying access to those critical system resources as intended. Second, there is no interprocedural analysis for fixing NPD bugs, disregarding global variable resetting, function argument resetting, and patch validation in the call chain. For instance, it is crucial to reset some global variables and function arguments; otherwise, the program's state could be misidentified [25]. It remains a challenge to correctly fix NPD errors with contextual information.

In this paper, we propose an NPD contextual check mechanism named CONCH for fixing NPD errors with contextual information. By assessing the specific context in which the errors occur, CONCH can effectively generate the NPD fixes for scenarios that insert if checks, which are the most popular cases in real-world vulnerability fixing [44]. To construct accurate NPD patches that work within the function and among the entire call chain, we address three challenges.

The first challenge lies in selecting the appropriate repair position that ensures the correct and efficient repair without redundant patching. VFix [48] uses path congestion to select the repair position; however, its vulnerability path selection relies on test case selection and the dynamic triggering of test cases. We propose a vulnerability fix tailored graph named NPD Context Graph, which only maintains the semantics related to patch generation. In the NPD context graph, we identify the null and error positions and their corresponding paths. Then

we summarize four fixing position selection policies based on the distribution of the null and error positions, ensuring the resolution of bugs without duplicate fixes.

The second challenge involves constructing an initial patch via intraprocedural state retrogression to guarantee the correctness of the generated patch within the local function. This includes building the if condition, retrogressing the local resources, and constructing the return statement. When generating the if condition, we focus on the callee function at the null position. We analyze the implementation of the callee function to determine the exception value it returns when it encounters a failure. This exception value is crucial to constructing the condition for the if check in the current erroneous function. To retrogress the local resources, we have to deal with the diverse naming conventions of memory allocation/deallocation and locking/unlocking functions. To the best of our knowledge, there is no existing research that specifically addresses freeing the allocated memory and releasing the occupied lock during NPD fixing. To retrogress the local resources, we initiate by extracting keywords such as "alloc" and "lock" from the kernel and general libraries. This extraction helps create a dictionary of paired functions. Subsequently, we identify any missing pairs for allocated memory and occupied locks within the buggy function. When constructing a correct return statement, we consider the return type and other return statements in the current function. If the error occurs within a loop, we ensure the function can continue or break the loop without exiting the entire program.

The third challenge entails conducting the interprocedural state propagation, including resetting the global variables/-function arguments and assessing the patch correctness in the call chain. To reset the global variables and function arguments, we first identify which variables should be reset. The search scope is from the function entry to the error position, and we exclude the local variables. Then we determine the value to be reset by referring to the existing error-handling statement within the current function and conducting data flow analysis to infer the condition in the caller that represents a failure state. To assess the NPD fixing handled in the entire call chain, we analyze the return type of the callee function. If the return type is `void`, it returns no value to its caller function. In such a case, if the caller function executes normally after the buggy function returns when detecting a null pointer, we recursively analyze the call chain from the buggy function to the function that handles the error and update the intermediate functions according to the error-handling semantics. If the return type is not `void`, it has a return value to its caller function. Then, we check if the caller handles the return value correctly. If not, we update the caller.

We implement a prototype of CONCH and evaluate its contextual correctness for NPD fixes using two real-world datasets. The first dataset includes 80 NPD vulnerabilities from MITRE [30] and CONCH can generate 68, with an accuracy of 85%, semantic equivalent patches as submitted by the developer, outperforming an accuracy of 26.25% compared to the SOTA approach. The second dataset comes from a well-known benchmark suite [42], containing 18 NPD programs. CONCH can generate 16 correct patches, while the SOTA approach can only generate 12 correct patches.

In summary, we make the following contributions:

- We propose to fix NPD errors with contextual checks, ensuring a more effective and complete vulnerability control throughout the entire call chain.
- We are the first to address local resource retrogression (including freeing the allocated memory and releasing the occupied lock) and reset global variable and function argument in NPD fixing.
- We implement a prototype of CONCH and conduct experiments to evaluate its effectiveness and accuracy. The experimental results show that CONCH outperforms the SOTA approach.

We will open-source our tool and release the tested vulnerability code and benchmark.

2 Background

2.1 Separation Logic

Separation Logic (SL) [38] is grounded in a set of inference rules and techniques that enable effective reasoning about memory allocation, deallocation, and relationships among different memory segments in program analysis. It provides a precise framework for examining the ownership and sharing of heap-allocated data structures and resources. Derived from Hoare Logic [43], SL inherits the fundamental notions of pre- and post-conditions, which define the required conditions before and after program execution, respectively. These concepts are unified in Hoare triples of the form $\{P\}C\{Q\}$, where P and Q represent pre- and post-conditions and C denotes an operation. The logic encompasses a collection of inference rules to facilitate reasoning about program statements, including assignments, conditionals, and loops. Additionally, SL introduces Frame Rule [38] for modular reasoning about program components. By incorporating these elements, SL is able to verify program correctness, detect bugs related to memory access, and reason about heap-related behaviors.

2.2 Incorrectness Separation Logic

Based on separation logic, several approaches were proposed to prove the absence of memory-related bugs [1,2,13], but not reasoning for bug presence. In 2020, Incorrectness Separation Logic (ISL) [37] is proposed to extend SL and catch heap-related errors. Due to the ISL principle, there are no false positives in error detection [37]. The key contribution of ISL is to provide inference rules to define memory safety violations

when accessing deallocated locations. Equation 1 shows the extended rules on NPD issues.

$$
\begin{aligned}
&\textsc{LoadErr}: \{y \not\mapsto\ \}\ x := [y]\ \{err: y \not\mapsto\ \} \\
&\textsc{LoadNull}: \{y = null\}\ x := [y]\ \{err: y = null\} \\
&\textsc{StoreErr}: \{x \not\mapsto\ \}\ [x] := y\ \{err: x \not\mapsto\ \} \\
&\textsc{StoreNull}: \{x = null\}\ [x] := y\ \{err: x = null\}
\end{aligned} \tag{1}
$$

The LOADERR rule specifies that an error occurs when attempting to dereference y that has already been deallocated. Similarly, the LOADNULL rule states that an error arises when attempting to dereference a null pointer y. The STOREERR rule indicates that an error occurs when trying to assign a value to x that has already been deallocated. Lastly, the STORENULL rule states that an error arises when attempting to assign a value to a null pointer x.

When reasoning on heap-related data structures, an NPD error is identified if a statement satisfies any of the ISL rules in Equation 1. These rules serve as the foundation for detecting NPD vulnerabilities in this paper and we select fault localization using ISL as our initial step to identify the null and error positions in an NPD program. Since SL and ISL mutually reinforce each other in detecting NPD vulnerabilities, we use SL to encompass both SL and ISL collectively.

3 Motivation Examples

Our work is motivated by the NPD fix cases where existing solutions yield incorrect security checks due to the neglect of intraprocedural/interprocedural context, leading to unreleased locks, unreset variables, or even new errors in the call chain. We illustrate three examples to demonstrate the root cause of the inappropriate NPD fixes and show how CONCH can overcome the existing limitations in intraprocedural state retrogression and interprocedural state propagation. For simplicity, we focus on error-related statements and their corresponding fixes, normalizing the function names and parameters without changing their execution logic.

3.1 Intraprocedural State Retrogression

We first show an example in which the intraprocedural state retrogresses, e.g., releasing locks, should be considered when generating NPD patches. A simplified code segment of CVE-2022-41858 is shown in Figure 1, with an NPD error due to the possible null pointer of `sl->tty` at Line 7. When constructing the repair statement, existing solutions only check if `sl->tty` is null and generate a return statement according to the `void` function type.

Fixed by CONCH. Apart from generating the if condition and return statements, we ensure resource retrogression of unlocking the occupied lock. First, we construct lock/unlock function pairs to record the locking state. Then, when analyzing the CFG from the function entry to the error position,

```
1   void buggy(param1, param2){
2       spin_lock(&sl->lock);
3 +     if(sl->tty == NULL){
4 +
5 +         return;
6 +     }
7       function(sl->tty, … );
8   }
```

(a) Fixed by SOTA work

```
1   void buggy(param1, param2){
2       spin_lock(&sl->lock);
3 +     if(sl->tty == NULL){
4 +         spin_unlock(&sl->lock);
5 +         return;
6 +     }
7       function(sl->tty, … );
8   }
```

(b) Fixed by CONCH

Figure 1: An example of the necessity of unlocking an occupied lock (CVE-2022-41858).

we match the functions in the pairs to check if further unlock operation is required. If so, we extract the paired unlock function and insert it before the return statement, as shown in the shadow rectangle of Figure 1(b).

3.2 Interprocedural State Propagation

We use two examples to show that the interprocedural state propagation (e.g., global variable resetting, function argument resetting, and entire call chain assessment) has been overly neglected by existing approaches. We also explain how our solution can correctly generate the fixes.

3.2.1 Function Argument Resetting

Figure 2 depicts an example where the patch generation is affected by the restriction of only returning a single value. In the C programming language, when a function necessitates multiple return values, since only one value can be conveyed through the return statement, the remaining values are transmitted via function arguments. However, in existing solutions, the return values passed through the arguments are often neglected and need to be reset.

The simplified code segment of CVE-2022-2153 is shown in Figure 2(a), where the function argument `*r` in function `buggy` needs to be reset before returning since `*r` value will be passed to the `caller` function as its return value. The resetting operation of `*r` value can rectify the function state. However, no existing fixing methods address this issue.

Fixed by CONCH. To fix this issue, the key is to obtain the expected `*r` value before returning. By constructing the call graph of the function `buggy` and analyzing the data flow of function argument `*r`, we can determine that the `*r` value is further returned by `caller` function as an if condition in the function `caller_caller`. Specifically, if `*r` is equal to 0 that signifies a failure state, the function `schedule_work` will not be called. Conversely, if `*r` has a non-zero value, the `schedule_work` function will be invoked to conduct a scheduling task. To generate a correct patch, it is essential to reset the value of `*r` to 0 before leaving the security check, as depicted by the shadow rectangle in Figure 2(b). This resetting step ensures the program behaves as intended with the NPD vulnerability mitigated.

```
 1 bool buggy(int *r, … ){
 2     *r = -1;
 3     if(condition1){
 4 +       if (src == NULL){
 5 +
 6 +           return true;
 7 +       }
 8         // return 0 if discarded
 9         *r = func(src->vcpu, …);
10         return true;
11     }
12 }
13 int caller( … ){
14     int r = -1;
15     if(buggy(&r, … ))
16         return r;
17 }
18 int caller_caller( … ){
19     if(caller( … ))
20         schedule_work();
21 }
```

(a) Fixed by SOTA work

```
 1 bool buggy(int *r, … ){
 2     *r = -1;
 3     if(condition1){
 4 +       if (src == NULL){
 5 +           *r = 0;
 6 +           return true;
 7 +       }
 8         // return 0 if discarded
 9         *r = func(src->vcpu, …);
10         return true;
11     }
12 }
13 int caller( … ){
14     int r = -1;
15     if(buggy(&r, … ))
16         return r;
17 }
18 int caller_caller( … ){
19     if(caller( … ))
20         schedule_work();
21 }
```

(b) Fixed by CONCH

Figure 2: An example of the necessity of resetting an argument in a function (CVE-2022-2153).

3.2.2 Call Chain Assessment

Figure 3 shows an example of how an NPD error should be handled in the call chain. The simplified code segment of CVE-2022-3112 has an NPD error at Line 7. Since the failure of `kzalloc` returns null to the pointer `new_ts`, dereferencing the null pointer triggers the NPD error. However, the function `caller` is unaware of this error as the `buggy` function of type `void` does not return any value to the `caller`. It leads to the uninterrupted execution of the `caller`, failing to handle this error appropriately. A correct patch requires analyzing the code implementation of `caller_caller` since it encompasses the value scope when `caller` encounters a failure. However, existing approaches can only construct fixing within the current function by checking if `new_ts` is null and returning directly, considering the `void` return type.

```
 1 void buggy(param1, param2,… ){
 2
 3     struct *new_ts;
 4     new_ts = kzalloc(sizeof());
 5 +   if(new_ts == NULL)
 6 +       return;
 7     new_ts->ts = ts;
 8 }
 9 int caller(param1, param2){
10     int ret;
11     if(error)
12         return -EAGAIN;
13     buggy( … );
14
15
16
17     …
18     return 0;
19 }
20 void caller_caller(param) {
21     if(caller( … ) < 0)
22         break;
23 }
```

(a) Fixed by SOTA work

```
 1 - void buggy(param1, param2,… ){
 2 + int buggy(param1, param2, … ){
 3     struct *new_ts;
 4     new_ts = kzalloc(sizeof());
 5 +   if(new_ts == NULL)
 6 +       return -ENOMEM;
 7     new_ts->ts = ts;
 8 }
 9 int caller(param1, param2){
10     int ret;
11     if(error)
12         return -EAGAIN;
13 -   buggy( … );
14 +   ret = buggy( … );
15 +   if(ret)
16 +       return ret;
17     …
18     return 0;
19 }
20 void caller_caller(param) {
21     if(caller( … ) < 0)
22         break;
23 }
```

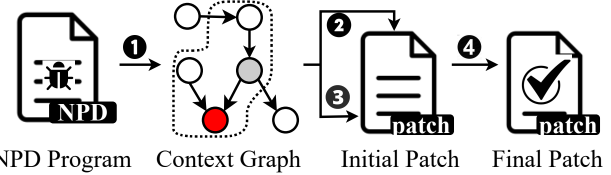
(b) Fixed by CONCH

Figure 3: An example of the necessity of handling an NPD error in the call chain (CVE-2022-3112).

Fixed by CONCH. To fix this NPD error, we first conduct intraprocedural analysis to generate an initial patch as shown in Figure 3(a). Here, the `caller` does not have error-handling statements, and the `caller_caller` handles the error when `caller` returns a negative value. To fix the NPD error in the call chain, the `caller` needs to return a negative value, while the `buggy` should have an explicit return value for the `caller`. Thus, we generate the patch in three steps. First, we modify the return type of function `buggy` from `void` to `int`, facilitating the `caller` to handle the error. Second, we add a security check so that the `buggy` returns a negative integer once the `new_ts` is null. Third, when the `caller` receives the negative integer, it returns the negative integer to the `caller_caller`. The generated patch is shown in Figure 3(b).

4 CONCH Design

CONCH consists of four steps to fix NPD errors with contextual checks, as shown in Figure 4. Note that CONCH focuses on the scenarios where the NPD errors can be fixed by adding if statement checks.



NPD Program Context Graph Initial Patch Final Patch

Figure 4: The CONCH overview: ❶ NPD context graph construction; ❷ path-sensitive fixing position selection; ❸ intraprocedural state retrogression, and ❹ interprocedural state propagation.

Given an NPD program, we first generate an NPD Context Graph to preserve the semantics related to patch generation by including the simplified control flow graph (CFG) of the vulnerable function and the calling pattern of the caller and callee functions. The context graph serves as a prerequisite for the subsequent components. Second, we define four policies to select the fixing position based on the distribution of the null position and the error positions. Given the policies, we overcome the challenge of selecting the appropriate repair point and we ensure the correct repair of vulnerabilities without introducing redundancy patches. Third, the intraprocedural state retrogression aims to generate an initial patch, ensuring that the NPD error is correctly handled in the current function. This step performs the if condition construction, local resource retrogression, and return statement construction. Fourth, we adopt the interprocedural state propagation to obtain the final patch, ensuring that the generated patch handles the NPD errors appropriately in the entire call chain. This step addresses global variable resetting, function argument resetting, and the entire call chain assessment.

4.1 NPD Context Graph Construction

For a given NPD program, we can generate an NPD context graph in three steps, as shown in Figure 5.

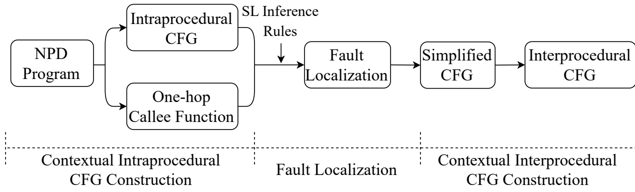


Figure 5: Context graph construction for NPD program.

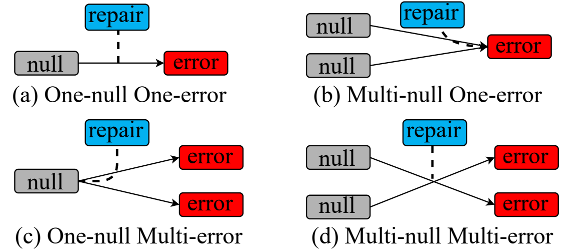


Figure 6: 4 policies of path-sensitive fixing position selection.

Contextual Intraprocedural CFG Construction. We construct intraprocedural CFGs for all functions in the NPD program, as well as the one-hop callee functions. The reasons that we only use one-hop of callee information are two-fold. First, if we consider all callee functions, the recursive tracing can lead to path explosion, while most of them are not necessary after processing. Second, according to real-world vulnerability analysis, the one-hop callee function can cover most cases in inferring the error positions. In some cases where one-hop callee function analysis is insufficient to obtain the required context, we conduct incremental analysis to analyze more hops until the error position is obtained. Additionally, the indirect control flow may hinder the connection of the callee function, and we adopt the SOTA approach MLTA [20] to refine the indirect targets. The generated contextual intraprocedural CFGs extend the contextual information, providing a prerequisite of call relations and path constraints for localizing error positions.

Fault Localization. We rely on the separation logic and its inference rules in Equation 1 to locate the NPD errors. We obtain the null positions where the pointers are set to null and the error positions where the null pointers are dereferenced to trigger errors. Separation logic is utilized to analyze different execution paths, which are identified explicitly by the intraprocedural CFG within the function. When a statement dereferences a null or invalid pointer, an NPD error is detected. In such cases, the null positions, the error positions, and their involved execution paths are identified and recorded, enabling precise error localization within the function and facilitating the subsequent NPD mitigation.

Contextual Interprocedural CFG Construction. After obtaining the contextual intraprocedural CFGs of all functions, along with the null and error positions and their corresponding paths, we construct the interprocedural CFG by streamlining the intraprocedural CFGs. We first prune the intraprocedural CFG of the buggy function and only retain the nodes of the path from the function entry to the error position. Then, for the callee function containing the null position, we preserve the error-handling statements and their return values in their nodes. Next, we analyze the caller function of the buggy function and maintain the statements from the function entry to the calling position of the buggy function. For special cases where the direct caller cannot cover the error handling, we conduct an incremental analysis to recursively trace deeper callers. Finally, we connect these pruned CFGs together to generate an interprocedural CFG, containing all the states related to the error handling while removing irrelevant statements.

4.2 Path-sensitive Fixing Position Selection

Based on the distribution of the null and error positions, we define four different policies to select path-sensitive fixing positions. As the SOTA approach, VFix [48] proposes a congestion calculation to select a fixing position; however, it has two limitations. First, the dynamic test case based localization limits its scalability. Second, this congestion calculation method may lead to ambiguity since all sequential statements in the key path have the same congestion result. Therefore, to select a proper fixing position, we analyze all the NPD records containing commit information in MITRE [33] and summarize four fixing position selection policies according to different control flows, i.e., one null and one error positions, multiple null and one error positions, one null and multiple error positions, and multiple null and multiple error positions, as shown in Figure 6. Our key principle of fixing position selection is that the fixing position should be between the null position and the error position, and repeated fixing should be avoided.

The first type (i.e., one null position and one error position) of fixing position selection is straightforward and widely occurs. When the null position and error position are on the same path without any branches, the typical fixing position is closely behind the null position. This approach ensures that the fix is placed between the null position and the error position, diminishing potential derivative bugs. This fixing pattern can be found in addressing the real-world vulnerabilities in CVE-2022-3112 [26] and CVE-2022-41858 [28].

The second type (i.e., multiple null positions and one error position) of fixing position selection arises when the null positions are located within multiple if branches, as shown in CVE-2023-1355 [32]. Instead of selecting an individual fixing position for each branch, we select the fixing position just prior to the error position to avoid repeated fixes. This strategy allows us to check errors for both branches without introducing redundant code.

The third type (i.e., one null position and multiple error positions) of fixing position selection is similar to the second type except that there is only one null position but multiple

error positions within the if branches. This fixing pattern appears in CVE-2022-3153 [27]. Though it is possible to select an individual fixing position for each error position, it may result in redundant fixing since the fixing statements are identical. To avoid redundancy, we select the fixing position immediately after the null position. This approach allows us to address both error positions while avoiding unnecessary duplication of the fixing process.

The last type (i.e., multiple null and error positions) of fixing position selection is the most complex. A real fixing example can be found in the `JFreeChart` project [48]. To determine the fixing position in such cases, we can consider the null positions as one node and the error positions as another node. The fixing position can be identified as the node through which the flow from all null positions intersects with the flow toward all error positions. This approach ensures that the fixing position captures the necessary context and effectively addresses both the null and error positions.

In general, the first type is the most general and popular case and the other types have more constraints compared to the first type, thus, we select different fixing position selection policies.

4.3 Intraprocedural State Retrogression

We use the intraprocedural state retrogression to generate an initial patch, which includes if condition construction, local resource retrogression, and return statement construction.

4.3.1 If Condition Construction

The if check is an effective and widely used manner in NPD fixes. When constructing the if conditions, a general format would be *if* a checked variable is equal to null or an exception value, *then* the function or loop is terminated. An alternative format is to determine *if* the checked variable is not equal to the exception value, *then* the existing code will be executed. Here, the challenges are how to select the checked variable and how to determine the exception value. Also, after obtaining the variable and value to be checked, we need to choose either the null checking to terminate the code or the non-null checking to execute the existing code.

To solve this issue, we first analyze the existing patches for previous CVEs and summarize three representative patch templates. In Figure 7(a), the variable `pcpu_sum` would be null once the allocation fails in the function `kvmalloc_array`; in this case, dereferencing `pcpu_sum` triggers an NPD error. To fix this issue, null checking is conducted for `pcpu_sum`, and the function terminates if it is null. In Figure 7(b), function `amvdec_add_ts` returns a negative exception value when it fails, and the returned value makes the received variable an invalid pointer. To address this vulnerability, the return value of `amvdec_add_ts` needs to be assigned to a state variable `ret` and check if `ret` is a non-zero value. While in Figure 7(c), variable `info->st_info_list` does not represent a return value, and the following code has a data flow dependency on this variable. If we check whether the variable is null and return when it is null, the other statements of this function, i.e., freeing a function argument, cannot be completed. Therefore, the generated patch executes the original statements when `info->st_info_list` is not null.

```
1     pcpu_sum = kvmalloc_array(param1, param2, param3)
2 +   if(pcpu_sum == null)
3 +       return;
4     this_sum = &pcpu_sum[cpu];
```

(a) CVE-2022-3107

```
1 -   amvdev_add_ts( … );  // return neg when fails
2 +   int ret = amvdec_add_ts( … );
3 +   if(ret)
4 +       return ret;
```

(b) CVE-2022-3112

```
1 +   if(info->st_info_list != NULL){
2         clist_foreach(info->st_info_list, NULL);
3         clist_free(info->st_info_list);
4 +   }
5     free(info);
```

(c) CVE-2022-4121

Figure 7: If condition construction in typical NPD fixes.

Thus, to select the checked variable, we trace the CFG of the current function and select the variable at the null position since it is the root cause of the NPD errors. To determine the exception value, we analyze the statements of the callee function at the null position and extract the return value when the callee function fails. The failure return value can be determined by analyzing the function specification and enumerating the existing return values of callee failure. For instance, if the specification defines a return value of 0 for success and 1 for failure, we can use 1 as the exception value. When the function specification is unavailable, we determine the exception value by analyzing the existing return patterns, e.g., null is an exception value if all error handling involves returning null; a negative integer can be an exception value when each failure state corresponds to a negative return value.

To construct the if condition, there are two scenarios for the relationship between the checked variable and the exception value. The most typical case is to check if the variable is equal to the exception value when the checked variable receives the failure return value of the callee function at the null position. However, if statements between the null position and the error position have a data flow dependency with the checked variable (usually a free operation), a non-null check should be conducted in the if condition, and the following code executes when the non-null check holds.

4.3.2 Local Resource Retrogression

If the buggy function terminates due to an NPD error, the local resources, e.g., allocated memory and occupied locks, should be retrogressed to the initial states before continuing the execution. Neglecting to perform these retrogressive operations could lead to unreleased memory in the process or

```
1       rcu_read_lock();
2       slave = rcu_dereference(bond->curr_active_slave);
3 +     if(!slave){
4 +         rcu_read_unlock();
5 +         return -ENODEV;
6 +     }
7       xs->xso.real_dev = slave->dev;
```

(a) lock releasing in CVE-2022-0286

```
1       not_checked = kmalloc(sizeof(*not_checked) * 2);
2       checked = kmalloc(sizeof(*checked) * 2);
3 +     if(!not_checked || !checked){
4 +         kfree(not_checked);
5 +         kfree(checked);
6 +         return;
7 +     }
8       checked->data[] = …
9       not_checked->data[] = …
```

(b) memory deallocation in CVE-2022-3104

Figure 8: Local resource retrogression in typical NPD fixes.

invisible locks to other processes.

Figure 8 illustrates two typical cases where retrogression is needed before the function termination. In Figure 8(a), the lock `rcu_read_lock` is invoked before the variable `slave` receives the return value of function `rcu_dereference`. If the function fails, `slave` obtains the exception value and the program terminates. However, before the program termination, the occupied lock should be released; otherwise, other processes may not request the lock successfully. In Figure 8(b), variables `not_checked` and `checked` request a piece of memory space using `malloc`; however, if any of the allocations fails, we need to free both memory spaces using `kfree` and exit the current function. Note that `kfree` does nothing when freeing an empty heap; thus, detecting a failed allocation and freeing both memory spaces will not introduce an error.

To retrogress the local resources, we refine the process into two phases. First, it is straightforward to free allocated memory when an NPD error occurs. However, there are diverse pairs of allocation and free functions, e.g., `malloc`/`free`, `kmalloc`/`kfree`, and `g_malloc0`/`g_free`, while developers can customize their own allocation/free functions. Thus, we traverse all the CVE records [30] until April 21, 2023, including 200,759 items, and the official documents of the Linux kernel and the GNU C Library (glibc), and retrieve the keyword pairs with "alloc" and "free". After manual verification, we generate a memory allocation/free dictionary. When analyzing an NPD program, we search the keywords "alloc" and "free" before the error position of the buggy function. We compare keywords with the generated allocation/free dictionary to determine if further free operations are needed. For the user-defined allocation/free functions, the function names usually contain the listed keywords; thus, we analyze the one-hop call chain from the user-defined allocation/free functions to the functions in the allocation/free dictionary to determine the free operations.

Second, to release the occupied locks, the key challenge is to find the lock/unlock function pairs. Intuitively, we search the keywords with "lock" and "unlock" in the kernel functions to generate a dictionary. Also, we analyze the CVE records [30] to extract the lock/unlock pair information, such as write_lock/write_unlock, write_lock_bh/write_unlock_bh, and rcu_read_lock/rcu_read_unlock. During NPD program analysis, we retrieve the key information "lock" and "unlock" before the error position of the buggy function, and compare them with the extracted lock/unlock pairs to determine the further unlock operations.

In addition, CONCH handles other functions beyond standard C functions by analyzing callee functions along each execution path. When a callee function belongs to the preceding parts of paired functions, CONCH seeks the subsequent part using name similarity and passed arguments.

4.3.3 Return Statement Construction

The repair process involves constructing a return statement, which heavily relies on both the function's return type and the existing return statements within the current function. For a `void`-type function, only a `return` statement is required without any return value. However, for a `bool`-type function, returning `true` or `false` represents a totally different result. In addition, the function can return a null, a predefined macro, or a negative integer. In some cases, the function executes the `continue` statement to enter the next iteration or the `break` statement to terminate the loop. The selection of return values depends on the context statements and an incorrect return value may lead to the chaotic logic of the program.

Figure 9 shows four different return types for handling NPD cases. In Figure 9(a), `input_node` receives the return value of `GetNode` and then the value of `input_node` is passed for further operation. Thus, a null-pointer checking is required, i.e., if `input_node` is null, the function returns to its caller function. Since the return type of current function is `bool` and similar return patterns for error handling (e.g., Line 2) can be referred to, the return value for this null pointer handling should be `false`. In Figure 9(b), function `kcalloc` returns null when fails, so the variable `imx_keep_uart_clocks` requires a null-pointer checking. Due to the `void` function type, the `return` statement is added without any value. In Figure 9(c), if `cl` is a null pointer, the function returns a predefined macro `FAIL`, according to the adjacency error handling code (i.e., Line 2 and 7). In Figure 9(d), if `val` is null, the `strcmp` operation cannot be completed. Because the operation is in a loop and a single iteration failure does not mean the whole loop failure, a `continue` statement is required when `val` is null.

Therefore, to construct the return value, we first analyze the return type of the current function. If the return type is `void`, a `return` statement is required as the initial attempt. However, for the example in Figure 3, a simple `return` is not correct in context and we talk about this scenario in § 4.4. Second, we transfer the existing error-handling statements to the added if check in the same function. For example, in Figure 9, three

```
1       if(IstensorIdControlling(tensor_id))
2           return false;
3       input_node = graph.GetNode(tensor_id.node());
4 +     if(input_node == nullptr)
5 +         return false;
6       return IsSwitch(*input_node);
```

(a) CVE-2022-23589

```
1       if(imx_keep_uart_clocks){
2           imx_uart_clocks = kcalloc(clk_count, … );
3 +         if(!imx_uart_clocks)
4 +             return;
5           if(!of_stdout)
6               return;
7       }
```

(b) CVE-2022-3114

```
1       if(rettv->vval.v_object == NULL)
2           return FAIL;
3       cl = rettv->vvval.v_object->obj->class;
4 +     if(cl == NULL)
5 +         return FAIL;
6       if(get_func_argument(…) == FAIL)
7           return FAIL;
```

(c) CVE-2023-1355

```
1       while(scanindent(s)){
2           var = scanname(s);
3 +         if(!val)
4 +             continue;
5           if(strcmp(var, "command") == 0)
6       }
```

(d) CVE-2021-30219

Figure 9: Four typical return types in the NPD fixes.

cases are learned from the existing fixing patterns. Third, we can learn the error-handling statements from the caller function, since the caller function may have the corresponding statements to handle different return values of the callee function. For example, in Figure 3, the function `caller_caller` handles a negative integer when the `caller` fails, so the patch of `caller` needs to return a negative integer for failure. In some cases, we only know the scope of the return value, such as an integer or a negative integer, not the exact value; however, we can match them with the official error macros [5] to obtain the exact return value, such as `ENOMEM` for *Out of Memory* and `EINVAL` for *Invalid Argument*. After constructing if condition, retrogressing local resources, and constructing return statements, an initial patch is generated, guaranteeing its correctness in the current function.

4.4 Interprocedural State Propagation

After obtaining the initial patch, the NPD error is handled in the current function; however, the fix cannot guarantee its correctness in the whole call chain. Thus, we introduce the interprocedural state propagation to obtain the final patch by resetting the global variables and function arguments and handling the NPD errors in the entire call chain.

4.4.1 Global Variable and Function Argument Resetting

To reset the global variables and function arguments, we can update the initial patch as illustrated in Algorithm 1. First,

Algorithm 1 Global Variable and Argument Resetting

Input: initial patch generated by intraprocedural analysis.
Output: updated patch with variable resetting.

1: **procedure** VARIABLE_RESETTING(DFG)
2: $\quad globals[\,] \leftarrow null$
3: $\quad arguments[\,] \leftarrow null$
4: $\quad$**while** $stmt \in (function_entry, error_position)$ **do**
5: $\quad\quad$**if** $var \notin local_vars$ **then**
6: $\quad\quad\quad$**if** $var \in func_arguments$ **then**
7: $\quad\quad\quad\quad arguments.append() \leftarrow var$
8: $\quad\quad\quad$**if** $var \in global_vars$ **then**
9: $\quad\quad\quad\quad globals.append() \leftarrow var$
10: $\quad$**while** $funcs \in (buggy, err_handled)$ **do**
11: $\quad\quad$**if** $var \in (globals[\,] \,||\, arguments[\,])$ **then**
12: $\quad\quad\quad$**if** $resetting \in other\ error\ stmt$ **then**
13: $\quad\quad\quad\quad var \leftarrow resetting$
14: $\quad\quad\quad$**else**
15: $\quad\quad\quad\quad var \leftarrow data\ flow\ analysis$
16: $\quad$**return** updated patch with resetting inserted

we need to identify the global variables and function arguments. In a buggy function, our analysis scope is from the function entry to the last error position, since the subsequent variables will not affect the error handling. We do not even consider local variables in this scope because our purpose is to reset the global variables and function arguments that affect the function (Line 5). If the variables are passed from the argument list, they are appended to the array *arguments* (Line 7). However, identifying the global variables is complex since their definitions are out of analysis scope; hence, we extend the search space to the global statements prior to the buggy function. Once the global variable definitions are out of the current file, we traverse the included header files. After identifying the global variables, we append them to the array *globals* (Line 9). The identification process is shown in Algorithm 1 of lines 2 to 9.

Then, we obtain the expected values of global variables and function arguments. Our overall strategy is to infer the values from either the existing error-handling code in the buggy function or the data flow in the caller function. In the first case, our observation is that if a global variable or a function argument needs to be reset in a code snippet, it may be reset in other statements of the current function (Line 12). For example, Figure 2 shows the function argument `*r` requires a reset operation while we can obtain its value from call chain analysis in the context. The value can also be obtained from the current function, since the `func` returns 0 to `*r` at Line 9 if the operation in the function is discarded. If no reset statement can be referred to in the current function, we gather information from its caller function via data flow analysis. The second case usually occurs when requiring more return values; however, only one value is directly returned while others are passed via global variables or function arguments.

Hence, we analyze the data flow of the target variable in the call chain from the buggy function to the caller of the function that returns the target variable in its return statement. By inferring the failure state in the caller, the target variable value can be determined (Line 15). The resetting process is shown in Algorithm 1 from Line 10 to Line 15.

4.4.2 Call Chain Assessment

In this step, we further update the patch to generate the final patch. Our goal is to keep the generated patches correct in the whole call chain; in other words, an NPD error should be addressed over all the caller functions. We first analyze the return statement in the updated patch to obtain the return type, e.g., `void`, `bool`, and `int`. If the return type is `void`, the current function does not return any value to its caller function, thus the caller function will not be notified when the current function fails. Hence, we need to check if the caller function can execute without interruption when the buggy function triggers the NPD return because an incorrect return value may propagate errors to the caller functions silently. If the caller function is also buggy, we recursively analyze the predecessor caller function of the buggy function until reaching the function in which there is a statement to handle the error (we call it the *error-handled function*). Then, for all functions between the buggy function to the error-handled function, we update them by adding error-handling statements according to the semantics in the error-handled function. Finally, we update the return type of the buggy function from `void` to `int` and return a value to its caller. Several real-world vulnerabilities, e.g., CVE-2022-3112 [26], follow this fixing pattern. If the return type is not `void`, it already returns specific exception values to its caller. Here, we can check if the caller handles the return value correctly. If not, we update the caller function according to the semantics in the updated patch. The process to generate the final patch is shown in Algorithm 2. After patching into the buggy program, we can obtain a correct program where the NPD errors are handled in the entire call chain.

Algorithm 2 NPD Fixing Assessment in Whole Call Chain

Input: updated patch with variable resetting.
Output: final patch with error handled in the call chain.

1: **procedure** ENTIRE_CHAIN_ASSESSING(CFG)
2:     *returns* ← *ReturnStmtInPatch*()
3:     **if** *returns* ∈ *void type* **then**
4:         **if** *caller has normal return value* **then**
5:             **while** $funcs \in (buggy, error_handled)$ **do**
6:                 *UpdateFunc(funcs)*
7:                 *returns* ← *ReturnUpdate()*
8:     **else**
9:         **if** *caller handles incorrectly* **then**
10:             *UpdateFunc(caller)*
11:     **return** final patch

5 Implementation

We implement CONCH with about 1,000 lines of Python code, which includes null and error position identifications, NPD context graph construction, fixing position selection, and patch generation.

Null and Error Positions Identification. We build our system on top of Facebook infer (FBinfer) [7]. Given an NPD program, FBinfer takes the source code as input and outputs the null and error positions and their corresponding paths according to the separation logic and its reference rule in Equation 1.

NPD Context Graph Construction. After obtaining the null and error positions and their related paths, we locate the callee functions at the null positions and analyze their implementation. Since we focus on the exception return values of the callee functions to construct the if condition, we only preserve the error-handling statements and their return statements of the callee functions. Then, we traverse the files to obtain the caller functions that call the buggy function and keep the calling statements along with the calling conditions and their return values. Finally, we generate the NPD context graph by connecting the CFG of the buggy function (from the function entry to the last error position) with the pruned CFG of the caller and callee functions.

Fixing Position Selection. We obtain a subgraph from the NPD context graph by only retaining the slice from the first null position to the last error position, along with their corresponding paths. In the subgraph, we mark each null position and each error position. The fixing positions are selected by calculating the distribution of the null and error positions according to the predefined patterns in § 4.2.

Patch Generation. In the intraprocedural state retrogression, the checked variable is obtained from the null position and the checked value comes from the exception value of the callee function. Then the relation between the checked variable and the checked value is determined by the data dependency of the subsequent code. For the local resource retrogression, we create a function pair dictionary by retrieving the keywords in the kernel and generic libraries. Then we match the functions before the error position to determine the release operations. CONCH extends its analysis beyond just standard C functions by examining the functions called within each execution path. If it detects that a called function is the first in a pair of related functions, CONCH identifies the matching second function based on the similarity in their names and the arguments passed to them. The return statement construction relies on the function type and existing error-handling statements in the current function. When conducting interprocedural state propagation, we rely on data flow analysis on the NPD context graph to reset the global variables and arguments and assess the initial patch in the entire call chain.

6 Evaluation

Our objective is to show that CONCH can significantly outperform the SOTA approaches (e.g., VFix) for repairing NPD errors in terms of contextual correctness. By comparing CONCH with SOTA approaches in fixing NPD vulnerabilities reported in CVE, we find that CONCH can correctly repair 85% vulnerabilities while the SOTA approaches can fix up to 58.75%, outperforming accuracy of 26.25 percentage points. In addition, we also evaluate the effectiveness of CONCH using the dataset of 18 NPD errors in `Defects4j`. CONCH can successfully fix 16 NPD errors by generating patches that are as correct as the developers' manual fixes, while the SOTA solutions can only generate up to 12 correct patches.

6.1 Experimental Setup

Datasets. To evaluate the accuracy of our method, we use two real-world datasets. The first dataset comes from real-world vulnerabilities, i.e., CVE records. We search in MITRE with the keywords "Null Pointer Dereference" and get 2,188 records by April 2023. Among them, 177 records have the commit information. We exclude the misclassified records (i.e., not NPD errors), and race conditions are also beyond our consideration. We obtain 81 CVE records and most of them are from the Linux kernel. Regarding patches submitted by developers, 80 vulnerabilities can be fixed by adding if conditions, and only one vulnerability (CVE-2023-1095 [31]) is fixed by initializing the pointer. In this paper, we focus on the case of adding the if check, i.e., 80 CVE records. The second dataset consists of 18 NPD errors, coming from a well-known benchmark, `Defects4j`, with 7 in `Chart`, 6 in `Closure`, 2 in `Lang`, 2 in `Math`, and 1 in `time`.

Other Program Repair Methods. VFix [48] represents the SOTA approach to addressing NPD issues with the best patch accuracy. One of its distinguishing features is its utilization of value-flow information within programs, enabling precise localization of suspicious statements and effectively reducing the search space for potential patches. By narrowing down the patch space, VFix significantly enhances the efficiency of the generate-and-validate process while increasing the likelihood of discovering the correct patches. In comparative evaluations focused on NPDs, VFix has demonstrated superior performance to other SOTA Automatic Program Repair (APR) techniques such as GenProg [15], ACS [47], CapGen [46], SimFix [12], and NPEfix [6]. In this paper, besides the VFix, we also select one NPD-related method (i.e., NFEfix) and one general-purpose APR solution (i.e., SimFix) as the baselines for measuring the accuracy of NPD fixing, since both of them represent the best patch generation accuracy evaluated in VFix.

System Runtime. Our experiments are conducted on a machine with an Intel Core i7 1.8GHz CPU and 16GB memory, running Ubuntu 22.04 with FBinfer 1.1.0.

6.2 Performance on CVE Dataset

There are 80 records in the CVE dataset. CONCH can generate 68 correct patches and 12 incorrect patches, with an accuracy rate of 85%. Among 68 correct patches, 36 fixings are consistent with the developers' patches, and 32 patches have semantic equivalence to the developers' patches. We also apply SOTA approaches to generate NPD fixes for the first dataset; however, they only achieve an accuracy rate of up to 58.75%. Thus CONCH outperforms SOTA approaches with an accuracy of 26.25%.

Same Fixes as Developers' Patches. Among 36 patches that are exactly the same as the developers' fixes, there are 23 null checkings, 12 not-null checkings, and 1 other value checking. For return statements, 11 fixings do not require return statements, 5 fixes return without value, 4 patches return bool values, 3 patches use `break` statement to exit the loop, and the other 13 patches return predefined values or goto labeled code segment. Additionally, there are 3 patches requiring extra operations to free allocated memory (CVE-2022-3104), release the occupied lock (CVE-2022-41858), and reset the function argument (CVE-2022-2153) before return. We list such cases in § A.

Semantic-Equivalent Fixes. The 32 patches that have semantic equivalence to the developers' fixes can be divided into three categories. First, the constructed fix identifies the attribute of the return value (e.g., positive/negative, zero/nonzero) but does not have the exact value compared to the developers' fixes. There are 15 patches for such cases. For instance, when fixing CVE-2022-47021 [29], a negative return value of function `op_get_data` represents the failure state, and the developer uses the macro `OP_EFAULT` (-129); however, we cannot obtain this macro in context and hence return another macro `OP_EINVAL` (-131).

```
1  commit 1540d334a04d874c2aa9d26b82dbbcd4bc5a78de
2  diff --git a/src/testing.c b/src/testing.c
3  @@ -616,6 +616,11 @@ f_assert_fails(typval_T *argvars,
       typval_T *rettv)
4       // no allocated memory, no lock, no global variable
5       expected = tv_get_string_buf_chk(tv, buf);
6  +    if (expected == NULL)
7  +        goto theend;
8       ... // one hundred lines of other codes
9  theend:
10     trylevel = save_trylevel;
11     suppress_errthrow = FALSE;
12     // many unrelated codes
13     ...
```

Listing 1: Fix of CVE-2022-3153 generated by developer.

```
1  diff --git a/src/testing.c b/src/testing.c
2  @@ -616,6 +616,11 @@ f_assert_fails(typval_T *argvars,
       typval_T *rettv)
3       // no allocated memory, no lock, no global variable
4       expected = tv_get_string_buf_chk(tv, buf);
5  +    if (expected == NULL)
6  +        return;
7       ...
```

Listing 2: Fix of CVE-2022-3153 generated by CONCH.

Second, seven patches provided by the developers jump to

existing error-handling statements using the `goto` statement. However, there could be hundreds of lines of code between the fixing position and the label position, which makes further analysis complex and confusing. Therefore, we try to avoid the `goto` statement and then simplify the generated patches. For instance, List 1 shows the patch for CVE-2022-3153 generated by the developer. The fix reuses the existing error-handling statements. When the pointer `expected` is null, the program will jump into the label to conduct very complex processing. Since there are one hundred lines of code between the if check position and the label, the difficulty of further analysis increases. As a comparison, when we generate the patch, we first verify that there is no allocated memory, no lock, and no modified global variable, before the fixing position. According to the `void` return type, we return directly when determining that `expected` is null, as shown in List 2.

Third, for the other 10 patches, the developers' patches and the patches generated by CONCH have the same results but are represented in different ways, which include two aspects. The first one is that the developers may use a range to indicate a specific value, but we assess that value directly. For example, when dealing with non-negative integers, a developer may use a check for a value less than 1 to signify an empty state, shown in List 3; however, we simply check whether the value is equal to 0. The second one is that the developer may check the member of a pointer for its empty state, while we cannot obtain the detailed member information and hence check the null pointer directly.

```
1 commit 4ec0ef3a82125efc36173062a50624550a900ae0
2 diff --git a/drivers/usb/misc/iowarrior.c b/drivers/usb/misc/
      iowarrior.c
3 @@ -787,6 +787,12 @@ static int iowarrior_probe(struct
      usb_interface *interface, const struct usb_device_id *id
      )
4 +     if (iface_desc->desc.bNumEndpoints < 1) {
5 +         dev_err(&interface->dev, "Invalid number\n");
6 +         retval = -EINVAL;
7 +         goto error;
8 +     }
```

Listing 3: Fix of CVE-2016-2188 generated by developer.

Incorrect Fixes. CONCH cannot generate correct patches for 12 vulnerabilities, as listed in Table 1. The key reason is that we cannot obtain the checked variables or functions in context when constructing the if conditions. First, some class/struct object members cannot be obtained in context. For example, we can get the definition and usage of an object pointer in a function and add null pointer checks; however, the member of this pointer may require further checks. Due to no specific information to determine which member to check, we can only generate a partially correct patch to check the object pointer. Second, the relation between the variables and the specific values may not be inferred in context. Additionally, some complex execution logic can only be obtained from experts' in-depth analysis, such as $uaddr == uaddr2$ in CVE-2021-6647, which breaks the rule of different addresses [24]. In such cases, CONCH can generate patches; however, the patches cannot fix the vulnerabilities. Third, high-customized functions are required in some special scenarios, such as the sanity check functions, validation check functions, and vCPU initializing functions. Even if we can locate the error positions; however, we cannot find the appropriate functions to fix the vulnerabilities, thus CONCH cannot generate patches in this case.

Comparison with SOTA Approaches. For those 80 NPD errors, we also apply SOTA approaches to generate patches. The comparison results between CONCH and SOTA approaches for repairing real-world vulnerabilities are shown in Table 2. We can generate 36 patches that are the same as the developers' patches. Specifically, three patches implement the `break` statement to leave a loop, one uses `spin_unlock` to relinquish a lock, another employs `kfree` to release memory, and another resets a function argument. Additionally, one patch checks a unique function available in the context. These listed cases are beyond the scope of SOTA approaches: VFix has the capacity to generate 29 patches in total, NPEfix generates 15 correct patches, and SimFix has 18 correct patches. We can also generate 32 semantic equivalence patches, among which 11 need contextual information to frame if statements, two are for unlocking designated locks, and one addresses memory release. Based on these results, VFix can formulate 18 patches, while NPEfix and SimFix can muster 4 and 8 patches respectively. Concerning the 12 vulnerabilities for which we couldn't create precise patches, four of them have no patches proposed, three of them cannot catch the error, and five of them generate patches but does not handle resource properly. All of the SOTA approaches cannot yield any results for these 12 vulnerabilities, with an accuracy of less than 58.75%.

The Necessity of Contextual Checks. To demonstrate the necessity of introducing intraprocedural state retrogression and interprocedural state propagation, we evaluate the impact of the local resource retrogression, global variable and function argument resetting, and entire call chain assessment. The results show that six patches require the unlocking or memory-freeing operation, one fixing needs the function argument resetting, and three patches require multiple layers of caller functions to handle the NPD error. These cases occupy a proportion of 14.7% in the correctly generated patches.

We also analyze the number of layers of the call relation from the NPD context graph between the first caller function and the error position. We focus on the programs that have been correctly handled by CONCH. The result is shown in Figure 10. 55 programs have only one caller function, while eight vulnerable programs have two caller functions. There is one buggy program with four caller functions, one with five caller functions, two buggy programs with eight caller functions, and one buggy program with 15 caller functions. Among them, 65 programs can be handled in one layer, while the other three buggy programs require two layers (i.e., the caller of the caller function handles the error case).

Performance Overhead. The performance overhead of

Table 1: The reasons that CONCH cannot generate correct patches and we provide some partially correct patches.

Category	CVE ID	If Condition	Generated Patches	Why CONCH Cannot Generate Correct Patches
Unobtainable Member	CVE-2022-1674	rmp->regprog != NULL	rmp != NULL	member *regprog* cannot be obtained in context
	CVE-2022-1620	rmp->regprog != NULL	rmp != NULL	member *regprog* cannot be obtained in context
	CVE-2016-2782	serial->num_bulk_in < 2	serial != NULL	member *num_bulk_in* cannot be obtained in context
	CVE-2014-0101	!net->sctp.auth_enable	net == NULL	member *sctp.auth_enable* cannot be obtained in context
	CVE-2013-0313	inode->i_op->removexattr != NULL	inode->i_op != NULL	`removexattr` is not a function in context
Unobtainable Relation	CVE-2022-2874	cctx->ctx_skip != SKIP_YES	cctx != NULL	relation with SKIP_YES cannot be obtained in context
	CVE-2018-1092	ino == EXT4_ROOT_INO	ino == 0	relation with EXT4_ROOT_INO cannot be obtained in context
	CVE-2012-6647	uaddr == uaddr2	uaddr && uaddr2	relation that *uaddr* is equal to *uaddr2* cannot be obtained in context
Special Function	CVE-2022-3621	nilfs_is_metadata_file_inode(inode)	-	special function for sanity check
	CVE-2022-2302	JFS_IP(ipimap)->i_imap	-	special function for validation check
	CVE-2013-5634	!kvm_vcpu_initialized(vcpu)	-	special function for initializing vCPU
	CVE-2013-4119	!SecIsValidHandle(handle)	-	special function for validation check

Table 2: Comparison between CONCH and SOTA approaches for repairing real-world vulnerabilities.

	Same Fixing	Semantic Equivalence	Incorrect Patches	Proportion
VFix	29	18	33	58.75%
NPEfix	15	4	61	23.75%
SimFix	18	8	54	32.5%
CONCH	**36**	**32**	**12**	**85%**

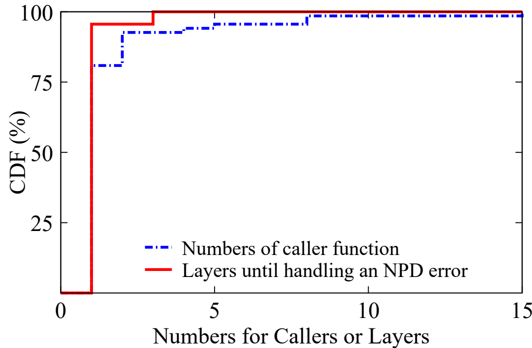


Figure 10: Numbers of callers and layers for CVE Dataset.

CONCH is mainly from locating the null pointer and the error positions using FBinfer [7], constructing the contextual graph for each vulnerable function, obtaining the required information, and generating patches. Specifically, for each NPD error, identifying vulnerable positions and outputting the error information cost 16.17 seconds on average. Constructing the contextual graph by combining the intraprocedural and interprocedural graphs takes about 10 seconds, in which the cost of constructing an intraprocedural graph is negligible (about 0.006 seconds), and querying and connecting each caller or callee function takes about six seconds. Finally, the cost of traversing the paths and obtaining the function pair according to the predefined patterns can be negligible, about 0.006 seconds.

6.3 Performance on Benchmark Dataset

The second dataset contains 18 NPD errors collected from `Defects4j`, which is a well-known benchmark suite used for evaluating APR tools. To be consistent, we select the same version 1.0.1 as tested by VFix, and the detailed information about the collected dataset is shown in Table 4 in § B. Since the accuracy of patch generation for NPEfix and SimFix in fixing this dataset has been tested in VFix and VFix outperforms both of them, we only compare CONCH with VFix. The comparison result is shown in Table 3. We can generate 16 correct patches and 2 incorrect patches, resulting in an accuracy of 88.89%. As a comparison, VFix can generate 12 correct patches and 6 incorrect patches, resulting in an accuracy of 66.67%. Our result outperforms VFix with an accuracy of 22.22%.

Same Fixes as Developers' Patches. Among these correct patches, VFix has the ability to generate 10 patches that are identical to the fixes made by developers. These 10 patches contain 6 null checks and 4 non-null checks. In the case of null checks, the patches simply return an existing variable, a bool value, or null without processing any intermediate states. The non-null checks execute the existing code. CONCH can address all of the vulnerabilities that VFix can fix, as well as an additional vulnerability that requires the use of a `continue` statement to skip a single iteration. Such a vulnerability is beyond the scope of VFix.

Semantic-Equivalent Fixes. Besides generating the 10 correct patches, VFix can also generate two semantic-equivalent patches. For the first patch, the developer checks the existence of the next node by calling `hasNext`, while VFix directly performs a null check for the next node. The second case pertains to the developer checking the invalid value (`NaN`), whereas VFix can produce a similar outcome. In contrast, CONCH can handle these two scenarios as well as three other cases that involve error message logging.

Incorrect Fixes. CONCH is capable of generating partially correct patches for two bugs. The first bug occurs when a variable is null in an if branch, and the developer replaces it

with a customized function. In such cases, we skip the branch, resulting in an inability to reproduce the same customized function. The second case involves resetting a special variable within a branch, which cannot be accomplished in the relevant context. As a consequence, CONCH is incapable of generating precise fixes for these two bugs. Apart from these cases, VFix is unable to address the remaining four bugs, which include throwing the error message and exiting the loop.

The Necessity of Contextual Checks. In the second dataset, 16 programs have only one caller function, while the remaining two have two caller functions. Furthermore, all of them only require a single layer of handling until the errors are resolved.

Performance Overhead. When reasoning about this benchmark, we extract the buggy codes and modify them to the format that FBinfer can compile without changing the previous logic. Then FBinfer does not require compiling the whole project, instead of running the functions in the tested dataset, with the performance overhead for creating the dictionaries and generating correct patches within 10 seconds.

Table 3: NPD errors in `Defects4j` and comparison between CONCH and VFix for repairing NPD errors in `Defects4j`.

Project	#NPD	Fixed by VFix			Fixed by CONCH		
		Same	Semantic	Incorrect	Same	Semantic	Incorrect
Chart	7	5	0	2	5	2	0
Closure	6	2	1	3	2	2	2
Lang	2	1	0	1	2	0	0
Math	2	1	1	0	1	1	0
Time	1	1	0	0	1	0	0
Total	18	10	2	6	11	5	2

7 Discussion

Scope and Limitations. Our tool is on top of FBinfer, which is capable of analyzing Java, C, and Objective-C. In this paper, we evaluated C for the first dataset and Java for the second dataset. We will test more different supported languages in further research. The limitations of CONCH are two-fold. First, CONCH may not precisely fix NPD bugs when indirect calls halt the connection of different CFGs. Though the SOTA approach MLTA [20] has been adopted to refine indirect-call targets, the unidentified indirect targets (due to the limitations of MLTA) may still affect accuracy. Second, CONCH may not generate correct patches when essential information is unavailable in context. Specifically, when inferring the relation between the variables and values in the if statement, we cannot get the correct results if the relation is unreachable in context. For example, in List 4, the relationship between the variable `ino` and macro `EXT4_ROOT_INO` cannot be obtained in context, leading to a failed patch. Also, when we infer the previous states for the global variables and return value, we rely on the information provided by the current function and the caller function. Therefore, failing to obtain this information affects the correct patch generation.

```
1  diff --git a/fs/ext4/inode.c b/fs/ext4/inode.c
2  @@ -4732,6 +4732,12 @@ struct inode *ext4_iget(struct
       super_block *sb, unsigned long ino)
3  +     if((ino == EXT4_ROOT_INO)&&(!raw_inode->i_links_count)){
4  +         EXT4_ERROR_INODE(inode, "root inode unallocated");
5  +         ret = -EFSCORRUPTED;
6  +         goto bad_inode;
7  +     }
```

Listing 4: Fixing for CVE-2018-1092 generated by developer.

Race conditions are beyond our scope. The reasons come at three folds. First, race conditions can be timing-dependent, which means that they may only occur under specific timing conditions, making it difficult to detect and reproduce the bug in a controlled environment. Second, race conditions are often non-deterministic, meaning that the outcome of a program can vary depending on the order in which threads execute. Third, fixing a race condition may introduce new bugs, especially when the fix involves changing the order of execution or adding synchronization.

Patch Quality. The quality of proposed patches relies on ISL accuracy, the coverage of fixing position selection, patch generation, and patch verification. As illustrated in ISL, there are no false positives in error detection if the code meets the proposed error patterns [37]. Also, the proposed four fixing position selection policies can cover all cases in our tested dataset. When generating the patches, the return value can be extracted from the function return type, existing error-handling statements in the current function, and error-handling statements in the caller function. We also conduct the intraprocedural and interprocedural analysis to evaluate it in the call chain to further verify the correctness of the obtained return value. After obtaining the generated patches, their quality can be guaranteed for the scenarios in which a simple pointer receives the return value of a failed function. However, for the struct pointer, the patches require further manual verification, since the relation with the member and special function of the struct pointer may not be obtained. In addition, after generating patches, we check the patch equivalence with the patches developed by the developers manually. We will automate the checking process in future work.

Scalability and Usability. In this paper, we generate patches for NPD errors with contextual checks, and it is promising to extend this method to a broader scenario where contextual checks are required to make the program consistent after adding if checks. Specifically, for all vulnerabilities that can be fixed by adding an if check, we can conduct path-sensitive fixing position selection, perform intraprocedural state retrogression to build if condition, retrogress the local resources including freeing the allocated memory and releasing the occupied lock, and construct return statements to obtain an initial patch. Finally, we can conduct the interprocedural state propagation to reset the global variable and function argument

and assess the initial patch in the call chain to generate the final patch.

We also input all the tested buggy programs into ChatGPT [35] to evaluate the performance on NPD fixing. Our test method is divided into two ways: only inputting the vulnerable code, and inputting the vulnerable code with labeled error position. Unfortunately, if we just input the vulnerable code, the result of fault localization in ChatGPT is much worse than FBinfer. If we input both the vulnerable code and the error position, the generated results fail to consider the contextual information, with a fixed format as "$if(p! = null)$ *return*;". The reasons that ChatGPT cannot generate correct patches come at two folds: i) when conducting intraprocedural analysis, there is no existing pattern to be learned; ii) for interprocedural analysis, the sequence-based method always ignores the call relation, failing to analyze the large code base projects.

8 Related Work

1) Automatic Program Repair. The search-based and constraint-based repairs are two popular automatic program repair techniques. Search-based repair techniques, such as GenProg [15], RSRepair [36], and HDRepair [14], utilize predefined templates and search algorithms to explore search spaces and generate potential bug-fixing patches. Another set of techniques, including AE [45], SPR [18], and Prophet [19], employ predefined transformation schemas and probabilistic models to identify and apply syntactic fragments, aiming to indirectly achieve the desired semantic effect for bug fixing based on a given test suite.

Constraint-based repair techniques, such as DirectFix [22], Angelix [23], and SemFix [34], utilize a set of constraints to guide the search for a solution and traditionally rely on test cases for patch validation, which can lead to overfitting problems [41]. SemGraft [21] references a correct program to generate input-output constraints for a semantic-equivalent buggy program, achieving higher-quality patches with partial correctness guarantees.

CONCH combines search-based and constraint-based repair techniques to conduct NPD error fixing. We first use separation logic to reason about the program errors and then limit the search space by utilizing the contextual information and with the assistance of caller functions.

2) Null Pointer Dereference Fixing. NPEfix [6] proposes runtime strategies, Sinha et al. [40] identify fault sources, and VFix [48] localizes statements using static and dynamic analysis. However, their approaches may produce incomplete or function-specific patches [17]. As a comparison, our approach contains contextual checking, guaranteeing one NPD error is handled successfully in all paths. Also, our approach takes the memory freeing and lock resetting operation into consideration, enhancing the usability of NPD fixing.

3) Separation Logic to Fix Memory-related Bugs. MemFix [16] simplifies the call graph and solves double-free and use-after-free issues using separation logic and a cover problem. SAVER [10] builds on MemFix by introducing an object flow graph to analyze memory-related object flow and generate patches within if branches. Compared to them, we rely on separation logic to generate patches for NPD issues. We first generate CFG and call graphs to locate the buggy positions and record the null position and error position of the buggy program. Then we select the fixing positions and construct repair statements combined with contextual checks, ensuring that NPD errors are handled in the entire call chain.

9 Conclusion

To address the challenges of generating accurate fixes for NPD errors, we propose CONCH to generate correct patches with contextual checks. We first generate an NPD context graph to preserve the semantics related to patch generation. Then we define four fixing position selection policies based on the distribution of the null and error positions, ensuring all errors can be resolved without introducing redundancy fixing. Next, we propose the intraprocedural state retrogression to generate an initial patch that includes building if condition, retrogressing local resources, and constructing return statements. Finally, we conduct interprocedural state propagation to reset some global variables and arguments and assess the initial patch in the whole call chain to obtain the final patch. We evaluate the effectiveness of CONCH with two real-world datasets. The experimental results show CONCH outperforms the SOTA approach over an accuracy of 22%.

Acknowledgments

We thank our Shepherd and the reviewers for their insightful feedback. This work was partially supported by ONR grant N00014-23-1-2122, NSF grant CNS-1822094, and NSFC under Grant 62132011.

References

[1] Anindya Banerjee and David A Naumann. Local reasoning for global invariants, part ii: Dynamic boundaries. *Journal of the ACM (JACM)*, 60(3):1–73, 2013.

[2] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*, pages 480–494. Springer, 2010.

[3] Cybersecurity Help. NULL pointer dereference in GnuTLS. https://www.cybersecurity-help.cz/vdb/SB2020090415, 2020.

[4] Cybersecurity Help. NULL pointer dereference in Vim. https://www.cybersecurity-help.cz/vdb/SB2023053071, 2023.

[5] Linux Kernel Developer. Error macros. https://elixir.bootlin.com/linux/latest/source/include/uapi/asm-generic/errno-base.h, 2023.

[6] Thomas Durieux, Benoit Cornu, Lionel Seinturier, and Martin Monperrus. Dynamic patch generation for null pointer exceptions using metaprogramming. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 349–358. IEEE, 2017.

[7] Facebook. Infer static analyzer. https://fbinfer.com, 2023.

[8] GeeksforGeeks. Security issues in C language. https://www.geeksforgeeks.org/security-issues-in-c-language, 2022.

[9] Harness Team. The Top 10 Exception Types in Production Java Applications – Based on 1B Events. https://www.harness.io/blog/10-exception-types-in-production-java-applications, 2020.

[10] Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. Saver: scalable, precise, and safe memory-error repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 271–283, 2020.

[11] ImmuniWeb. NULL Pointer Dereference. https://www.immuniweb.com/vulnerability/null-pointer-dereference.html, 2020.

[12] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pages 298–309, 2018.

[13] Ioannis T Kassios. The dynamic frames theory. *Formal Aspects of Computing*, 23:267–288, 2011.

[14] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 1, pages 213–224. IEEE, 2016.

[15] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011.

[16] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. Memfix: Static analysis-based repair of memory deallocation errors for c. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 95–106, 2018.

[17] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. NPEX: repairing Java null pointer exceptions without tests. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1532–1544, 2022.

[18] Fan Long and Martin Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering*, pages 702–713, 2016.

[19] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–312, 2016.

[20] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1867–1881, 2019.

[21] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. Semantic program repair using a reference implementation. In *Proceedings of the 40th International Conference on Software Engineering*, pages 129–139, 2018.

[22] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 448–458. IEEE, 2015.

[23] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*, pages 691–701, 2016.

[24] MITRE. CVE-2012-6647. https://github.com/torvalds/linux/commit/6f7b0a2a5c0fb03be7c25bd1745baa50582348ef, 2012.

[25] MITRE. CVE-2022-2153. https://github.com/torvalds/linux/commit/00b5f37189d24ac3ed46cb7f11742094778c46ce, 2022.

[26] MITRE. CVE-2022-3112. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-3112, 2022.

[27] MITRE. CVE-2022-3153. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-3153, 2022.

[28] MITRE. CVE-2022-41858. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-41858, 2022.

[29] MITRE. CVE-2022-47021. https://github.com/xiph/opusfile/commit/0a4cd796df5b030cb866f3f4a5e41a4b92caddf5, 2022.

[30] MITRE. All CVE Records. https://cve.mitre.org/cve/search_cve_list.html, 2023.

[31] MITRE. CVE-2023-1095. https://github.com/torvalds/linux/commit/580077855a40741cf511766129702d97ff02f4d9, 2023.

[32] MITRE. CVE-2023-1355. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-1355, 2023.

[33] MITRE. Null pointer dereference records. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=NULL+pointer+dereference, 2023.

[34] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE, 2013.

[35] OpenAI. ChatGPT. https://openai.com/blog/chatgpt, 2023.

[36] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265, 2014.

[37] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. Local reasoning about the presence of bugs: Incorrectness separation logic. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II 32*, pages 225–252. Springer, 2020.

[38] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[39] Seth Jenkins. Exploiting null-dereferences in the Linux kernel. https://googleprojectzero.blogspot.com/2023/01/exploiting-null-dereferences-in-linux.html, 2023.

[40] Saurabh Sinha, Hina Shah, Carsten Görg, Shujuan Jiang, Mijung Kim, and Mary Jean Harrold. Fault localization and repair for java runtime exceptions. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 153–164, 2009.

[41] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 532–543, 2015.

[42] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo A. Maia. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In *Proceedings of SANER*, 2018.

[43] WIKIPEDIA. Hoare logic. https://en.wikipedia.org/wiki/Hoare_logic, 2023.

[44] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. Patchdb: A large-scale security patch dataset. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 149–160. IEEE, 2021.

[45] Westley Weimer, Zachary P Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 356–366. IEEE, 2013.

[46] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th international conference on software engineering*, pages 1–11, 2018.

[47] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 416–426. IEEE, 2017.

[48] Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. Vfix: value-flow-guided precise program repair for null pointer dereferences. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 512–523. IEEE, 2019.

A Same Fixes As Developers' Patches

In Listing 5-7, we show the generated patches that are the same as the original developers' fixes.

```
1  commit 4a9800c81d2f34afb66b4b42e0330ae8298019a2
2  diff --git a/drivers/misc/lkdtm/bugs.c b/drivers/misc/lkdtm/
       bugs.c
3  @@ -327,6 +327,11 @@ void lkdtm_ARRAY_BOUNDS(void)
4       not_checked=kmalloc(sizeof(*not_checked)*2,GFP_KERNEL);
5       checked = kmalloc(sizeof(*checked) * 2, GFP_KERNEL);
6  +    if (!not_checked || !checked) {
7  +        kfree(not_checked);
8  +        kfree(checked);
9  +        return;
10 +    }
```

Listing 5: Freeing the allocated memory before returning (CVE-2022-3104).

```
1  commit ec4eb8a86ade4d22633e1da2a7d85a846b7d1798
2  diff --git a/drivers/net/slip/slip.c b/drivers/net/slip/slip.
       c
3  @@ -469,7 +469,7 @@ static void sl_tx_timeout(struct
       net_device *dev, unsigned int txqueue)
4       struct slip *sl = netdev_priv(dev);
5       spin_lock(&sl->lock);
6       if (netif_queue_stopped(dev)) {
7  -        if (!netif_running(dev))
8  +        if (!netif_running(dev) || !sl->tty) {
9  +            spin_unlock(&sl->lock);
10 +            return;
11 +        }
12          tty_chars_in_buffer(sl->tty);
13     }
```

Listing 6: Releasing the occupied lock before returning (CVE-2022-41858).

B Dataset Details

`Defects4j` is a well-known benchmark that collects multiple reproducible bugs, supporting infrastructure with the goal of advancing software engineering research. Here, we list the detailed information about the dataset, which contains the projects of `Chart`, `Closure`, `Lang`, `Math`, and `Time`.

```
1  commit 00b5f37189d24ac3ed46cb7f11742094778c46ce
2  diff --git a/arch/x86/kvm/lapic.c b/arch/x86/kvm/lapic.c
3  @@ -1024,6 +1024,10 @@ bool kvm_irq_delivery_to_apic_fast(
       struct kvm *kvm, struct kvm_lapic *src, struct
       kvm_lapic_irq *irq, int *r, struct dest_map *dest_map)
4       *r = -1;
5       if (irq->shorthand == APIC_DEST_SELF) {
6  +        if (!src) {
7  +            *r = 0;
8  +            return true;
9  +        }
10          *r = kvm_apic_set_irq(src->vcpu, irq, dest_map);
11          return true;
12     }
```

Listing 7: Resetting the function argument before returning (CVE-2022-2153).

Table 4: Detailed information about the collected dataset in `Defects4j`.

Project	Description	Bug ID	#NPD
Chart	Free chart library for Java platform	1, 4, 14, 18, 19, 25, 26	7
Closure	JavaScript checker and optimizer	2, 30, 40, 53, 96, 110	6
Lang	Helper utilities for java.lang API	33, 39	2
Math	Mathematics and statistics components	4, 53	2
Time	Java date and time library	3	1