Mitigating Security Risks in Linux with KLAUS
– A Method for Evaluating Patch Correctness –

Anonymous Author(s)

Abstract

The Linux kernel's growth introduces daily bugs that are often detected and eliminated using code analyzers. However, creating accurate Linux patches remains challenging and poses security risks. To address this, we manually analyzed 182 incorrectly developed Linux kernel patches and discovered that the inaccuracies usually result from changes to variable read and write operations by the patch. Based on this finding, we created KLAUS, a new method for evaluating patch quality.

KLAUS leverages abstract interpretation to extract modified read and write operations caused by the patch in the Linux kernel. It combines these alterations with branch-resolving mechanisms to guide a kernel fuzzer toward relevant code and contexts. Testing KLAUS on numerous real-world Linux kernel patches demonstrates its superior effectiveness and efficiency in detecting incorrectly developed patches. So far, KLAUS has identified and reported 30 incorrect patches to the Linux community, some of which could enable privilege escalation on Android and Ubuntu systems.

1 Introduction

The Linux kernel is complex and prone to bugs, as evidenced by the high number of bugs identified by various automated tools (e.g., [38, 47, 50–52, 58]). This highlights the need for continual bug fixing by the kernel community, especially for security-critical issues. As the Linux kernel forms the foundation for numerous computing platforms, fixing bugs is always a top priority.

To tackle the large volume of kernel bugs, the Linux community relies on crowd-sourcing to remediate them. Over the past ten years, thousands of bugs have been analyzed and fixed by numerous kernel researchers. Despite these efforts, many kernel patches prove to be challenging to develop correctly on the first try. From 2012 to 2022, over 3,500 kernel patches that aimed to address one issue actually introduced new bugs. Unfortunately, some of these new bugs were even more exploitable than the original bugs they were trying to fix, as demonstrated in Section 2.2 and 7.4.

Our manual analysis of hundreds of incorrect kernel patches reveals that the existence of these incorrect patches can be attributed to two main factors. First, the complexity of the Linux kernel makes it difficult for researchers and developers to ensure their patches do not disrupt the existing kernel logic. Second, there is a lack of effective mechanisms for assessing patch quality. Today, the only way to evaluate a patch is through regression testing, where the patch is applied to the kernel and its ability to eliminate the original bug's error behavior is tested by re-running the trigger input. However, as demonstrated in the following sections, the absence of an error behavior does not guarantee the patch's correctness.

One solution to address this pressing issue is to use widely adopted kernel fuzzing techniques such as Syzkaller [52], SyzVegas [53], StateFuzz [59], and HFL [42] to test the patched kernel. However, using these methods may result in inefficiencies as they often prioritize higher kernel code coverage, leading to a waste of time and resources on code that has no relation to the newly introduced bug.

In this work, we propose a new technical method to assess kernel patches, called KLAUS (standing for KerneL pAtch qUality aSsessor). Unlike traditional kernel fuzzing techniques, KLAUS first conducts a static analysis of the code before and after the patch is applied. It then uses abstract interpretation to track changes in read and write operations made by the patch. Our manual analysis of 182 incorrect Linux kernel patches, described in Section 3, shows that newly introduced bugs in incorrect patches primarily result from changes in read and write operations. Based on this observation, our static analysis informs a directed fuzzing mechanism that uses changes in read and write operations as indicators to guide the fuzzer toward code and context relevant to the patch.

While sharing the same goal of existing directed fuzzing schemes in the userspace (e.g., SemFuzz [57] and AFLGo [33]), our method – also recognized as a directed fuzzing mechanism – does not primarily focus on optimizing input to reach specific program sites. Going beyond reachability, it also takes into account the order and type of patch-altered read and write operations during kernel fuzzing. Additionally, our method utilizes branch-resolving mechanisms to further improve KLAUS's ability to detect incorrect patches. Our evaluation shows that KLAUS outperforms Syzkaller in detecting incorrect patches. With KLAUS, we discovered 30 incorrect patches from hundreds of Linux kernel patches. So far, 25 false patches have been confirmed and fixed. Out of all the bugs we discovered, we also found 3 that offered attackers higher exploitability than the original bugs. We successfully demonstrated privilege escalation against the latest Ubuntu and Android systems using these 3 vulnerabilities.

```
 1 // description
 2 nfc: fix refcount leak in llcp_sock_connect()
 3 Fixes: c7aa12252f51 ("NFC: Take a reference ...")
 4 // diff file
 5 diff --git a/net/nfc/llcp_sock.c b/net/nfc/
        llcp_sock.c
 6 --- a/net/nfc/llcp_sock.c
 7 +++ b/net/nfc/llcp_sock.c
 8 @@ -704,6 +704,7 @@ static int llcp_sock_connect()
 9 ...
10 + nfc_llcp_local_put(llcp_sock->local);
11 }
```

Listing 1: The snippet of commit 8a4cd82d62b5.

This work, to the best of our knowledge, is the first of its kind to explore the correctness of Linux kernel patches. It highlights a concerning reality that an improperly developed patch can result in a more severe security vulnerability. KLAUS, as a new testing tool, offers an effective and efficient method for uncovering bug patch incorrectness in the Linux kernel. Moreover, it empowers kernel developers to evaluate the quality of their patches, potentially reducing the risk of transforming a non-exploitable bug into a highly exploitable vulnerability.
In summary, this paper makes the following contributions.

- We have conducted a thorough examination of 182 incorrect Linux kernel patches and uncovered crucial insights about the root causes of incorrect patches. Our findings highlight the limitations of current methods for identifying incorrect patches and inspired the development of KLAUS.

- We propose a new method that leverages abstract interpretation to capture the change of the variables' read and write operations made by a kernel patch. Using the extracted read-and-write operation alteration, we also propose a new directed fuzzing mechanism for incorrect Linux kernel patch identification.

- We implemented KLAUS and thoroughly evaluated its performance by using hundreds of real-world Linux kernel patches. We have uncovered and reported 30 incorrectly developed and/or deployed kernel patches, resulting in many of these patches being immediately re-patched. Upon acceptance of this submission, we plan to make KLAUS available to the community by open-sourcing it.

2 Background & Working Example

This section delves into the background of Linux kernel patches. To clarify the issue of improperly created patches, we also provide a real-world example of how a non-critical kernel bug can turn into a vulnerable Linux kernel exploit.

```
 1 struct nfc_llcp_sock {
 2    struct nfc_llcp_local *local; // NFC device
 3 };
 4
 5 void *nfc_llcp_local_get(struct nfc_llcp_local *
        local) {
 6    kref_get(&local->ref); // local->ref++
 7 }
 8
 9 int nfc_llcp_local_put(struct nfc_llcp_local *
        local) {
10    if (local == NULL)
11       return 0;
12    // local->ref-- and free local via
13    // local_release() if local->ref == 0
14    return kref_put(&local->ref, local_release);
15 }
16
17 static int llcp_sock_bind(struct nfc_llcp_sock *
        llcp_sock) {
18    nfc_llcp_local_get(llcp_sock->local);
19    if (llcp_sock->ssap == LLCP_SAP_MAX) {
20       // 1st patch which is incorrect
21       nfc_llcp_local_put(llcp_sock->local);
22       llcp_sock->sock = NULL; // 2nd patch
23       goto put_dev;
24    }
25 put_dev:
26    return -EADDRINUSE;
27 }
28
29 void nfc_llcp_sock_free(struct nfc_llcp_sock *sock
        ) {
30    if (sock->local != NULL)
31       nfc_llcp_local_put(sock->local);
32 }
```

Listing 2: The code snippet of the Linux kernel patches. The 1st patch in commit 8a4cd82d62b5 fixes a memory leak bug but incorrectly introduces a double-free vulnerability. This new vulnerability resided in the kernel for almost two months until finally being fixed by the 2nd patch in commit c61760e6940d.

2.1 Kernel Commit and Patch

As shown in List 1, a kernel commit contains two major parts. The first part is a description starting with a title (line 2). If the commit is to patch a bug, the description will include a line with "Fixes:" as the prefix (line 3). The commit ID following this prefix is a previous commit that introduces the bug. The second part is a diff file which lists code modification to the kernel. It illustrates the location of code modification - in function llcp_sock_connect (line 8) of file net/nfc/llcp_sock.c (line 5-7) from kernel code line 704. The code modification is further specified in line 10 which starts with a "+" symbol and inserts a calling to kernel function nfc_llcp_local_put. Correspondingly, if a line will be removed, it starts with a "-" symbol in the diff file. By applying the code modification in the diff file, the vulnerable kernel is patched.

A patch typically goes through three phases before finally being merged into the upstream kernel. In the first phase, kernel developers craft a patch draft in the local workspace and post it to a corresponding mailing list for review and discussion. If other developers and maintainers reach an agreement, then this patch will be merged into the git tree of the kernel subsystem that contains the bug. Finally, when a merge window is open, all patches in the subsystem will be pushed to the upstream kernel. In the second and third phases, automatic testing will be performed to examine the patch quality. Common testing techniques include regression testing (*e.g.*, kselftest [8], 0-day kernel test [1], and KernelCI [7]) and Syztest [25]. They replay the input PoC program that triggers the bug against the patched kernel. If no kernel error is observed, they conclude the correctness of the patch. However, as we will show in the following session, these existing techniques are insufficient to detect mistakes in patches.

2.2 From Memory Leak to double-free

The patch shown in List 1 is to fix a memory leak bug. In List 2, we include other relevant kernel code illustrate the root cause. As shown in List 2, all `nfc_llcp_sock` sockets share one `NFC` device which is represented in the `local` field (line 2). To use the device, the kernel calls the function `nfc_llcp_local_get` (line 5) to increase its reference count (refcount for short) of `nfc_llcp_sock->local`. After finishing the usage, the kernel calls the function `nfc_llcp_local_put` (line 9) to decrease the refcount for balance. When the refcount is decreased to zero, it means the `NFC` device is no longer used. Then, the kernel will call function `local_release` to free `nfc_llcp_sock->local` (line 14).

The root cause of the memory leak is in function `llcp_sock_bind`. In line 18, the refcount of `llcp_sock->local` is increased to use `NFC` device. However, if `llcp_sock->ssap` is equal to `LLCP_SAP_MAX` (line 19), the kernel jumps to the label `put_dev` and returns the caller without decreasing the refcount, resulting in imbalance. As a consequence, the memory referred by `llcp_sock->local` is never freed and reclaimed, causing a memory leak. To rebalance refcount, the patch shown in List 1 inserts the call to `nfc_llcp_local_put` (line 21) in function `llcp_sock_bind`.

This patch is straightforward but incorrect because it mistakenly introduces a double-free vulnerability. To be specific, after the `NFC` device is probed, its refcount is initialized to 1. Then, the kernel creates two `nfc_llcp_sock` sockets. Binding the 1st socket to the device, the kernel executes lines 18-21, which increases the refcount to 2 (line 18) and decreases it to 1 (line 21). The refcount is well maintained so far. However, when the 1st socket is closed, the kernel will call function `nfc_llcp_sock_free` (line 29) to destroy it. Function `nfc_llcp_sock_free` further calls `nfc_llcp_local_put`, decreasing the refcount to 0 and freeing the memory referenced by `nfc_llcp_sock->local`. Recall that all `nfc_llcp_sock` sockets share one `NFC` device by having their `local` refer to the same `nfc_llcp_local`. After the `nfc_llcp_local` is freed, the `local` field of the 2nd socket becomes a dangling pointer. Then, binding the 2nd socket, the kernel decreases the refcount of `nfc_llcp_local->local` to 0 again in line 21, which frees the memory referenced by `nfc_llcp_local->local` for the second time.

Compared with memory leak, double-free is widely believed to be more exploitable [55]. This double-free was assigned with CVE-2021-23134 [5] and kernel developers crafted another patch with commit `c61760e6940d` [11] for remediation. This 2nd patch nullifies `nfc_llcp_sock->local` in line 22 so that `nfc_llcp_local_put` is not called in line 31 when the 1st socket is destroyed. As such, the refcount is not decreased to 0, avoiding mistaken free of the memory referenced by `nfc_llcp_local->local`.

In summary, the 1st patch in commit `8a4cd82d62b5` [18] tries to fix a memory leak but incorrectly introduces a new double-free vulnerability. This new vulnerability was not detected by any testing techniques and resided in the kernel for almost two months. To fix the new vulnerability, kernel developers had to revisit the same piece of code, diagnose the root cause again, and craft the 2nd patch in commit `c61760e6940d` [11] for remediation.

3 Empirical Analysis and Design Overview

We collected incorrect Linux kernel patches and analyzed them to understand why patches can mistakenly introduce new vulnerabilities and why existing testing techniques are unable to detect them.

3.1 Incorrect Patch Collection

We took the following steps to collect a set of incorrect patches for empirical study. First of all, we crawled the upstream Linux kernel reports [24] to get all commits with the “Fixes” tag from 2012 to 2022. Recall that, the “Fixes” tag in a commit indicates that the purpose of this commit is to patch a bug in the kernel. To filter out incorrect patches from these commits, we tried to construct fixing relationships. More specifically, we traversed from the newest commit c_0 in this set through a depth-first search. If the commit c_1 referred to by the “Fixes” tag in c_0 is also a patching commit, we built a fixing relationship between them $c_0 \rightarrow c_1$ and continue the traversing from c_1. We repeated this process until all commits are examined. Finally, we got 4,534 chains representing fixing relationships. There are 8 longest chains built in this approach, containing 6 commits across the span of 1,115 days at most. In other words, it took 1,115 days and 5 patches for kernel developers to fix the initial bug. All commits in a chain, other than the head and the tail, are fixing the previous commit and in the meanwhile fixed by the next commit. Therefore, these commits are incorrect patches we are searching for. In total, we found 3,706 incorrect patches.

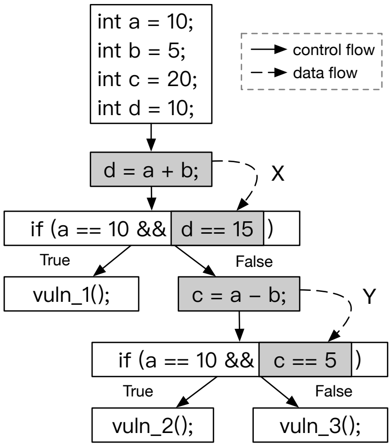

Figure 1: A synthetic program to explain how AWRPs introduced incorrect patches finally result in new vulnerability.

Considering the huge amount of incorrect patches, we sampled 182 (5%) from them for manual analysis. We only sampled the reports generated by Syzkaller so that we can estimate their security impacts and reproduce them using the essential information in Syzkaller's reports.

We contend that these 182 samples serve as representative examples for the following reasons. First, the sample size surpasses that of previous empirical studies focused on kernel patches (*e.g.*, [39, 41, 44–46, 56]), which typically featured between 10 to 90 test cases. In comparison, our dataset consists of a significantly larger sample size, providing a more comprehensive perspective.

Second, these 182 samples encompass a broad spectrum of vulnerability types. Among them, 53 cases involve use-after-free vulnerabilities, 11 pertain to out-of-bound issues, 38 relate to general protection faults, 20 involve concurrency problems, 16 concern memory leaks, and 9 involve uninitialized variables. Additionally, there are 35 cases classified under various other vulnerability types. This diverse range of vulnerabilities ensures that our dataset captures a wide array of kernel security issues.

To understand why these 182 patches are incorrect, we built a team of three security analysts. All team members have rich experience in kernel patching and vulnerability exploitation. Each incorrect patch was randomly assigned to two analysts, and conclusions were drawn only when their analysis results converged. It took the whole team about 1,080 man-hours to finish. On average, each commit requires nearly 6 hours to conclude.

3.2 Key Observations

Through the analysis, we gain two key observations which not only explain why existing techniques are incapable of detecting incorrect patches but also enlighten the design of `KLAUS`.

Observation 1: Old and new vulnerabilities share similar contexts. Intuitively, the incorrect patch fixes the old vulnerability and, at the same time, introduces a new vulnerability. Therefore, the kernel codes pertaining to the new vulnerability have a large overlap with those pertaining to the old vulnerability, which means the contexts are similar. In our working example, to trigger the memory leak, a `nfc_llcp_sock` socket must be created and bound to the `NFC` device. To misbalance the refcount, `llcp_sock->ssap` must be equal to `LLCP_SAP_MAX`. These conditions are also needed to trigger the double-free vulnerability introduced by the incorrect patch.

This observation hints that to construct a PoC program (i.e., an input) to trigger the new vulnerability, we can reuse the context provided by the PoC program for the old vulnerability to save time. From this perspective, it seems regression testing is an ideal solution for incorrect patch detection because it replays the PoC program and thus the same context. The same context can reproduce the old vulnerability but to trigger the new vulnerability, the context must be varied. In our working example, on the basis of one `nfc_llcp_sock` socket used to trigger memory leak, binding a second `nfc_llcp_sock` socket to the `NFC` device is critical to manifest the double free.

Therefore, an efficient strategy to explore the huge codebase of the kernel and trigger the new vulnerability is to stick the kernel execution to the old context and in the meanwhile vary it subtly. To this end, an instinctive solution is to perform direct fuzzing towards code changes introduced by the patch (*e.g.*, [54, 57]). Since the changes are close to the vulnerable code, fuzzing towards them is likely to preserve the context. By covering the changes introduced by the patch, the context is varied to accommodate to the new vulnerability. However, this intuitive design can hardly work in practice. Again looking at our working example, we can note that the patch does not modify `nfc_llcp_sock_free` function. A naive directed fuzzing will not set this function as the target and make attempts to reach it. As a consequence, no matter how many times the added `nfc_llcp_local_put` function is executed, the expected double free will never be triggered.

Designing a smarter directed fuzzing to trigger the new vulnerability resulting from the incorrect patch, we, therefore, need more in-depth guidance rather than merely based on code changes. A recent work PatchVerif [41] proposes a method for patch applied to robotic vehicles. It executes the code before and after modification and then monitors the variation of the vehicle's speed, location, and altitude etc. If the variation is significant, PatchVerif concludes that the corresponding code change might be faulty. While this method demonstrates effectiveness in patch assessment, it is designed specifically for robotic vehicles. As a result, it is not able to be applied to our problem.

Observation 2: New vulnerability results from Altered Write-Read Pairs (AWRP). The old vulnerability is fixed because the code changes introduced by the patch make modifications to the data flow and control flow of the kernel. Due to these changes, the triggering condition of the old vulnerability can no longer be satisfied. In our working example, the patch decreases the refcount by inserting `nfc_llcp_local_put` before returning from function `llcp_sock_bind`. This new write operation of the refcount rebalances the refcount and prevents memory leak. However, from a broader perspective, this new write operation in `nfc_llcp_local_put` is unintentionally associated with the read of the refcount in `nfc_llcp_sock_free` function, which causes an unexpected double free.

Through empirical study, we discovered that it is not an coincidence that the new vulnerability is caused by a write-read pair like the one in our working example. Further, the root cause of all incorrect patches can be attributed to the occurrence of Altered Write-Read Pairs regarding the same variable (AWRP for short). Generally, we consider a pair of write-read operations of the same variable is an AWRP if one of the following situations happens. ❶ A new write or read operation is added by the patch. This new operation can be matched with existing read or write operations to form a new pair. ❷ An existing write or read is removed by the patch. As a result, an existing pair is eliminated. ❸ The writing value or the reading behavior is changed. In other words, either the interesting variable is assigned with a different value or the interesting variable is used in a different way. This situation can be deemed as a combination of the first and second situations: removing an old write/read and adding a new write/read. ❹ The path condition to execute the write or the read operation in an existing pair is changed.

In Figure 1, we use a synthetic example to illustrate when a patch is applied, how AWRPs affect the data flow and the control flow, finally resulting in new vulnerabilities in the kernel. There are two distinct write-read pairs in Figure 1: `d = a + b` matches with `d == 15` with regard to variable `d` in pair X and `c = a - b` matches with `c == 5` with regard to variable `c` in pair Y. Now we consider three possible patches that can be applied individually. In our first synthetic patch, we assume that `d = a + b` is newly added and all others exist in the original kernel. According to situation ❶ mentioned above, pair X is an AWRP because `d = a + b` is a new write operation of variable `d`. With this new write, the condition in the first `if` statement (*i.e.*, `if(a == 10 && d == 15)`) will be evaluated to `True`. Then the kernel will take the branch and triggers the vulnerability `vuln_1`. In our second synthetic patch, we assume that `d = a + b` is removed and all others remain unchanged in the kernel. According to situation ❷, pair X is an AWRP because `d = a + b` is a deleted write operation of the variable `d`. As a result, the code in the false branch of the first `if` statement is activated. It is no longer a piece of dead code. According to situation ❹, now pair Y is also an AWRP. After executing `c = a - b`, the kernel will evaluate the second `if` statement to be `True` and takes the branch to trigger the vulnerability `vuln_2`. In our third patch, `d = a + b` is removed and `c = a - b` is changed to `c = a + b`. According to ❸, pair Y is an AWRP. The condition in the second `if` statement will be evaluated to `False` which leads to the triggering of vulnerability `vuln_3`.

Algorithm 1: WorkList-Based Abstract Interpretation

```
Input: Function $F$ and its $CFG$ with $e$ as the entry node ;
Output: $S[]$;
Initialize: $waitList = \{e\}, S = \{e \to \top, \forall n \neq e \to \bot\}$ ;
while $\exists\, n \in waitList$ do
    $waitList = waitList - \{n\}$ ;
    $s' = Transfer(S[n], n)$;
    foreach $n' \in Successors(CFG, n)$ do
        if $s' \nsubseteq S[n']$ then
            $S[n'] = S[n'] \cup s'$;
            $waitList = waitList \cup \{n'\}$;
```

The AWRP methodology provides a way to analyze the impacts of patches on data and control flow. Our findings show that incorrect patching results from mistakenly altering write-read pairs. Additionally, we have observed that incorrect patches may trigger kernel errors when the AWRP-related code is executed.

3.3 Design of KLAUS

Based on the key observations discussed above, we present the design of KLAUS which aims to generate concrete input to execute kernel code related to patch and thus demonstrate kernel errors if the patch is incorrectly developed or deployed.

The key idea of KLAUS is to leverage AWRP as the indicator to efficiently search the huge codebase of the Linux kernel. Through executing the write operation first and then the read operation in an AWRP, KLAUS generates the desired input sequence. Following this idea, the first component of KLAUS utilizes static analysis techniques to identify AWRPs caused by the patch. The static analysis component starts by profiling the influence of AWRPs through abstract interpretation. By collecting instructions the abstract states of which are changed because of the patch, KLAUS constructs AWRPs by pairing write instructions and read instructions that are pertaining to the same variable.

With the set of AWRPs as the guidance, the second component of KLAUS performs directed fuzzing. As mentioned above, vulnerabilities related to incorrect patches can be triggered by executing AWRPs. Inspired by this, our directed fuzzing first makes attempts to reach the write operation and then switches to reach the read operation belonging to the same pair. As we will elaborate in Section 5, KLAUS uses both AWRP-related code and type coverage as the feedback for exploration. Considering that write and read operations can be dominated by branches that are hard to trigger because of complex conditions, KLAUS explores symbolic tracing and an instrumentation-based mechanism to bypass these branches in an efficient fashion. In Section 5.3, we will elaborate on more design details. It should be noted that considering the complexity of the Linux kernel and the limitation of constraint solvers, we do not use symbolic execution to explore the kernel and thus execute AWRPs. Instead, we leverage more efficient directed fuzzing solutions. Furthermore, it is important to emphasize that our system, KLAUS, is specifically designed to address memory errors within the Linux kernel. As such, it is not intended or optimized for handling logic errors that may arise in the kernel.

4 AWRP Identification

As is mentioned above, the first component of KLAUS is to identify AWRPs introduced by a patch. Here, we introduce how to achieve it by using abstract interpretation. In the following, we first introduce abstract interpretation at a high level. Then, we present our definitions of abstract state, transfer function, and joint operator with regard to AWRP. Finally, we discuss our algorithm to identify AWRPs on the basis of abstract interpretation analysis.

4.1 Abstract Interpretation

Abstract interpretation is a static analysis framework that considers all paths and inputs to obtain a sound over-approximation of the state at every program location [37, 49]. For the sake of efficiency, the state is abstract and often represented by a set of constraints in a certain abstract domain. For example, in an interval domain employed in a value-set analysis, each constraint could be in the form of $lb \leq x \leq ub$, where x is a variable and lb, ub are the lower and upper bounds. The joint of two states, $s_1 = lb_1 \leq x \leq ub_1$ and $s_2 = lb_2 \leq x \leq ub_2$ is notated as $s_1 \sqcup s_2 = min(lb_1, lb_2) \leq x \leq max(ub_1, ub_2)$. The $\sqcup$ denotes the join operator which returns an over-approximation of the set union. The goal of using an abstract domain to represent states is to reduce the computational overhead. Therefore, abstract interpretation is especially useful for the analysis of large-scale programs like OS kernel.

Computing on the control flow graph (CFG) of the program, we can reach a fixed-point of states. Without losing generality, in KLAUS, we assume the CFG has a unique entry code and a unique exit code. The nodes in CFG are associated with instructions (LLVM IR instructions in our implementation) and edges represent control flows. Algorithm 1 presents a generic worklist-based algorithm to compute the fixed-point. Every node n in the CFG has an abstract state $S[n]$ which is a sound over-approximation of all possible states at n. Initially, $S[n]$ is $\perp$ (empty) for all CFG nodes except the entry node which is $\top$ (tautology). The transfer function in computation takes as an instruction *inst* and its state s an input, and returns a new state s' which is the result of executing *inst* in state s. To ensure that analysis converges, when the program has a loop or is non-terminating, a widening operator ∇ is needed in addition to join operator ($\sqcup$). Due to the space limit, we omit details of the widening operator. A more complete introduction can be found in [37, 49].

4.2 The Abstract State

In KLAUS, we assume the kernel has a set of variables $V = \{v_1, \cdots, v_n\}$. Each variable $v \in V$ is mapped to $type(v)$ which is a set of annotated types. We use $type(v)$ to construct AWRPs: a write operation and a read operation are paired if the two pertaining variables share the same $type(v)$. More details will be discussed in Section 6. Depending on where the variables are stored, $type(v)$ is defined differently.

For local variables the life cycle of which starts at the function prologue and ends at the function epilogue, $type(v)$ = {'func+x'} where 'func' is the function name and 'x' is the offset of the local variable on the stack frame from the return address. For global or static variables which are valid since kernel bootup and across system calls, we directly use their variable name to denote $type(v)$. That is to say, for the variable like `static struct proto llcp_sock_proto`, its $type(v)$ is `llcp_sock_proto` rather than `struct proto`. For variables stored on the kernel heap (*e.g.*, SLAB/SLUB region and vmalloc region), we further divided them into three situations. If the variable is an individual object in a structure or union type, we denote it using the type name. If the variable is a structure or union field, we denote its $type(v)$ as 'type+x' where 'type' is the type name and 'x' is the field offset. If the variable is an ordinary buffer, its $type(v)$ is denoted as 'char*'. In our working example, KLAUS denotes the $type(v)$ of `llcp_sock->local`'s refcount as `nfc_llcp_local+0x18` because the type of `llcp_sock ->local` is `nfc_llcp_local` and the offset of the refcount is `0x18`.

Further, each variable $v \in V$ has its value, denoted $value(v)$, which is a set of $\langle cond, content \rangle$ tuples. This tuple indicates that under the condition $cond$, the value of v is equal to $content$. Both $cond$ and $content$ are represented as symbolic strings and KLAUS does not seek to evaluate it using SAT/SMT solvers. In our working example, after the kernel executes line 22, the $value(v)$ set of `llcp_sock->sock` is {⟨ 'llcp_sock ->ssap == LLCP_SAP_MAX', 'NULL' ⟩}. For readability, here, we use variable names from the source code. In our implementation, variable names are in SSA (Single Static Assignment) form.

Applying the above definitions, the abstract state S for our analysis over kernel functions is defined as $S = \{cond, \langle type(v_1), value(v_1) \rangle, \cdots, \langle type(v_n), value(v_n) \rangle\}$ where $cond$ is the path condition accumulated so far from the entry node.

4.3 The Transfer Function and Join Operator

To model the impact of executing *inst* in the state S, we let the transfer function be $Transfer(S, inst)$. The new state returns by the transfer function is $S' = \{cond', \langle type'(v_1), value'(v_1) \rangle, \cdots, \langle type'(v_n), value'(v_n) \rangle\}$.

Algorithm 2: Sketch of AWRPs Identifcation

```
1  Input: Kernel $K$ and Patch $P$ ;
2  Output: Pair[] ;
3  $funcWL$ = GetPatchedFunc($K$, $P$) ;
4  $I$ = { }, $Pair$ = { } ;
5  while $\exists$ <$F$, $F'$> $\in$ $funcWL$ do
6      $funcWL = funcWL$ - {<$F$, $F'$>} ;
7      $S$ = AbstractInterpret($F$) ;
8      $S'$ = AbstractInterpret($F'$) ;
9      foreach $n \in F'$ do
10         if $n \notin F$ or $S[n] \neq S'[n]$ then
11             $I = I \cup$ {n} ;
12         if $IsCall(n)$ then
13             if $\exists$ $arg$, $value(arg) \neq value'(arg)$ then
14                 $funcWL = funcWL \cup$ Callee(n) ;
15         if $IsRet(n)$ then
16             if $value(ret) \neq value'(ret)$ then
17                 $funcWL = funcWL \cup$ Caller(n) ;
18 foreach $n \in I$ do
19     if $IsWrite(n)$ then
20         m = FindRead($type(n)$, K) ;
21         $Pair = Pair \cup$ <n, m> ;
```

If $inst$ writes to a variable v, we update $value(v)$ by removing all tuples and adding a new tuple $\langle cond, content\rangle$ where $cond$ is the path condition in S and $content$ is the writing value. If $inst$ casts variable v from one type to another type, we update $type(v)$ by appending the new type. For any other variables $w \in V$ that are not written or casted, $value(w) = value'(w)$ and $type(w) = type'(w)$. If $inst$ is a conditional jump, $cond$ in S is conjuncted with the jump condition to produce $cond'$. For the instruction $inst$ that neither writes or casts a variable nor performs a conditional jump, $S' = S$.

We define the $Transfer$ for a sequence of instructions $insts = \{inst_0, inst_1, \cdots, inst_n\}$ as $Transfer(S, inst) = Transfer(\cdots(Transfer(S, inst_0), inst_1), \cdots, inst_n)$.

To avoid an exponential increase in state number, states computed along two paths are joined when the control flow merges. For $cond$ in the state, our join operator updates it through disjunction ($cond \cup cond'$). In this way, we can profile the patch condition change introduced by the patch (situation ❹ for AWRPs in Section 3.2). Similarly, to capture how variables are affected by the patch and thus construct AWRPs, for each variable $v \in V$, our join operator computes its $type(v)$ and $value(v)$ by combining the sets from two states respectively.

Formally, given two states $S = \{cond, \langle type(v_1), value(v_1)\rangle, \cdots, \langle type(v_n), value(v_n)\rangle\}$ and $S' = \{cond', \langle type'(v_1), value'(v_1)\rangle, \cdots, \langle type'(v_n), value'(v_n)\rangle\}$, we define $S'' = S' \cup S$ as $\{cond \cup cond', \langle type(v_1) \cup type'(v_1), value(v_1) \cup value'(v_1)\rangle, \cdots, \langle type(v_n) \cup type'(v_n), value(v_n) \cup value'(v_n)\rangle\}$.

4.4 Identification Algorithm

At a high level, the identification algorithm applies the abstract state, transfer function, and the join operator described above to abstract-interpret the kernel functions before and after patching. By using the differences in analysis results, we identify instructions the state of which are changed by the patch and pair write operation with read operation to construct AWRPs. Algorithm 2 is the sketch of our identification algorithm. In the following, we elaborate on more details.

Intra-procedural analysis. Given a kernel and a patch commit, `KLAUS` can easily find patched functions by parsing the diff file in the commit (line 3 in Algorithm 2). In our working example, the patched function is `llcp_sock_connect` according to line 8 in List 1. Our identification starts by applying Algorithm 1 to abstract-interpret these patched functions. More specifically, `KLAUS` analyzes these functions to get S and S' - abstract states before and after patching respectively (line 7-8). For every instruction n in the patched function F, it is an altered instruction if it is newly added in the patch (*i.e.*, $n \notin F$) or its state changes (*i.e.*, $S[n] \neq S'[n]$). We deem two states are different as long as the $type(v)$ or $value(v)$ of any one variable v has changed. Note that we do not consider instructions that are deleted by the patch. On the one hand, it is because the triggering of the new vulnerability needs not execute the deleted instruction. On the other hand, it is because the impact of deleting instructions can be modeled by abstract interpretation and is reflected in the $type(v)$ or $value(v)$ of influenced variables. Finally, we add instructions with altered states to the I set for AWRP construction (line 10-11).

Inter-procedural analysis. The state changes introduced to the patched functions will make impacts on other functions. For a variable v which is passed as an argument to a callee function, if its $value(v)$ changes, all computations in the callee function that depend on this argument will propagate the changes. Similarly, if a function returns a different $value(v)$ to the caller, the computations that use the returned value will change states as well. Therefore, we perform an inter-procedural analysis to further identify these changes. In our algorithm, we use a worklist to maintain functions worth analysis. In line 14 and 17 , we add affected callee and caller functions to the work list until finally, the analysis reaches a fixed-point (line 5).

AWRPs construction. Abstract Interpretation profiles the impact of the patch. Through the above analysis, we obtain a set of instructions I the states of which are altered. According to our discussion in Section 3.2, new vulnerabilities result from AWRPs. Therefore, we construct AWRPs from the instruction set I. Recall that the write and read operations in one AWRP pertain to the same variable. Here, we continue to over-approximate the construction. Instead of using memory alias analysis to determine whether two operations belong to the same variable, we examine whether the $type(v)$ of the written variable v and the $type(w)$ of the read variable w have an overlap. If they share a common annotated type, we will pair these two operations. As shown in line 20-21, for any write instruction in the set I, we will search the whole kernel to find a read instruction that satisfies our criteria. In our working example, the type of the variable `llcp_sock->local->ref` in function `llcp_sock_bind` is `nfc_llcp_local+0x18` which is also the type of the variable `sock->local->ref` in function `nfc_llcp_sock_free`. So, we pair the write operation of the former with the read operation for the latter, and thus construct an AWRP for our output.

5 AWRP-Driven Fuzzing

With the identified AWRPs in hand, as mentioned in Section 3.3, the second component of KLAUS is designed to perform directed fuzzing. Recall that AWRPs are used to describe the change of control and data flow introduced by a kernel patch. If the patch is problematic, by executing the code relevant to AWRPs, a kernel error could be potentially manifested. In this section, we discuss how KLAUS leverages AWRPs as guidance to search for input that could execute the kernel code pertaining to AWRPs and thus unveil the incorrectness of the corresponding kernel patch.

5.1 Two-Dimension Coverage

Before describing the coverage metrics in KLAUS, we need to clarify that AWRP is a necessary but insufficient condition for the new vulnerability. The AWRPs introduced by the patch are intended to fix the old vulnerability. There might not be a new vulnerability. Even if the new vulnerability exists, executing an AWRP does not guarantee the triggering of it unless the variable is assigned with the problematic value (*e.g.*, a large index to trigger the out-of-bound write) following a certain program path.

Therefore, our fuzzing goal is not only executing identified AWRPs in the order of write and read but also diversifying the writing value and execution path so that we can have more chances to trigger the potential new vulnerability. From this perspective, directed fuzzing that minimizes the distance between the target site and explored site is not an optimal solution for KLAUS. On the one hand, it is because the distance metric overemphasis reaching the target site and fails to enrich variable values along the execution paths. On the other hand, it is because fuzzing that minimizes distance can be easily stuck in a single path and thus is not generalized to consider paths that are longer but essential to manifesting new vulnerabilities.

Instead of relying on one dominating metric, KLAUS employs a two-dimension coverage strategy to execute code relevant to AWRPs. The first metric is type coverage. Recall that in AWRP identification, the write operation and the read operation are paired if their variables have overlapping in the $type()$ set. Without a doubt, the types in this set are of interest. Favoring inputs that operate kernel objects in these types, KLAUS sticks itself to code that is related to AWRPs so that it does not diverge in the huge kernel codebase. Compared with distance metric, type metric focuses the fuzzing on executing code related to AWRPs and meanwhile allows path exploration. The second metric in KLAUS is code coverage which is commonly used in traditional fuzzing. We do not discard this metric because the new code creates new values that can be assigned to the variables pertaining to AWRPs. With different values, especially extreme values such as a too-large index for buffer access, repeatedly executing write and read operations, KLAUS can ultimately trigger the potential new vulnerability.

5.2 Seed Selection

During the course of fuzzing, KLAUS maintains a matrix to filter out interesting seeds. In the matrix, each row represents an identified AWRP. Considering that there will be multiple variable instances that are corresponding to one write operation, in the column, KLAUS stores the pair of the variable address and the writing value.

When an interesting write operation is executed, we compare the address and the value with those already in the matrix. We will add one more pair to the matrix if either the address is new or the value is new. When an interesting read operation is executed, KLAUS examines whether the variable address and the read value exist in the matrix. If not, we will skip this time because to count an AWRP, the writing operation must be executed first.

With this matrix, we assign the highest priority to the input that either executes the interesting write operation with a new writing value or executes the interesting read operation after the corresponding write is executed. Any other inputs that increase type coverage or code coverage are also selected for the following round of fuzzing but not with the highest priority.

5.3 Resolving Branch Condition

The design of two-dimension coverage and seed selection keeps the fuzzing on an efficient way of discovering the new vulnerability. However, it does not guarantee that the fuzzing will not be hindered by branch conditions that are hard to satisfy.

Prior works mainly use two approaches to resolve this problem. The first approach is to do symbolic tracing from the system call entry and kick in SAT/SMT solvers to solve the accumulated constraints [42, 43, 62]. The second approach is to instrument the branch and implement a feedback mechanism to favor seeds that bring changes to the variables involved in the branch conditions [32, 35, 47]. However, for large-scale programs like kernel, as we will show in Section 7.3, both approaches have their limitations.

For symbolic tracing, though the size and complexity of its constraints are largely reduced compared with symbolic execution, it can go beyond the solvers' capacity from time to time. The major reason is the long calling chain in the kernel. For example, as the kernel is implemented layer by layer when a function in the abstract layer (*e.g.*, virtual file system) is called, the kernel will execute a series of functions until reaching a low-level layer (*e.g.*, USB device driver). It will take a very long time for the solver to satisfy the constraints collected in this procedure. In the worst case, the solver might fail to solve the constraints after wasting too much time.

Regarding the instrumentation-feedback mechanism, its main limitation is that it suffers from satisfying complex conditions. It is possible that our targeted write or read operation resides in a likely-taken branch. So its complex conditions can be easily satisfied with the feedback from instrumentation. However, when it is in a rare branch, the condition can be hard to satisfy. To this end, we experiment with both approaches in `KLAUS` and thus determine which branch-solving method is more suitable to our problem. We show the evaluation of both methods in Section 7.

6 Implementation

Below, we delve into the specifics of our implementation.

6.1 Abstract Interpretation

In Section 4, we explain the abstract interpretation at the source code level. The implementation using LLVM brings about certain challenges, which we address below.

First, our AWRP identification process begins with functions that have been altered by the patch. These functions can easily be obtained from the commit. However, when represented at the LLVM IR level, they may have been inlined into their caller functions. To ensure that these caller functions are also considered in our implementation, we utilize `llvm-diff` to identify the patched functions at the LLVM IR level.

Second, `KLAUS` computes the $type()$ of variables based on the memory region where they are stored. Global variables are easy to identify as their names are unique and present in the LLVM IR. To determine the $type()$ of local variables stored on the stack, `KLAUS` relies on debugging information to obtain the function names and offsets, which are then combined into the format `'func+x'`. For variables stored in the heap, `KLAUS` determines the parent structure's type and the variable's offset within the structure using the `Getelementptr` instruction. This instruction is responsible for getting a field element from a structure using a pointer. Therefore, it has an operand referring to the parent structure and an operand indicating the offset of the field. At the source code level, each variable has its own memory region, making it possible to compute its $type()$ using the above approach. However, in the LLVM IR, some variables are temporary and only exist for the SSA form. In our implementation, we address temporary variables by initially assuming their $type()$ is blank and later computing it as the union of the $type()$ of all operands in the instruction that defines the temporary variable.

Third, `KLAUS` faces a challenge as the LLVM IRs generated from the unpatched and patched kernels are not identical. For example, a variable in the same function may have the label `%var1` in the IR of the unpatched kernel and `%var2` in the IR of the patched kernel. To accurately identify AWRPs, `KLAUS` needs to know if the $type()$ and $value()$ of a variable have changed due to the patch. To overcome this, `KLAUS` maps variables in both LLVM IRs back to their source code level representation using debugging information. This way, `KLAUS` can compare variables' $type()$ and $value()$ and determine if their corresponding write or read operations should be added to the AWRP set. It is worth noting that in certain situations, static variables across different modules may share the same name, leading to potential name collisions. To avoid this from impacting the accuracy of our analysis, we automatically annotated each static variable with module names for differentiation.

Finally, in our implementation, our AWRP identification process first performs an intra-procedural analysis before it spreads the effects of the patch across functions. To handle the `Call` instruction, we treat it as a `Store` operation and consider the called function as a variable. The $value()$ of this pseudo-variable changes if the called function returns a different $value()$. This change is then propagated in an iterative manner, with a list of waiting functions being kept track of, until the $type()$ and $value()$ of all variables reach a fixed point.

6.2 AWRP-Driven Fuzzing

Recall that `KLAUS` experiments two branch-resolving mechanisms to resolve branch conditions – ❶ symbolic tracing, which traces the system call entry symbolically and utilizes an SAT/SMT solver to generate the necessary inputs, and ❷ branch instrumentation, that selects and alters the input seed based on how critical variables are affected.

For symbolic tracing, our implementation follows the method described in HFL [42]. It forks a process specifically for constraint collection and resolution when a branch is touched but has not been entered too many times. For the instrumentation-based mechanism, our implementation instruments critical variables in the kernel during compilation so that the fuzzer can track changes to these variables when the instrumented sites are executed.

In our implementation, two types of variables are treated as critical variables: those directly used in the branch conditions, and those that have a data flow towards the first type of variables. The latter is included in our feedback mechanism because they have the potential to affect the value of the first type of variables. After capturing changes to the critical variables, our implementation forces the system to revisit how the input was mutated from the previous seed. This allows the system to examine whether the change of the critical variable stems from the mutation of a specific system call argument. If so, this implies an implicit dependence between the argument and the critical variable, and our implementation forks a process to focus on mutating this specific argument to accelerate branch taking.

After KLAUS triggers a kernel error during its directed fuzzing process, it is possible that the error is not caused by the kernel patch under examination. To address this, KLAUS's implementation also includes a triaging component to determine the root cause of the error. If the same error occurs in the unpatched kernel when the patch is removed and the input is replayed, the input is discarded and treated as not being relevant to the patch. If the error does not occur in the unpatched kernel, we conclude the patch is incorrect. If the error occurs but is different from the error prior to the patch removal, a manual validation procedure is performed to confirm the relevance of the error to the patch. More information on the manual validation process can be found in the Appendix.

7 Evaluation

In this section, we first discuss how we construct a ground truth dataset. Then, we describe how to use that dataset to evaluate the effectiveness and efficiency of KLAUS comprehensively and thoroughly. Lastly, we demonstrate the utility of KLAUS on more Linux kernel patches (the correctness of which is unknown).

7.1 Dataset

Evaluating KLAUS requires a dataset of incorrect Linux kernel patches. Although 3,000+ incorrect patches were gathered and 5% of them were manually analyzed, using them as the ground truth dataset is not feasible. The tool was developed on GCC 9.0 and LLVM 12, limiting the compatibility with some old-version kernels and restricting the available incorrect patches [1]. Also, KLAUS requires the PoC program that triggers the original kernel bug, which was not available for some patches. Thus, only 23 patches could be used as the ground truth dataset. To overcome this shortage, 250 recently released patches were randomly selected for testing. These patches were compatible with the implementation and had available PoC programs for triggering the original bugs. If KLAUS could detect incorrectness in these patches, it would demonstrate the critical utility of the proposed technique.

```
void bpf_tcp_close(struct sock *sk) {
-- write_lock_bh(&sk->sk_callback_lock);
    smap_release_sock(psock, sk); // free psock
-- write_unlock_bh(&sk->sk_callback_lock);
}

int sock_map_ctx_update_elem(){
-- write_lock_bh(&osock->sk_callback_lock);
    smap_list_map_remove(opsock);
    smap_release_sock(opsock, osock);
-- write_unlock_bh(&osock->sk_callback_lock);
}

void smap_list_map_remove(struct smap_psock *psock
     )
{
++  spin_lock_bh(&psock->maps_lock);
}
```

Listing 3: The snippet of commit e9db4ef6bf4c.

7.2 Experiment Setup

Using the ground truth dataset described above, we compared the effectiveness of KLAUS with that of the most commonly used Linux kernel fuzzing tool – Syzkaller [52]. Besides, we evaluate how the two different branch-resolving mechanisms – instrumentation-based and symbolic tracing-driven solutions – benefit incorrect patch identification. Following this effort, we also study how the two AWRP-based coverage mechanisms each individually contribute to KLAUS's effectiveness in pinpointing incorrect kernel patches.

For all of our experiments, we ran Ubuntu 20.04 LTS on a machine with an AMD EPYC 7702 64-Core CPU and 384GB RAM. We enabled KASAN as a sanitizer for error detection. For each virtual machine used by our fuzzer, we allocated 2 virtual CPU cores and 2GB of RAM and ran the fuzzer for 5 rounds with each round for 3 days We collected the results from the fuzzer logs and other files in the fuzzer's work directory. If nothing interesting is found in all rounds, we mark the result as N/A. Otherwise, we calculate the average time by treating N/A in some rounds as the maximum time. To ensure fairness, we give all fuzzers the same initial seed. Our initial seed was derived from the PoC that triggers the original bug.

7.3 Performance in Ground Truth Dataset

Recall that KLAUS employs abstract interpretation to identify altered read-write pairs. However, the implementation of abstract interpretation tends to over-approximate the read-write pairs that could potentially be relevant to a patch. To assess the extent of this over-approximation, our evaluation measures the accuracy of the identified read-write pairs by comparing them with the true pairs relevant to a patched code fragment. Any identified pairs that are unrelated to the patched code fragment are considered false positives.

[1] Note that this is only a limitation of the implementation, our methodology is generic.

Case ID	KLAUS $_{B.I.}$	Syzkaller	KLAUS $_{N.R.}$	KLAUS $_{S.T.}$	KLAUS $_{B.I.}$**(co)**	KLAUS $_{B.I.}$**(to)**	Over-approx.
730c5fd42c1e [20]	6m	18m	37m	9m	240m	< 1m	0.08
7f700334be9a [6]	< 1m	< 1m	< 1m	< 1m	< 1m	< 1m	0.50
8f5c5fcf3533 [26]	2m	N/A	1m	< 1m	2m	N/A	0.50
951c6db954a1 [22]	180m	480m	25m	955m	360m	N/A	0.00
95fa145479fb [4]	1080m	N/A	1902m	2989m	N/A	N/A	0.12
9ebeddef58c4 [21]	44m	1680m	60m	119m	60m	N/A	0.00
b196d88aba8a [27]	< 1m	< 1m	< 1m	< 1m	< 1m	< 1m	0.00
c3e2219216c9 [2]	< 1m	4260m	4m	1535m	N/A	9m	0.50
d10523d0b3d7 [17]	4m	3960m	8m	7m	4m	9m	0.00
e5e1a4bc916d [31]	< 1m	< 1m	< 1m	< 1m	< 1m	< 1m	0.00
e9db4ef6bf4c [3]	720m	N/A	N/A	51m	N/A	N/A	0.00
009bb421b6ce [29]	< 1m	N/A	< 1m	< 1m	< 1m	N/A	0.75
0fedc63fadf0 [13]	2665m	N/A	N/A	N/A	3518m	3800m	0.72
100f6d8e0990 [9]	< 1m	1m	< 1m	< 1m	< 1m	< 1m	0.70
1548bc4e0512 [30]	451m	N/A	199m	N/A	1m	3488m	0.12
301428ea3708 [16]	< 1m	< 1m	< 1m	< 1m	< 1m	< 1m	0.00
304e024216a8 [12]	21m	N/A	3m	1m	< 1m	62m	0.50
44d4775ca518 [14]	1835m	N/A	N/A	4212m	N/A	N/A	0.00
6289a98f0817 [23]	< 1m	< 1m	< 1m	< 1m	< 1m	< 1m	0.00
6a21dfc0d0db [19]	3m	1200m	1m	62m	5m	N/A	0.00
6d6dd528d5af [15]	592m	N/A	54m	86m	2904m	1141m	0.60
7a68d9fb8510 [28]	4116m	N/A	657m	3474m	N/A	658m	0.00
c47cc304990a [10]	24m	20m	17m	3m	6m	7m	0.06
total #	23	13	20	21	18	15	N/A

Table 1: The performance of KLAUS. Note that the case ID represents the commit ID of the incorrect patch. The number in the table indicates the time spent for a corresponding fuzzing method to pinpoint an incorrect patch in minutes. "N/A" denotes the fuzzing method fails to find the patch's incorrectness within 3 days. The last row of the table shows the total number of incorrect patches that the corresponding fuzzing method unveils in a 3-day period. KLAUS $_{B.I.}$ and KLAUS $_{S.T.}$ mean our fuzzer that utilizes Branch Instrumentation and Symbolic Tracing to resolve branch conditions, respectively. KLAUS $_{N.R.}$ means that no branch condition resolving technique is used. KLAUS $_{B.I.}$(to – type only), KLAUS $_{B.I.}$(co – code only) represent two different coverage guidance implementations – KLAUS $_{B.I.}$(type only): KLAUS $_{B.I.}$ that uses only the type information pertaining to AWRP to guide the fuzzing process; KLAUS $_{B.I.}$(code only): KLAUS $_{B.I.}$ that employs only the code relevant to AWRP for fuzzing guidance. Over-approx. represents the rate of the read-write pairs that KLAUS mistakenly identifies.

In Table 1, the last column illustrates the false positive rate of AWRP. The results demonstrate that among the 23 cases analyzed, 11 cases have no false positives, and the majority of cases exhibit a false positive rate below 0.5. This indicates that the impact of over-approximation on our fuzzing results is minimal, although it may slightly reduce the efficiency of our fuzzing process.

In addition to the evaluation of our static analysis technique used in KLAUS, we also evaluate the dynamic component of KLAUS. Our fuzzing method recorded the time spent on each test case where an error was triggered by an incorrect patch. As seen in Table 1, our approach outperforms the traditional kernel fuzzing tool, Syzkaller. By using symbolic tracing and branch instrumentation, respectively, KLAUS accurately identified incorrectness in 21 and 23 kernel patches, while Syzkaller only found 13. The reason for this improvement is shown in Figure 2 (see Appendix B). Incorrect patches are often caused by AWRP, and our method showed higher coverage of AWRP, leading to a greater ability to detect patch errors.

As is described in Section 5.3, we experiment with two methods to solve branch conditions – symbolic tracing and branch instrumentation. Table 1 shows the performance of KLAUS when integrating the corresponding approach. As we can observe, KLAUS with the integration of branch instrumentation slightly outperforms that with symbolic tracing in terms of the ability to pinpoint incorrect patches (23 vs. 21). In addition, we also observed that, for many test cases, KLAUS with branch instrumentation spends less time on exposing the patch incorrectness. The reason is that, when enabling symbolic tracing, KLAUS needs more resources to perform constraint solving. While symbolic tracing, to some extent, helps the fuzzer bypass branch predicts more efficiently, it also plays the role of a double-edged sword, slowing down the process by which our fuzzer evaluates inputs. This implies that the branch-instrumentation method is more suitable for kernel patch quality assessment.

In addition to the performance that KLAUS exhibits in different branch-solving mechanisms, Table 1 also shows KLAUS's performance when it was implemented with other proposed techniques. Recall that, in addition to the two branch-solving schemes, KLAUS is also integrated with an AWRP-based two-dimension coverage mechanism to enhance incorrect patch identification. As is elaborated in Section 5.3, the two-dimension coverage scheme contains two parts. One is to utilize the code relevant to AWRP as coverage to guide KLAUS. The other is to employ the type relevant to AWRP as coverage to drive our fuzzer. In our experiment, we break down the impact of each coverage mechanism and show their corresponding performance in Table 1.

As we can observe from the table, when disabling branch resolving method and enabling both coverage mechanisms (i.e., utilizing two-dimension coverage alone), KLAUS demonstrates a slight decrease in tracking down incorrect patches (20 vs. 23). It indicates that the branch-resolving mechanism is helpful for improving incorrect patch identification. However, it is not a key driving force for the success of KLAUS.

Looking at the results depicted in the 6th and 7th column of Table 1, we can discover that by using the code relevant to AWRP as the coverage guidance alone, KLAUS could exhibit significantly better performance than Syzkaller (18 vs. 13). In contrast, the similar performance gain does not manifest when KLAUS utilizes only the type information pertaining to AWRP as coverage guidance (15 vs. 13). This indicates AWRP-code-based coverage is the key driving force for incorrect patch identification. Besides, the 2nd column implies that AWRP-type-based coverage could be used as a complementary component to improve incorrect patch discovery further.

Case Study. To demonstrate that AWRP identification could benefit incorrect patch detection, we use commit e9db4ef6bf4c as a case study. In List 3, bpf_tcp_close and sock_map_ctx_update_elem functions removed the lock operations over sk_callback_lock. However, this code change breaks the mutual exclusion between the two functions, resulting in data races on the psock. With this data race, if smap_list_map_remove is executed after smap_release_sock in bpf_tcp_close, the psock will be freed in bpf_tcp_close, leading to a null pointer dereference error in smap_list_map_remove.

In the case above, the AWRP contains the read operation on the lock in bpf_tcp_close and the write operation on the lock in sock_map_ctx_update_elem. As such, KLAUS instrumented the two sites and directed the fuzzing toward them. By focusing on the two sites, KLAUS increases the likelihood of triggering the data race, which manifests incorrectness. In comparison, the baseline fuzzers lack this guidance and are easily distracted from the two sites, missing the opportunity of triggering the data race.

```
1  struct sock *sock_clone(struct sock *sk) {
2      struct sock *newsk = inet_csk_clone_lock(sk);
3      ...
4      return newsk;
5  }
6
7  int sys_disconnect(struct sock *sk) {
8  -- free(sk->uaf);
9  -- sk->uaf = NULL;
10 }
11
12 int sys_connect(struct sock *sk) {
13     struct sock *clinet = sock_clone(sk);
14 }
15
16 int sys_close(struct sock *sk) {
17     free(sk->uaf);
18     free(sk);
19 }
20
21 void sk_timer_func(struct sock *sk) {
22     // accessing sk->uaf
23     sk->uaf->a = 1;
24 }
```

Listing 4: A simplified kernel code fragment illustrating patch incorrectness.

7.4 Performance in the Wild

Recall that the examination of KLAUS' utility involved running it against 250 randomly selected kernel patches with unknown correctness. We discover that KLAUS identified 30 incorrect patches, accounting for 12% of all the patches under our testing. It is worthy of noting that this percentage is higher than that reported in our empirical study (about 6% reported in Section 3.1). We argue that this difference is presumably because of our sampling bias. The test cases in our evaluation were randomly selected from recently released patches with most of them in "holding-area" branches like dev and next. The code quality of them is typically worse than the upstream branch which is used in our empirical study.

For the 30 incorrect patches, we promptly reported to the Linux community. To date, the community has confirmed and fixed 25 of these patches. We manually evaluated the bugs introduced by the 30 incorrect patches and found that 3 are exploitable across various Linux distributions, including Ubuntu and Android. Interestingly, two of the bugs that the patch intended to fix were not likely to be exploitable, highlighting the danger of incorrect patches which can increase security risks. Due to page limitations, we use only one of the incorrect patches as an example to showcase the change in exploitability.

Case Study II. The example in List 4 presents a simplified code snippet highlighting the vulnerability caused by an incorrect patch. The function sock_clone (line 1) is used to clone a sock object, which includes a pointer sk->uaf used for creating a client socket upon new connection establishment (line 13). The function `sys_disconnect` is designed to free `sk->uaf` and set it to null (lines 8 and 9) when the socket is disconnected. In the Linux kernel, freeing a null pointer is acceptable and will not cause any issues.

However, if a timer is set on the socket, the `sk_rest_timer` function will be activated, accessing `sk->uaf`, which may cause problems if it has already been freed. Imagine a scenario where the timer function is being executed by the kernel while the user closes the socket from the user space, triggering `sys_disconnect` to free `sk->uaf`. In such a case, the timer function will attempt to access the dangling pointer `sk->uaf`, resulting in a use-after-free vulnerability.

The resolution of this vulnerability seemed straightforward initially. Three years ago, kernel developers removed the freeing of `sk->uaf` in `sys_disconnect` (lines 8 and 9) to eliminate the race condition, as reflected in the code snippet. However, upon testing the patch using KLAUS, it was discovered to be incorrect, and the bug introduced by this patch was even more susceptible to exploitation.

By delving into KLAUS running against the simple patch above, we observe that KLAUS first detects the altered variable `sk->uaf` from deleted lines 8 and 9. Using the type information associated with `sk->uaf`, KLAUS then conducts inter-procedure analysis and identifies related variables in functions `sock_clone`, `sys_close`, and `sk_timer_func`. These identified variables together form the AWRPs, guiding KLAUS's fuzzing component to proceed as follows.

First, the fuzzer executes `sys_disconnect`, causing the altered write on `sk->uaf`. The annotated pair then directs the fuzzer to run the `sk_clone` function, where the copy of `sk->uaf` is made. After the socket is closed, executing `sys_close` and freeing `sk->uaf`'s memory, the leftover socket's `sk->uaf` becomes a dangling pointer. Compared with the dangling pointer introduced by the original kernel bug, the dangling pointer caused by the incorrect patch is more stably exploitable because it does not require a critical race to trigger the use-after-free.

8 Related Work

Patch correctness analysis. Prior works have explored various methods to evaluate the quality of bug patches. For example, Gu *et.al.*proposed Fixation, a system that uses distance-bounded weakest precondition to identify partially fixed exceptions in Java programs [39]. Kim *et.al.*introduced the concept of bug neighborhood to check for persistence of NULL pointer references after patch application [44]. Le and Pattison proposed a new program representation, the multi-version interprocedural control flow graph, which describes control flow variations between different software versions. By performing path-sensitive symbolic analysis on the graph, they showed that their representation could efficiently verify patches [45].

The prior works mentioned are not applicable to our problem. They were designed to address a specific error (e.g. pinpointing exceptions in Java programs [39] or NULL pointer references [44]), while our goal is to uncover incorrect patches for various memory corruption errors. The prior research focuses on examining the completeness of the given patch, but our aim is to not only identify partially fixed bugs but also to expose new bugs or vulnerabilities introduced by the patch. Lastly, the prior research focuses on examining problematic bug fixes in userspace programs, whereas our goal is to unveil incorrect kernel patches. The complexity of the kernel limits the utility of many program analysis techniques (such as symbolic execution adopted in [45]).

Directed Fuzzing. In the past, various directed fuzzing methods have been introduced. AFLGo [33] prioritizes the input seed with the shortest path to the target program site at the basic block level. Hawkeye [34] calculates the similarity between the input seed and the potential execution trace, and uses this similarity to guide the selection of input seeds. To perform directed fuzzing on the Linux kernel, SemFuzz [57] combines NLP techniques with program analysis. KATCH [48] selects the input with the shortest distance to the target, and then uses symbolic execution. Savior [36] only employs symbolic execution if it visits branches that can reach targets with potentially buggy code. BEACON [40] calculates preconditions for reaching the target and prunes paths that cannot reach the target, avoiding wasted computational resources. FuzzGuard [61] trains a predictor classifier and uses it to prioritize inputs more likely to reach bugs. Regression greybox fuzzing [60] proposes a new power scheduling mechanism, favoring seeds that cover frequently changing code.

Unlike prior directed fuzzing techniques, our approach utilizes a distinct feedback mechanism to guide the fuzzer towards the target code. Our directed fuzzing method prioritizes not just the reachability of the target code, but also diversifying the paths towards it, resulting in a wider exploration of execution contexts. Additionally, our directed fuzzing takes into account the sequential reachability of the code.

9 Conclusion

Incorrect kernel patches can be problematic, transforming a non-exploitable bug into a highly exploitable vulnerability. Our research discovers that a kernel fuzzer, which uses alteration to the variables' read-write operations as a guide, has the potential to be a more effective tool for pinpointing incorrect kernel patches. We believe that an automated method for tracking down incorrect kernel patches could greatly improve the overall quality of patches for the Linux kernel. As a next step in our research, we plan to explore the identification of incorrect patches for userland programs. This will help to ensure the reliability and security of the Linux kernel and its userland programs, a crucial component of modern computing.

References

[1] 0-day CI. https://www.intel.com/content/www/us/en/developer/articles/technical/0-day-ci-linux-kernel-performance-report-v5-3.html.

[2] block: free sched's request pool in blk_cleanup_queue. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c3e2219216c9.

[3] bpf: sockhash fix omitted bucket lock in sock_close. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e9db4ef6bf4c.

[4] bpf: sockmap/tls, close can race with map free. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=95fa145479fb.

[5] CVE-2021-23134. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-23134.

[6] ip6_gre: proper dev_{hold|put} in ndo_[un]init methods. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=7f700334be9a.

[7] Kernel CI. https://kernelci.automotivelinux.org/.

[8] Linux kernel selftests. https://www.kernel.org/doc/Documentation/kselftest.txt.

[9] net: correct zerocopy refcnt with udp msg_more. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=100f6d8e0990.

[10] net: kcm: fix memory leak in kcm_sendmsg. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c47cc304990a.

[11] net/nfc: fix use-after-free llcp_sock_bind/connect. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c61760e6940d.

[12] net_sched: add a temporary refcnt for struct tcindex_data. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=304e024216a8.

[13] net_sched: commit action insertions together. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=0fedc63fadf0.

[14] net/sched: sch_taprio: reset child qdiscs before freeing them. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=44d4775ca518.

[15] net/smc: fix refcount non-blocking connect -part 2. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6d6dd528d5af.

[16] net/smc: fix refcounting for non-blocking connect. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=301428ea3708.

[17] net/tls: free the record on encryption error. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d10523d0b3d7.

[18] nfc: fix refcount leak in llcp_sock_connect. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8a4cd82d62b5.

[19] Rdma/ucma: Limit possible option size. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6a21dfc0d0db.

[20] rxrpc: Fix local endpoint refcounting. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=730c5fd42c1e.

[21] rxrpc: rxrpc_peer needs to hold a ref on the rxrpc_local record. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=9ebeddef58c4.

[22] sctp: fix memleak on err handling of stream initialization. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=951c6db954a1.

[23] sit: proper dev_hold|put in ndo_[un]init methods. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6289a98f0817.

[24] Syzkaller dashborad. https://syzkaller.appspot.com/.

[25] Syztest. https://github.com/google/syzkaller/tree/master/pkg/runtest.

[26] tipc: call start and done ops directly in __tipc_nl_compat_dumpit. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8f5c5fcf3533.

[27] tun: fix use after free for ptr_ring. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b196d88aba8a.

[28] Usb: usbdevfs: sanitize flags more. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=7a68d9fb8510.

[29] workqueue, lockdep: Fix an alloc_workqueue error path. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=009bb421b6ce.

[30] xfrm: policy: delete inexact policies from inexact list on hash rebuild. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=1548bc4e0512.

[31] xsk: Fix possible memory leak at socket close. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e5e1a4bc916d.

[32] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, 2020.

[33] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[34] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[35] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy*, 2018.

[36] Yaohui Chen, Peng Li, Jun Xu, Shenjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. Savior: Towards bug-driven hybrid testing. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, 2020.

[37] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1977.

[38] Navid Emamdoost, Qiushi Wu, Kangjie Lu, and Stephen McCamant. Detecting kernel memory leaks in specialized modules with ownership reasoning. In *Proceedings of The Network and Distributed System Security Symposium*, 2021.

[39] Zhongxian Gu, Earl T. Barr, David J. Hamilton, and Zhendong Su. Has the bug really been fixed? In *Proceedings of the 32nd International Conference on Software Engineering*, 2010.

[40] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Beacon: Directed grey-box fuzzing with provable path pruning. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy*, 2022.

[41] Hyungsub Kim, Muslum Ozgur Ozmen, Z Berkay Celik, Antonio Bianchi, Dongyan Xu, Ruoyu Wu, Taegyu Kim, Dave Jing Tian, Dongyan Xu, Raymond Muller, et al. Patchverif: Discovering faulty patches in robotic vehicles. In *Proceedings of the 32nd USENIX Security Symposium*, 2023.

[42] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kernel. In *Proceedings of The Network and Distributed System Security Symposium*, 2020.

[43] Kyungtae Kim, Taegyu Kim, Ertza Warraich, Byoungyoung Lee, Kevin RB Butler, Antonio Bianchi, and Dave Jing Tian. Fuzzusb: Hybrid stateful fuzzing of usb gadget stacks. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy*, 2022.

[44] Mijung Kim, Saurabh Sinha, Carsten Görg, Hina Shah, Mary Jean Harrold, and Mangala Gowri Nanda. Automated bug neighborhood analysis for identifying incomplete bug fixes. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, 2010.

[45] Wei Le and Shannon D Pattison. Patch verification via multiversion interprocedural control flow graphs. In *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[46] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[47] Zhenpeng Lin, Yueqi Chen, Dongliang Mu, Chengsheng Yu, Yuhang Wu, Xinyu Xing, and Kang Li. Grebe: Facilitating security assessment for linux kernel bugs. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy*, 2022.

[48] Paul Dan Marinescu and Cristian Cadar. Katch: High-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013.

[49] Antoine Miné. The octagon abstract domain. In *Proceedings of the 8th Working Conference on Reverse Engineering*, 2001.

[50] Nick L Petroni Jr and Michael Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[51] Xin Tan, Yuan Zhang, Xiyu Yang, Kangjie Lu, and Min Yang. Detecting kernel refcount bugs with two-dimensional consistency checking. In *Proceedings of the 30th USENIX Security Symposium*, 2021.

[52] Dmitry Vyukov. Syzkaller, 2016. https://github.com/google/syzkaller.

[53] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V Krishnamurthy, and Nael B Abu-Ghazaleh. Syzvegas: Beating kernel fuzzing odds with reinforcement learning. In *Proceedings of the 30th USENIX Security Symposium*, 2021.

[54] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *Proceedings of the 42nd International Conference on Software Engineering*, 2020.

[55] Qiushi Wu, Yue Xiao, Xiaojing Liao, and Kangjie Lu. OS-Aware vulnerability prioritization via differential severity analysis. In *Proceedings of the 31st USENIX Security Symposium*, 2022.

[56] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011.

[57] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[58] Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V Krishnamurthy, and Paul Yu. Ubitect: a precise and scalable method to detect use-before-initialization bugs in linux kernel. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[59] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. StateFuzz: System Call-Based State-Aware linux driver fuzzing. In *Proceedings of the 31st USENIX Security Symposium*, 2022.

[60] Xiaogang Zhu and Marcel Böhme. Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[61] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. Filtering out unreachable inputs in directed greybox fuzzing through deep learning. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[62] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian. SyzScope: Revealing High-Risk security impacts of Fuzzer-Exposed bugs in linux kernel. In *Proceedings of the 31st USENIX Security Symposium*, 2022.

A Procedure of Manual Validation

In Section 6, it was stated that when a kernel error is encountered against an input (e.g., a PoC program), on a patched kernel, we revert the patch and rerun the input on the unpatched kernel. If the error occurs on the unpatched kernel and is different from the one observed on the patched kernel, it is referred to as manual validation. This process involves manually verifying if the error is a result of the patch. Below, we outline the steps involved in our human validation procedure.

When KLAUS generates a kernel error report, we first run the associated PoC program and capture a detailed record of the kernel execution using ftrace. If the kernel execution does not include the patch's code changes, we conclude that the patch is correct since the relevant code was not executed. In other cases, from the error report, we extract the series of kernel function calls leading up to the error, along with the relevant kernel object. Starting from the latter, we perform manual backward analysis to pinpoint the root cause of the error. This enables us to determine the correctness of the patch and conclude our human validation process.

B AWRP Coverage

Figure 2 shows the coverage of AWRP across three different kernel fuzzers – – Syzkaller, KLAUS implemented with symbolic tracing, KLAUS implemented with branch instrumentation.

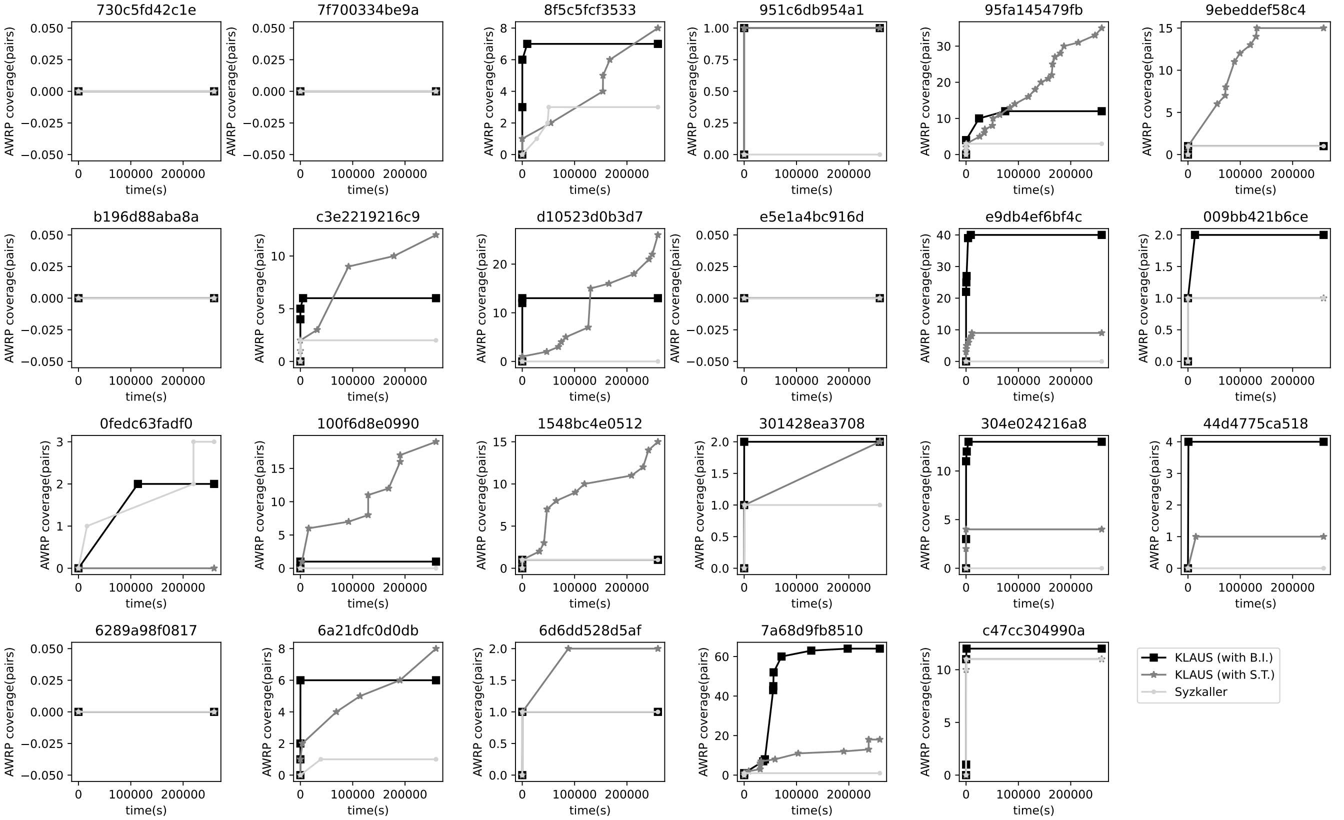

Figure 2: The coverage of AWRP across three different kernel fuzzers – Syzkaller, KLAUS implemented with symbolic tracing (denoted by KLAUS with B.I), KLAUS implemented with branch instrumentation (denoted by KLAUS with S.T.). Note that the x-axis represents the time spent on fuzzing, and the y-axis indicates the AWRP coverage.