Undo Workarounds for Kernel Bugs

Seyed Mohammadjavad Seyed Talebi*, Zhihao Yao*
Ardalan Amiri Sani*, Zhiyun Qian†, Daniel Austin‡
**UC Irvine,* †*UC Riverside,* ‡*Atlassian*[*]

Abstract

OS kernels are full of bugs resulting in security, reliability, and usability issues. Several kernel fuzzers have recently been developed to find these bugs and have proven to be effective. Yet, bugs take several months to be patched once they are discovered. In this window of vulnerability, bugs continue to pose concerns. We present workarounds for kernel bugs, called bowknots, which maintain the functionality of the system even when bugs are triggered, are applicable to many kernel bugs, do not cause noticeable performance overhead, and have a small kernel footprint. The key idea behind bowknots is to undo the side effects of the in-flight syscall that triggers a bug, effectively neutralizing the syscall. We also present a static analysis tool, called Hecaton, that generates bowknots automatically and inserts them into the kernel. Through extensive evaluations on the kernel of Android devices as well as x86 upstream kernels, we demonstrate that bowknots are effective in mitigating kernel bugs and vulnerabilities. We also show that Hecaton is capable of generating the right bowknots fully automatically in majority of cases, and requires minimal help from the analyst for the rest. Finally, we demonstrate the benefits of bowknots in improving the efficiency of kernel fuzzing by eliminating repetitive reboots.

1 Introduction

Commodity OS kernels are monolithic, large, and hence full of bugs. Bugs in the kernel cause important problems. First, they risk the system's security as some bugs might be exploitable vulnerabilities. The kernel is a highly privileged layer in the system software stack and hence is attractive to attackers. Indeed, OS kernels are hot targets for security attacks these days. For example, according to Google, an increasing number of attacks on mobile devices are targeting the kernel (i.e., 44% of attacks in 2016 vs. 9% and 4% of them in 2015 and 2014, respectively) [9]. Second, they impact the reliability and usability of the system. Even a simple crash bug, e.g., a null pointer dereference, results in a system hang or reboot, causing usability issues for the users. Even worse, bugs can corrupt the state of the software and hardware and lead to unexpected behavior. Finally, as we will show, kernel bugs can even pose practical challenges for kernel fuzzing by inducing repetitive reboots and wasting the fuzzing time.

The common practice today is to find these bugs and patch them. There has been a lot of progress recently to automate the first step (i.e., finding bugs). More specifically, several kernel fuzzers have been recently developed such as Syzkaller [13], kAFL [36], Digtool [32], and MoonShine [31]. Indeed, these fuzzers have been successfully used to find bugs in the kernel [10, 12, 37]. However, the second step (i.e., patching bugs) remains a highly manual and lengthy process. In practice, this requires reporting the bug to the developers of the code, e.g., the vendor in charge of a device driver, and waiting for a patch. Unfortunately, this wait can take months for the bug to sit in a queue, be evaluated by developers, and get a patch developed, tested, and merged into the kernel. Our study of bugs found by Syzkaller [12] shows that bugs have taken on average 66 days to be patched. Moreover, at the time of the study (November 2019), there were several open bugs that were waiting for a patch for an average of 233 days. While waiting for a patch, the kernel remains vulnerable.

In this paper, we introduce workarounds for kernel bugs before they are correctly patched. We refer to such a workaround as a *Bug undO Workaround for KerNel sOlidiTy (bowknot)*. A bowknot has five important properties. First, it is fast to generate. Unlike a proper patch for a bug that takes months to be ready, a bowknot takes at most a few hours. Second, it is designed to maintain the system's functionality even if the bug is triggered[1]. Kernel bugs almost always are triggered when unanticipated syscalls are issued, either by mistake by a faulty application or intentionally by malware. A bowknot undoes the side effects of this faulty or malicious syscall in-

[*]This research started while Mr. Austin was with Google, as part of the Android Security team.

[1]In the paper, we use the term "trigger a bug" to mean either executing buggy code or triggering a kernel sanitizer warning (or even a manual check) right before executing buggy code. See §4.1 for more details.

vocation, allowing the kernel to continue to correctly serve well-structured syscalls. Third, a bowknot does not require any special hardware support, e.g., power management support in a driver needed for checkpointing (§8), and hence is applicable to a large number of bugs in various devices. Fourth, a bowknot does not add any noticeable performance overhead. This is because it does not do much as long as the bug is not triggered. Only when the bug is triggered, it is invoked to undo its side effects. Finally, a bowknot requires small changes to the kernel. It requires modifications only to the functions in the execution path that triggers the bug.

The key idea behind a bowknot is to undo the effects of the syscall that triggers a bug. In other words, when a syscall is issued and triggers a bug, the bowknot gets activated and neutralizes the effects of that syscall. Undoing the syscall at arbitrary points of execution is challenging since not only a syscall can affect the kernel memory state, it can even change the state of I/O devices, e.g., a camera. The latter is especially important for device drivers, which contain most of the kernel bugs (e.g., 85% of bugs in Android kernels [44]). To address this problem, we leverage existing undo statements in error handling blocks in the kernel to generate the right undo blocks for the functions in the execution path of the bug.

Bowknots, as described, achieve all the aforementioned properties, except for one. More specifically, generating a bowknot manually, while feasible, is challenging and time-consuming. Therefore, to satisfy this requirement, we introduce Hecaton, a static analysis tool that helps generate bowknots automatically[2]. Hecaton analyzes the whole kernel to find the relationship between state-mutating statements in the kernel and their corresponding undo statements in error handling basic blocks. It then uses this knowledge to generate the right undo block for the function containing the bug and the parent functions in the call stack. It also automatically inserts the undo blocks into the kernel. Due to the limitations discussed in §5.3, in some cases, Hecaton's automatically-generated bowknots need manual alterations. As a result, Hecaton provides a confidence score for each bowknot. This score helps the analyst determine whether a manual fix is required, before spending any time on testing the bowknot. Our evaluations with real bugs show the confidence score correctly predicts the completeness of the automatically generated bowknots in 90% of the cases.

We evaluate bowknots and Hecaton with 113 real bugs, CVEs, and automatically injected bugs in several kernel components including the IPC subsystem, networking stack, file system, and device drivers in different Android devices and x86 upstream Linux kernels. First, we show that bowknots are effective workarounds for bugs. More specifically, we show that bowknots can effectively mitigate 92.3% of real bugs and CVEs and 94.6% of injected bugs. Second, we show that bowknots manage to maintain the system functionality in 87.6% of these cases. Third, we show that Hecaton automatically generates complete bowknots for 64.6% of kernel bugs. For the rest, it only requires adding on average 3 statements and less than 2 hours of work by the analyst. Fourth, we evaluate the correctness of bowknots' undo capability with a manual case-by-case study on 10 randomly selected real bugs. We show that for 6 out of these 10 bugs, automatically generated bowknots completely undo the side effects of the buggy syscall. Fifth, we show the effectiveness of bowknots in improving the efficiency of kernel fuzzing by effectively eliminating repetitive reboots. Sixth, we empirically compare bowknots with a recent bug workaround solution, Talos [18]. Bowknots significantly outperform Talos for bug mitigation, for maintaining the system functionality, and for improving kernel fuzzing in the face of repetitive reboots. Finally, we also evaluate the performance overhead of bowknot on normal execution of kernel components. We show that bowknots' overhead is less than the baseline variations for TCP throughput and GPU framerate even if we instrument all their corresponding kernel functions with bowknots.

[2]Hecaton's source code is available at https://trusslab.github.io/hecaton/

2 Motivation

2.1 Unpatched Kernel Bugs

As mentioned, kernel bugs pose security, reliability, and usability problems. Unfortunately, even when discovered, these bugs do not get patched immediately and there is a noticeable delay from when a bug is reported until when a patch is available. One reason behind this delay is that bugs can be complex and fixing them requires time and effort. To demonstrate this, we studied the bugs found by Syzbot [12], an automated fuzzing system based on Syzkaller [13]. At the time of the study (November 2019), there were 1691 bugs that were fixed. Our analysis shows that these bugs took an average of 66 days to get fixed. Moreover, there were 503 bugs that were still open, for an average of 232 days.

Moreover, bugs in device drivers (which constitute 85% of the kernel bugs [44]) might take even longer as the bug needs to be reported to the developers of the driver. For example, bugs in several drivers of Android smartphones based on Qualcomm chipsets need to be fixed by Qualcomm. Qualcomm says, "the company hopes to patch disclosed flaws and vulnerabilities within 90 days" [7].

2.2 Problems with Unpatched Kernel Bugs

Security. The most important problem with unpatched kernel bugs is that they endanger the system's security. Bugs might be exploitable, allowing attackers to mount privilege escalation attacks. Given the high privileges of the kernel, a successful attack can be devastating for the victim's device.

Reliability and usability. Even if not exploitable, kernel bugs cause reliability and usability problems, e.g., due to a

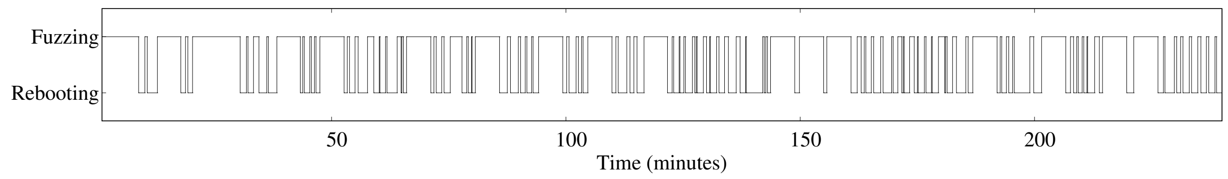

Figure 1: *Repetitive reboots when fuzzing the camera device driver of Nexus 5X.*

hang or reboot. Even worse, a bug might corrupt the state of the hardware and software, resulting in unexpected behavior. **Inefficient kernel fuzzing.** A lesser-known problem of unpatched kernel bugs is that they cause practical problems for fuzzing the kernel by causing repetitive reboots [38]. Kernel bugs, when triggered by the fuzzer, result in the reboot of the system. Unfortunately, reboots waste a noticeable amount of fuzzing time. The reboot itself takes 10s of seconds to minutes according to our own experience with various Android-based mobile devices and according to others [6]. In addition to wasting fuzzing time, a reboot resets the state of the system, throwing away the progress made by the fuzzer in mutating the state in order to find new bugs.

Unfortunately, modern feedback-driven fuzzers such as Syzkaller and AFL may trigger the same bug many times resulting in *repetitive reboots*, i.e., costly and useless reboots caused by the same bug, due to the feedback-driven fuzzing algorithm [5, 8] and some bugs being easy to trigger.

Figure 1 shows the timeline for one of these fuzzing sessions (i.e., fuzzing the camera device driver of Nexus 5X using Syzkaller). As can be seen, reboots happen very frequently, resulting in only 44.6% of the overall fuzzer uptime being spent on fuzzing (i.e., fuzzing time). The main reason for most reboots is triggering only 6 unique bugs again and again.

2.3 Current Approaches

Approach I: mitigation through code disabling. One possible approach is to try to mitigate a bug by disabling the part of the code that contains the bug. This can be done at different granularities. For example, the buggy subcomponent within the code can be disabled. If applied to the kernel, one can imagine disabling a device driver if it has a bug. It can also be applied at the function level. Talos uses this approach [18]. It neutralizes a vulnerability in a codebase by disabling the function that contains it. The function instead is instrumented to return an appropriate error message.

Although disabling the code can mitigate the bugs and vulnerabilities in many cases, it very likely breaks the system functionality. Losing functionality in a system will deter the use of this approach in practice. This approach does not help with the kernel fuzzing efficiency either. This is because code disabling limits the code coverage of the fuzzer (see §7.1.1 and §7.4 for empirical results).

Approach II: dirty patching. One might wonder whether the analyst can perform a "quick and dirty patch" to fix the bug. For example, if the bug is a null pointer dereference, they can add a null pointer check to return directly to avoid crashing. Unfortunately, dirty patching suffers from similar drawbacks as code disabling. That is, it can break the functionality of the system or result in unexpected behavior if not done carefully. In addition, such patches might still need engineering effort. For example, a dirty patch for a use-after-free bug resulting from a race condition is not trivial.

3 Overview

3.1 Goals

Our goal is to design a bug workaround solution that can mitigate the undesirable side effects of a bug until a proper patch is available. In other words, the applicability of the workaround is in the window of vulnerability from when the bug is first discovered until when the correct patch is available.

The main users of kernel bug workarounds are kernel security analysts, OS vendors, and IT departments. For example, the security team in an OS vendor company might find a bug and report it to the corresponding developers, e.g., another company in charge of a device driver or a development team within the same company. While they wait for the patch, they can use a workaround to mitigate the bug. Or an IT department might apply a workaround for a known bug in the company's servers or employees' workstations. Finally, security analysts can leverage this tool to mitigate kernel bugs in their own devices, e.g., to improve the efficiency of their kernel fuzzing sessions. To show our solution's applicability, we implement and test it on several targets, such as ARM-based Android smartphones and x86-based Linux kernels.

We identify five important properties that a bug workaround solution must satisfy. First, it should be fast to generate, otherwise it will not be available soon enough to help in the aforementioned window of vulnerability. Second, a workaround for a kernel bug should maintain the system's functionality even if the bug is triggered. Third, the workaround approach should be widely applicable to different kernel components and different kernels. Moreover, it should not require special hardware support, e.g., to checkpoint the state of an I/O device (§8). Fourth, a workaround should not add any noticeable performance overhead. Finally, a workaround should require

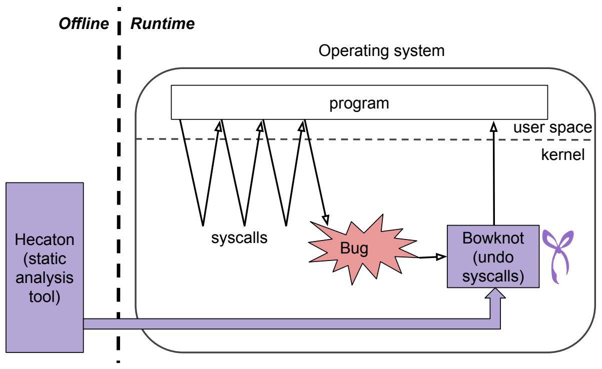

Figure 2: *High-level idea behind bowknots and Hecaton.*

small changes to the kernel, otherwise it will not be accepted by vendors for release in the window of vulnerability.

3.2 Key Idea & Design

Bowknots. In this paper, we introduce a workaround for kernel bugs called *Bug undO Workaround for KerNel sOlidiTy (bowknot)*. The key idea behind a bowknot is to undo the effects of the in-flight syscall that triggers a bug. That is, if a syscall is issued and triggers a bug, the bowknot generated for that bug undoes the syscall and returns, effectively neutralizing the syscall. It is important to note that a bowknot does not disallow a syscall, e.g., disallow all `ioctl` syscalls. It allows the syscall to be used as long as it does not trigger the bug. Only when an invocation of the syscall results in the bug getting triggered (e.g., due to using unexpected inputs), the bowknot kicks in to undo it so that the system can continue its execution and serve other well-structured syscalls.

Bowknots protect the kernel from corruption, which is critical for continued use of the system. They, however, can impact the program issuing the syscall. For example, they might result in the program breaking or terminating with an error message. We believe this is acceptable for three reasons. First, we do not anticipate most kernel bugs to be triggered by well-behaved applications. Many kernel bugs are only triggered when a meticulously-crafted syscall is issued, typically by malware. Second, applications can be restarted, if corrupted. Finally, kernel bugs that unconditionally break the usability of well-behaved applications are rare. This is because the kernel is tested for basic functionality by kernel developers.

Hecaton. Bowknots, as described so far, satisfy all but one of the aforementioned properties. More specifically, generating them manually requires noticeable engineering effort as one needs to study the execution path that triggers the bug and figure out how to undo the syscall. Therefore, to satisfy this last property, we introduce Hecaton, a static analysis tool that generates bowknots and inserts them into the kernel automatically. To do so, Hecaton leverages existing *undo statements* found within *error handling blocks* in the kernel to generate the right undo blocks for the functions in the execution path of the bug. Existing error handling blocks in the kernel undo the effects of a syscall on the software and hardware state in case of *expected* errors, such as a null pointer or a busy I/O error in some fixed code locations. While the kernel does not have error handling code for arbitrary bug sites in the execution of a syscall, the idea in Hecaton is to leverage existing undo statements in these blocks to generate the right undo code needed for a bowknot. More specifically, Hecaton leverages existing error handling blocks to discover undo statements for each state-mutating statement. Using such knowledge, Hecaton can then automatically generate the required bowknot for different functions. Figure 2 shows the high-level idea behind bowknots and Hecaton.

3.3 Workflow

Assume that the OS analyst has identified a bug in the kernel and would like to apply a bowknot to it. They take the following steps to achieve this.

In the first step, they need to identify the functions in the execution path from the beginning of the syscall until where the bug is triggered, i.e., the call stack. The call stack must include the inline functions since it will be used by Hecaton, which operates at the source code level. Bugs found by Syzkaller, such as the reported bugs in the Syzbot system [12], come with enhanced call traces, including all the inline functions and their location in the source code. For other bugs, the analyst can use any tool to find the stack. However, finding the inline functions in the stack might not be trivial. To make this step easy for the analyst, we provide support in Hecaton. That is, Hecaton instruments all the functions in the kernel component under study with some logging messages. The analyst then executes the Proof-of-Concept (PoC) program of the bug, checks the kernel logs, and extracts the list of functions executed in the syscall. They then feed this list back into Hecaton, which uses it to generate a copy of the kernel where only these functions are instrumented with bowknots. Hecaton provides a confidence score for each bowknot. If all the confidence scores for the instrumented functions are higher than a predefined threshold, the analyst goes to the next step to test the instrumented kernel. Otherwise, they can decide to investigate the bowknots with low confidence score and manually correct them, or altogether drop working on these bowknots if they are unwilling to spend time and manual effort to fix the bowknots.

The analyst then tests the instrumented kernel using the PoC and test programs. The purpose of test programs is to demonstrate proper functionality of the system after undo by bowknots. More specifically, the analyst first runs the PoC to verify that it does not succeed, e.g., it does not crash the kernel. They then run the tests to verify that the kernel component under test is still functional. If either fails, the analyst checks the generated bowknots. The analyst spends a few hours (e.g., up to 2 hours in our evaluation) to identify the problem, e.g., a missing undo statement. In fact, some of the bowknots might have explicit warnings from Hecaton (§5.2), which makes the manual step more straightforward. After a fix, they run the tests again. If the analyst does not find a fix in this period (e.g., the two hours), they declare the use of bowknots ineffective.

It is noteworthy that the analyst does not even need a fully functional PoC to test the bowknots. A program that results in the execution of the same functions but does not even trigger the bug suffices. We have indeed used this in our own evaluations. We tested a reported PoC that reached the bug but did not trigger it. Yet, by adding an explicit crash just before the bug site, we emulated the behavior and tested the undo behavior by the bowknot.

Finally, we note some bugs might be triggered through more than one call stacks. While such bugs are not common, to mitigate them, the analyst needs to generate bowknots for each call stack separately.

4 Bowknots

Bowknots are workarounds for kernel bugs. The key idea behind bowknots is to undo the side effects of the syscall that triggers the bug. More specifically, bowknots undo the side effects of state-mutating statements from the syscall's kernel entry point until where the bug is triggered. We define a state-mutating statement as one that alters the state of the kernel or an underlying I/O device.

For example, imagine a camera device driver `ioctl` syscall, which when called, allocates a memory buffer using `kmalloc()`, acquires a spin lock (`spin_lock()`), and turns on the flash for the camera (using the hypothetical function `turn_on_flash()`). Now imagine there exists a bug after this where a pointer might be null depending on the syscall input. To mitigate this bug, the analyst can apply a bowknot. It first turns off the camera flash (by calling `turn_off_flash()`), unlocks the spin lock (by calling `spin_unlock()`), and frees the allocated memory buffer (by calling `kfree()`). As can be seen, the state of the system (including the kernel memory state as well as the I/O hardware state, e.g., the camera hardware state) after undo is the same as the state before issuing the syscall. Therefore, the system can now resume its execution as if the syscall did not happen.

Our strategy for undoing a syscall is to leverage existing undo statements in error handling code in the kernel to generate the proper undo code that undoes the effects of all state-mutating statements in the syscall. Existing error handling code in the kernel undoes the effect of these statements when facing an expected error. The insight behind this approach is that OS kernels have to be robust and handle various corner cases or errors. Therefore, we attempt to reuse the existing undo statements to generate the right undo code for a bug location. In this section, we show how a bowknot can be used for a bug. In the next section, we discuss how Hecaton helps to automatically generate the undo code for bowknots.

4.1 Function Instrumentation

The goal of function instrumentation for a bowknot is to undo the executed statements in a function when a bug is triggered. We support two types of bowknots for a function: automatically-triggered and manually-triggered. Automatically-triggered bowknots are the common ones and are used for crash bugs and bugs automatically detected by a kernel sanitizer. The manually-triggered ones are for more complex bugs, such as race conditions and memory leaks.

Automatically-triggered bowknots. Figure 3 (Up) shows an instance of an automatically-triggered bowknot for a function in Qualcomm's KGSL GPU driver. This function is the handler for one of the supported `ioctl` syscall commands for this driver and is called by the main `ioctl` handler, `kgsl_ioctl`. The function instrumentation has several parts. The first part is an undo block at the end of a function, which contains all the undo statements corresponding to the state-mutating statements in the function. There are two state-mutating statements in this function: `kgsl_context_get_owner()`, which returns a `context` object while incrementing its reference counter, and `mutex_lock()`, which acquires a lock. The corresponding statements to undo the effects of these statements in the function are, respectively, `kgsl_context_put()` and `mutex_unlock()`. This undo basic block is also protected by an always-false global variable (`bowknot_global_always_false`) preventing it from being used in the normal execution of the function. It is only accessible through explicit jumps to `bowknot_label`.

The second part of the instrumentation is for detecting, at runtime, the state-mutating statements that are executed before the crash. This is because not all execution paths within a function execute the same set of state-mutating statements. If not taken into account, in the case of a specific bug, an unnecessary undo statement might get executed. Therefore, we instrument the function to keep track of the execution of the state-mutating statements. To do this, we use a per-function mask variable. We add the mask update statements after each state-mutating and undo statement. We also make the undo statements in the undo block conditional based on the bits in this mask. In our example, after a call to `mutex_lock()`, we set a bit in the mask variable. After a call to the corresponding `mutex_unlock()`, we reset the same bit in the mask variable. Then in the undo block of the bowknot, we check the bit. If set, we execute the `mutex_unlock()` statement.

The third part of the instrumentation, which is used for automatically-triggered bowknots, is the automatic redirection of the execution to the undo block when a bug is triggered. To do this, we add conditional `goto` statements (`CGOTO`) after all statements. The goal of these statements is to redirect the execution to the undo block in case of a bug. When a crash happens or a bug is detected by the kernel sanitizer, the execution is redirected to the kernel exception handler, which

```
1  #define CGOTO if(unlikely(current->bowknot_flag))
2                    goto bowknot_label
3
4  long kgsl_ioctl_device_waittimestamp_ctxtid(
5       struct kgsl_device_private *dev_priv, unsigned int cmd,
6       void *data)
7  {
8    uint64_t bowknot_pairmask = 0;
9
10   struct kgsl_device_waittimestamp_ctxtid *param = data; CGOTO;
11   struct kgsl_device *device = dev_priv->device; CGOTO;
12   long result = -EINVAL; CGOTO;
13   struct kgsl_context *context; CGOTO;
14
15   mutex_lock(&device->mutex); CGOTO;
16   bowknot_set_bit(bowknot_pairmask, 2);
17
18   context =
19     kgsl_context_get_owner(dev_priv, param->context_id); CGOTO;
20   bowknot_set_bit(bowknot_pairmask, 1);
21
22   if (context == NULL) {
23      goto out;
24   }
25   ...
26 out:
27   kgsl_context_put(context); CGOTO;
28   bowknot_unset_bit(bowknot_pairmask, 1);
29   mutex_unlock(&device->mutex); CGOTO;
30   bowknot_unset_bit(bowknot_pairmask, 2);
31   return result;
32
33   if (bowknot_global_always_false < 0) {
34 bowknot_label:
35     current->bowknot_flag = 0;
36     if(bowknot_check_bit(bowknot_pairmask, 2))
37      mutex_unlock(&device->mutex);
38     if(bowknot_check_bit(bowknot_pairmask, 1))
39      kgsl_context_put(context);
40     current->bowknot_flag = 1;
41     return -1;
42   }
43 }
44
45 long kgsl_ioctl(struct file *filep,
46                unsigned int cmd, unsigned long arg)
47 {
48   ...
49   ret = kgsl_ioctl_device_waittimestamp_ctxid(...); CGOTO;
50   ...
51   if (bowknot_global_always_false < 0) {
52 bowknot_label:
53   ...
54     return -1;
55   }
56 }
```

```
     ...
15   mutex_lock(&device->mutex); CGOTO;
16   bowknot_set_bit(bowknot_pairmask, 2);
17
   if(unlikely(param == unexpected_ctx))
      goto bowknot_label;
18   context =
19     kgsl_context_get_owner(dev_priv, param->context_id); CGOTO;
     ...
```

Figure 3: *Example function in the Qualcomm KGSL GPU device driver after instrumentation with a bowknot. (Up) Automatically-triggered, (Down) Manually-triggered bowknot. The blue and bold text highlights the automatically added code. The green and italic text highlights the manually added lines. The code presented here is slightly modified from the actual function code and from the one generated by Hecaton for better readability.*

we instrument. Our exception handler code sets the redirection flag (`bowknot_flag`), which is a thread-specific flag, and then returns the execution back to the function resulting in a jump to the undo block. In the previous example, assume that `param` is null and results in a crash at line 19. The exception handler is then invoked, sets the flag, and resumes the execution in the function (by skipping the crashing instruction), which then executes the conditional `goto` statement in the same line and jumps to the undo block. This condition is typically false during normal execution in the kernel. Hence, we use the compiler's `unlikely` directive, which helps with performance in normal execution by instructing the compiler to insert some instructions in the binary to assist CPU's branch prediction.

We also support automatic redirection for bugs detected by a kernel sanitizer (if activated, e.g., during a fuzzing session). In this case, we force-execute the kernel exception handler for bugs detected by the sanitizers, e.g., memory safety bugs detected by KASAN [11].

Note that automatically-triggered bowknots only get triggered on system crashes and warnings generated by kernel sanitizers. As a result, for non-crashing bugs that can potentially result in kernel corruption, the security of automatically-triggered bowknots depends on the appropriate use of kernel sanitizers (e.g., KASAN and KMSAN) to catch the bug before the corruption happens. Although currently sanitizers are enabled only during testing due to their memory and performance overhead, there are recent efforts to enable efficient sanitizers to be used in deployed products as well [40] [25].

Manually-triggered bowknots. There are two important scenarios when manually-triggered bowknots are desired or needed. First, some bugs do not result in a crash nor are detected by a kernel sanitizer. However, the security analyst knows the condition under which the bug is triggered. In this case, the analyst can add an explicit condition to the function containing the bug to redirect the execution to the undo block before the bug is triggered. Figure 3 (Down) shows an example. In this (hypothetical) case, if the `param` parameter is equal to a known global object, the behavior is buggy resulting in the corruption of the object. Therefore, the analyst can add the conditional block between lines 17 and 18 to jump to bowknot's undo block. The analyst does not need to generate the bowknot nor figure out which undo statements need to be called. She only needs to determine where and under what conditions the bowknot needs to be executed.

Second, in some production systems, instrumenting the kernel exception handler or deploying a kernel sanitizer (as needed for automatically-triggered bowknots) might not be acceptable. In such cases, manually-triggered bowknots can be used, even for simple bugs such as crash bugs.

4.2 Recursive Undo of Call Stack

When a bug is triggered, bowknot executes the undo code for the function the bug is in. It then needs to undo the effects of the statements in the parent functions.

To do this, we undo the parent functions similar to the buggy function. Figure 3 shows the parent function as well. We perform the recursive undo through the use of the thread-specific flag mentioned earlier (`current->bowknot_flag`). When returning from the buggy function, this flag is set. Moreover, the parent function is also instrumented with the conditional `goto` statements. Therefore, after returning from the buggy function, the parent function jumps to its own bowknot and executes its own undo code. This recursive undo continues until the syscall returns, at which point the flag is cleared.

It is important to note that the bowknots in the parent functions are always automatically-triggered. Only the last function in the stack might need manual triggering of the bowknot.

Also, note that it is feasible to rely on the existing error handling blocks in some functions rather than using bowknots. We use this approach for the first few functions in the execution paths of a syscall, which receive a syscall and route them to an underlying component to handle. As a practical guideline, when dealing with a bug in a specific kernel component, e.g., a device driver, we only apply bowknots to the functions in the path within the driver. When recursively undoing the functions, the entry function in the kernel component simply returns an error, which is elegantly handled by existing kernel code by routing the error to the user space. We take this approach for two reasons. First, the functions parsing and routing a syscall are triggered for every syscall and hence have impact on the system's performance. Second, these functions are mature and have adequate error handling code, eliminating the need to inject custom undo code for them.

5 Automatic Generation of Bowknots

In this section, we describe how Hecaton generates the undo block of the bowknot automatically. Hecaton also automatically instruments the designated kernel functions, which we do not discuss further here.

We build Hecaton as a static analysis tool. It generates the undo block by analyzing the entire kernel to infer the relationship between state-mutating statements and their corresponding error handling undo statements. Hecaton achieves so in two main steps: (*i*) generating a kernel-wide knowledge database of function pairs and (*ii*) generating the undo block using the database as well as function-level analysis. We next describe these two steps.

5.1 Function-Pair Knowledge Database

The goal of the function-pair knowledge database is to store pairs of functions that mutate and undo the kernel state. In other words, a state-mutating function and an undo function are paired, if the latter undoes the effect of the former. (`kmalloc, kfree`), (`mutex_lock, mutex_unlock`), and (`msm_camera_power_down, msm_camera_power_up`) are a few examples of such function pairs. The function-pair knowledge database can be reused across various kernels, e.g., the kernels of different Android devices, with minimal changes. Therefore, our general approach is to automatically extract function pair candidates, manually inspect them, and add them to the database if verified. This approach provides high confidence in the database. Moreover, since generating the database is mostly a one-time effort, the manual effort is not significant. (We provide some quantification of the manual effort later in this section and in §7.2).

Identifying function pair candidates. Hecaton statically analyzes the entire kernel to identify function pair candidates. It uses two methods to identify the candidates. First, it uses the function names. In this method, Hecaton considers a function pair as a candidate, if the names of two functions only differ in one word and the difference is one of the following: (`put, get`), (`put, create`), (`release, get`), (`release, create`), (`remove, create`), (`deinit, init`), (`unregister, register`), (`unlock, lock`), (`down, up`), (`disable, enable`), (`sub, add`), (`dec, inc`), (`unset, set`), (`clear, set`), (`free, alloc`), (`stop, start`), (`suspend, resume`), (`disconnect, connect`), (`unmap, map`), (`dequeue, enqueue`), (`unprepare, prepare`), and (`detach, attach`). Using this method, for example, Hecaton found 540 pairs of function in the Linux kernel used in the Pixel3 smartphone.

Unfortunately, not all function pairs differ in one word only. As a result, Hecaton employs a second method, in which it uses existing error handling blocks in the kernel to identify undo functions and then match them to candidate state-mutating functions in the same function using string matching. More specifically, Hecaton marks all the functions in error handling blocks as undo functions. Then, for each undo function, it matches it with a candidate state-mutating function in the same function using similarity in their names and input/output variables. For the similarity score, Hecaton calculates the sum of the lengths of all mutually-exclusive substrings. To do so, Hecaton finds the longest common substring (LCS) and adds its length to the similarity score. Then it deletes the LCS from both strings and repeats the previous steps recursively until there is no common substring with more than two characters.

Towards this goal, Hecaton needs to be able to identify error handling blocks in the kernel. Hecaton does so by looking for common conditional statements used to identify and handle an error in the kernel. By investigating a large amount of kernel code, we have identified four such conditional blocks including (*i*) `if (rc < 0) {...}` where `rc` is an integer, (*ii*) `if (IS_ERR(p)) {...}` or `if (p == NULL) {...}`, where `p` is a pointer, (*iii*) `if (...) {...; return ERROR;}` where `ERROR` is a constant negative integer, often one of the commonly used error numbers in the kernel such as `-ENOMEM` and `-EFAULT`, and (*iv*) `if (...) {...; goto LABEL;}`. It also considers simple variations of these four categories such as checking within the `else` block rather than the `then` block for categories (*iii*) and (*iv*).

Once it identifies the error handling blocks, Hecaton needs to match the undo functions in them with state-mutating functions. That is, it assumes that every undo function call statement undoes the effects of a single state-mutating function call in the same parent function. For example, `kfree()` is an undo function statement that corresponds to the state-mutating function statement `kmalloc()`. Hecaton uses the same heuristic string matching discussed above to identify the candidates. For example, `kgsl_context_put(context)` is paired with `context = kgsl_context_get_owner(...)`. To do this, Hecaton calculates the string-based similarity score between the undo statement and all statements prior to the corresponding error handling block. It then chooses the function with the highest similarity score. Using this method, for example, we identified 1158 candidate pairs in the Pixel3 kernel (excluding the pairs found using the previous method).

Manual inspection of function pair candidates. Not all function pair candidates are true pairs of state-mutating and undo ones. This is because the method discussed above, i.e., string matching, is not precise. Therefore, we perform manual inspection on the candidates to identify the true pairs. In this step, we use our knowledge of kernel code. In addition, we use the frequency of appearances of a function pair candidate as a hint to facilitate the manual inspection. Pairs that appear many times together in many functions are less likely to be false pairs. Using manual inspection, in the case of the Pixel3 kernel, we verified all 540 pairs identified using the first method and 658 of the function pairs identified using the second one, bringing the total number of function pairs in the database to 1198. This manual inspection took one of the authors 7 days to complete. However, as mentioned, this is largely a one-time effort. Supporting a new version of the kernel or a new device driver adds a small number of new candidate pairs, which can be verified fast. As an example, once we had the database for the Pixel3 kernel, we ran our static analysis tool on a Nexus 5X driver that we needed to test. Doing so resulted only in 9 new candidate pairs, which we quickly inspected. We evaluate the amount of manual effort for x86 kernels in §7.2.

5.2 Generating the Undo Block

To generate the bowknot's undo block, we need to identify all the state-mutating statements in the function, and generate the corresponding block. Hecaton is not currently able to generate an undo statement, as it might require fixing the parameters passed to a function. Therefore, Hecaton tries to *reuse existing undo statements* in a function and match them with the state-mutating ones. If Hecaton does not find a match for an undo statement in a function, or if it does not find a match for a state-mutating one, it inserts a warning in the undo blocks that it generates so that the analyst can manually fix the problem. Simply reusing existing statements is adequate in a large number of functions (§7.1.1).

As mentioned, Hecaton attempts to find all undo statements in the function for which it generates the undo block. An undo statement might be a function call or not. Hecaton uses the knowledge database to identify all the undo function call statements. For other undo statements, e.g., a counter decrement, it relies on the error handling blocks in the function.

To identify the error handling block candidates, we use the patterns often used for these blocks as discussed earlier. In addition, we also inspect all blocks that have one of the following jump statements in their bodies: `break`, `continue`, `return`, and `goto`. If such a block contains an undo function call (determined by consulting our knowledge database), we mark that block as an error handling one as well. In addition to the error handling blocks, some functions incorporate undo statements prior to the return statement. For example, it is common in kernel functions to allocate, acquire, enable, or turn on a resource, perform a task on it and then free, release, disable, or turn off that resource before returning a success value. Hecaton reuses these undo statements as well.

Having all the undo statements, the next step is to find their corresponding state-mutating statements. For error handling statements that are function calls, Hecaton uses its knowledge database. If there are multiple instances of the same state-mutating function, Hecaton chooses the one that shares more variables with the error handling statement. For all other types of statements, Hecaton uses string matching to pair them with state-mutating statements.

5.3 Incompleteness and Confidence Score

As mentioned, a small portion of bowknots generated automatically by Hecaton are not complete and require manual amendments. We analyze the underlying reasons for this incompleteness through experiments and a case-by-case study. We enable Hecaton to automatically detect features in functions that may result in the generation of an incomplete bowknot. For each generated bowknot, Hecaton provides a confidence score, indicating the probability of its effectiveness. Also, in cases that manual effort is necessary, Hecaton highlights the function(s) in the call stack that have the most negative effect on the confidence score and need manual corrections. Our experience and analysis show that six features play critical roles in generating complete bowknots. We quantify these features and linearly combine them into a single confidence score using adjustable coefficients. Finally, we tune these adjustable coefficients using real bugs (§7.1.3).

The first feature we use is the **location of the bug**. Our experience shows that if the last function of the call stack of the bug is inside a kernel component (e.g., a device driver), it is more likely that Hecaton could generate a complete working bowknot. In cases that the bug is in core kernel, for example, inside an inline function that manipulates kernel objects, it is less likely that Hecaton could generate complete bowknots.

The second feature is the presence of **missing undo statements**. As we discuss in §5.2, Hecaton currently cannot generate undo statements from scratch. We decrease the confidence score when Hecaton does not find an undo match for a state-mutating function found in its knowledge database.

The third feature is the **method of error handling block detection** used in a function. As we discuss in §5.2 and §5.1, Hecaton uses different patterns to identify error handling blocks. Some of these patterns are used both in error handling and non-error handling blocks and hence might produce false undo statements. Therefore, we decrease the confidence score if such patterns are used.

The fourth feature is the **presence of function pointers**. As Hecaton currently cannot pair the state-mutating function pointers with its correct undo statement using its knowledge database, it solely relies on the string matching heuristic to pair them. As a result, we decrease the confidence score in the presence of such statements.

The fifth feature is the **presence of multi-statement undo code**, where multiple statements are used to undo one or more state-mutating statements. One important example is when a loop is used to undo the effects of another loop. Another important example is when a critical section is used in the error handling block. Hecaton assumes a one-to-one mapping between state-mutating and undo statements, and hence does not currently automatically handle such cases.

Finally, to take the miscellaneous unknown sources of inaccuracy in Hecaton's static analysis into account, we decrease the confidence score as the **number of state-mutating statements** in a function increases since having more state-mutating statements to pair increases the error probability.

6 Implementation

Static analysis tool. We implement Hecaton in C++ and Python with about 4,550 LoC. We use Clang for static analysis in Hecaton as it allows us to perform the analysis at the source code level. While we mainly test our solutions with the Linux kernel of Android devices and upstream x86 Linux kernels, we note that they are applicable to other OSes as well. Our static analysis tool is implemented as a plug-in for the Clang compiler. We use our plug-in alongside Android Clang version 5.0.1 for our Android devices, and we use the same plug-in (with a small modification to make it compatible with the newer version of Clang) alongside Clang version 11.0.0 for our upstream x86 Linux kernels.

We perform our analysis on the Abstract Syntax Tree (AST). When using the AST, we do not need to worry about parsing and lexing the source code. Moreover, we have high-level information of the source code needed for our analysis, such as functions and variables names. In addition, the organized structure of the AST facilitates finding the error handling blocks. In AST, all the statements and expressions are organized in a hierarchical structure as nodes of a tree, and Clang provides many helper functions to traverse the AST in an efficient way. There are also many helper functions to obtain attributes of each node of the AST. To obtain the AST of the source code, we use `ASTFrontendAction` with a custom `ASTConsumer`. We override the `VisitFunctionDecl` function of our custom `ASTConsumer` to obtain all the function declaration nodes in the AST. All the statements in the body of each function appear as children nodes of the function declaration node. To perform our analysis, we recursively visit all the children nodes in several passes. In these passes, using AST, first, we identify and pair undo nodes and state-mutating nodes to generate a bowknot for each function. As discussed in §4.1, a bowknot includes a generic undo block, several conditional `goto` statements, and several mask update statements. Then, using the AST helper function, `getSourceRange`, we identify the locations of these nodes in the source files. Finally, using Clang's Rewriter tool, we directly inject the generated bowknot into the source code.

Exception handler. We have implemented Hecaton with automatically-triggered bowknots for two Android devices naming Pixel3 and Nexus 5X and various versions of three x86 kernel branches naming upstream Linux kernel, Google's KMSAN kernel, and Linux-Next kernel. Nexus 5X runs CyanogenMod-13 Android OS with Linux kernel 3.10.73, Pixel3 runs Android-9.0.0 r0.43 with Linux kernel 4.9.96, and the x86 Linux versions vary between 5.5.0 and 5.8.0.

As discussed in §4.1, to support automatically-triggered bowknots, we need to instrument the kernel's exception handler. First, we need to distinguish between bowknot-supported faults and normal faults. To achieve this goal, we statically disassemble and parse the kernel image and extract the address ranges of bowknot-supported functions and save them into a header file. When any exception occurs, we use this header file to execute our modified exception handler for bowknot-supported faults and execute the unmodified exception handler otherwise. In our modified exception handler, after setting `bowknot_flag`, before returning to the buggy function, we advance the Program Counter (PC) register to skip the crashing instruction. In ARM architecture, all instructions have the same length, and we simply advanced the PC register by four. However, x86 instructions have variable lengths. As a result, we need to decode the current instruction's length to advance the PC to the next instruction. We use Zydis for this purpose, which is a lightweight open-source disassembler library for x86 and x86-64 instructions implemented in C [4]. Since Zydis is implemented with no third-party dependency (not even libC), we can build Zydis as a part of the Linux kernel. To minimize code added to the kernel, we only port parts of the Zydis necessary to decode the instructions' length.

For ARM, we add 72 lines of C code and 42 lines of assembly code to the kernel exception handler. For x86, we add 136 lines of C code to the kernel exception handler and port 4677 lines of C code from the Zydis library.

7 Evaluation

7.1 Effectiveness

7.1.1 Effectiveness in Bug Mitigation

Methodology. To test the effectiveness of Hecaton and bowknots, we test our bug workaround against 113 bugs in Android and x86 Linux kernel consisting of real CVEs, unpatched real bugs, and injected bugs. Using a combination of real and synthesized bugs to evaluate the effectiveness of fault-tolerant systems is a common practice [18] [21]. However, previous similar work, Talos [18], only used 11 real-world vulnerabilities and FGFT [21] tested no real-world bugs. In contrast, we use 39 real-world bugs. Similar to Talos and FGFT, to evaluate the effectiveness of bowknots, we measure two factors for each bug. First, whether the bug is successfully mitigated, and second, whether the system including the buggy module remains functional after the undo.

In our experiments, we use PoCs to trigger the bugs. In a successful mitigation, we make sure that the PoC still triggers the bug after bowknots insertion but that the execution of bowknots neutralizes the syscall that triggers the bug in a way that prevents the system from crashing, freezing, or generating further warnings by kernel sanitizers.

In addition, we test the functionality of the buggy module after the execution of bowknots as a result of triggering each bug. For our functionality test, we use standard benchmarking and self-test programs when they are available for a kernel module (e.g., GPU benchmarking application or Linux self-tests for a file system). Self-tests are small test programs that kernel developers have designed to exercise individual code paths in the kernel and report whether or not they achieve the expected outcomes. If no standard benchmark or self-test is available for a module, we manually test the underlying device of the buggy device driver in different configurations (e.g., taking pictures and videos in different settings to make sure the camera is functional.)

For comparison, we also test and report mitigation and functionality preserving for each bug using Talos [18], which uses code disabling (§2.3). Since Talos disables parts of the code, it might seem unnecessary to test Talos workarounds for functionality. However, in some cases, the disabled function does not play a crucial role in the functionality of the device, for example, when the bug is located in a function that logs the device driver's events. In these cases, code disabling (Talos) might preserve the functionality of the device.

As we discuss in §9, bowknots cannot be used for the bugs located in the kernel's clean-up paths. Hence, we only measure and report (in §9) how common this limitation is, and we do not consider them in our effectiveness evaluations.

We also evaluate the effectiveness of Hecaton in generating complete bowknots. First, we report whether the bowknots get executed automatically or if we manually encode the condition for its execution. Second, we report whether the automatically generated bowknots are complete or if we manually add statements to complete them. For each bug, we limit the amount of manual effort to complete its bowknots to 2 hours. If we could not fix a bowknot manually in 2 hours, we record it as unsuccessful.

CVEs and Real Bugs in Android To evaluate the effectiveness of bowknots and Hecaton in mitigating real bugs and vulnerabilities of Android devices, we use 9 real bugs and reported CVEs in four kernel components of the Pixel3 smartphone: binder IPC, camera driver, GPU driver, and the TCP layer in the network stack (used with WiFi).

Table 1 shows the result. It shows that bowknots are effective in mitigating the bugs and vulnerabilities in 100% of cases and maintain the system functionality in 100% of these cases. 88.9 % of bowknots use automatic triggers and only one case uses manual triggers. Moreover, Hecaton is capable of generating complete bowknots in 55.6% of cases. In contrast, Talos can only mitigate the bugs in 66.7% of cases and preserve the functionality in 22.2% of these cases. We discuss five of these vulnerabilities in Appendix.

Unpatched Real Bugs in x86 Linux kernel To further evaluate the applicability of bowknots and Hecaton to different targets and unpatched bugs, we use 30 real bugs in x86 Linux kernels reported by Syzbot [12]. We choose the 30 latest unpatched bugs (as of July 2020), which have reproducer PoC programs. The 30 bugs we test are located in various parts of the Linux kernel such as network stack, file system, memory management, HCI Bluetooth driver, and TTY driver.

Table 2 shows the results. It shows that bowknots are effective in mitigating the bugs and vulnerabilities in 90% of cases and maintain the system functionality in 90% of these cases. Moreover, Hecaton is capable of generating complete bowknots in 60% of cases. In contrast, Talos can only mitigate the bugs in 66.7% of cases and preserve the functionality in 26.7% of these cases.

Injected Bugs in Android To further test the ability of bowknots in maintaining the system functionality, and test the robustness of Hecaton against the location of the bugs in the kernel functions, we use bug injection. More specifically, we inject 41 bugs in the camera driver of Pixel3 and 33 bugs in its binder IPC subsystem. To avoid any bias in favor of or against Hecaton, we randomly choose the bug injection location. To do so, first, we fuzz each module using Syzkaller to identify all lines of code reachable through the syscall interface. Next, after excluding the locations in the kernel's clean-up paths (see §9), we randomly choose one of the reachable lines and insert an explicit `BUG()` function there. Since the inserted `BUG()`'s location is random, an arbitrary number of state-mutating statements might get executed prior to the bug, which needs to be undone by a bowknot. As a result, this evaluates the ability of Hecaton in generating effective bowknots in various cases. We then generate bowknots using Hecaton and apply them for each bug. Table 3 shows the results. It shows that bowknots are effective in mitigating the

Kernel Modules	Bug/ Vulnerability	Talos Mitigate?	Talos Preserve Function?	Bowknot Mitigate?	Bowknot Preserve Function?	Bowknot Trigger Mode	Hecaton's Generated Bowknots
Binder IPC	CVE-2019-2215	✓	✗	✓	✓	Manual	Not-Complete
	CVE-2019-1999	✓	✗*	✓	✓	Automatic	Complete
	CVE-2019-2000	✗	✗	✓	✓	Automatic	Complete
Camera Driver	CVE-2019-2284	✗	✗	✓	✓	Automatic	Not-Complete
	bug: msm_camera_power_down	✗	✗	✓	✓	Automatic	Not-Complete
	CVE-2019-2293	✓	✗	✓	✓	Automatic	Not-Complete
GPU Driver	CVE-2019-10529	✓	✗*	✓	✓	Automatic	Complete
	CVE-2018-5831	✓	✓	✓	✓	Automatic	Complete
Network (TCP)	CVE-2019-18805	✓	✓	✓	✓	Automatic	Complete

Table 1: *CVEs and real kernel bugs tested with bowknots. (* In these cases, the system was functional right after mitigation by Talos, but it stopped working after a while due to a memory leak resulting from code disabling.)*

Total # of tested Bugs	# mitigated by Talos	# function preserved by Talos	# mitigated by bowknots	# function preserved by bowknots	# automatic bowknot trigger	# complete bowknots by Hecaton	Avg. # added undo statements for incomplete bowknots by Hecaton
30	20	8	27	27	30	18	2

Table 2: *Unpatched bugs experiments (x86 Linux kernel bugs reported by Syzbot).*

bugs in 94.6% of cases and maintain the system functionality in 85.1% of these cases. Moreover, Hecaton is capable of generating complete bowknots in 70.4% of cases. In contrast, Talos can only mitigate the bugs in 64.9% of cases and preserve the functionality in 23.9% of these cases.

For all bugs for which Hecaton's bowknots were incomplete (injected bugs as well as real bugs and vulnerabilities), we needed to add on average 3 statements.

7.1.2 Effectiveness of Syscall Undo

We perform a detailed case study to evaluate bowknots' syscall undo capability. We perform a manual line-by-line investigation on the execution path of 10 real bugs (5 Android kernel and 5 x86 Linux bugs randomly chosen from the bugs discussed in §7.1.1). In this investigation, we search for any statement that changes the global state of the system but is not undone by bowknots. The result of this analysis shows that, to the best of our knowledge, for 6 cases the undo was complete and there were no changes to the system global state that did not get undone by the bowknots. Additionally, in 3 of the 4 failed cases, we could manually add the undo statements for the missed state-mutating statements and complete the bowknot in less than 2 hours. In the remaining one case, the state gets corrupted in a way that we even could not generate a complete bowknot manually. We discuss this case-by-case analysis in detail in the Appendix.

7.1.3 Effectiveness of Confidence Score

To evaluate Hecaton's confidence score, we use our corpus of 30 unpatched real bugs in x86 Linux kernel, which we discussed in §7.1.1. As mentioned in §5.3, Hecaton generates a confidence score for each bowknot instrumented function. Even if only one bowknot fails to undo the side effects of a partially executed function, the system state might remain inconsistent. As a result, to evaluate each bug, we consider the minimum confidence score for the bowknot instrumented functions in its call stack. We divide these 30 bugs into two sets of 20 and 10 bugs for respectively tuning and testing our confidence score. We tune the six coefficients of the confidence score (§5.3) in a way that it best separates the tuning set of bugs into two groups, one with complete bowknots and one that needs manual effort. Then we measure how well the tuned confidence score can predict the completeness of the bowknots Hecaton generates for 10 bugs in the testing set. Note that a false negative prediction is more acceptable than a false positive because in the case of a false negative the confidence score predicts an incomplete bowknot, which ends up being complete. Figure 4 shows that the confidence score works for 95% of the cases in the tuning set, and it predicts the completeness of generated bowknot with 90% accuracy in the testing set. Please note that there is no false positive in the results. In other words, whenever the minimum confidence score is greater than 50, the bowknots are complete.

7.2 Manual Effort for the Pair Database

We measure how much manual effort is needed to keep Hecaton's function-pair knowledge database updated with the ongoing updates in the kernel. For this purpose, we use Hecaton to generate the databases for 9 consecutive versions of x86 upstream Linux kernel, i.e., v5.0 to v5.8. As we discuss in §5.1, this database needs to be manually inspected and veri-

Kernel Modules	# Injected Bugs	# mitigated by Talos	# function preserved by Talos	# mitigated by bowknots	# function preserved by bowknots	# automatic bowknot trigger	# complete bowknots by Hecaton	Avg. # added undo statements for incomplete bowknots by Hecaton
Camera	41	34	5	40	33	33	26	2
Binder	33	14	12	30	30	26	24	4

Table 3: *Bug injection experiments (camera device driver and Binder IPC).*

device	driver	version	bugs	U. reboots	U. up time	U. fuzz time	B. reboots	B. up time	B. fuzz time
Pixel3	Camera	2018-08-22	3	1035 ± 60	24h	$12h18m \pm 9m$	98.3 ± 114	24h	$22h49m \pm 1h5m$
Nexus 5X	Camera	2016-10-13	6	622.3 ± 48	24h	$12h10m \pm 19m$	12.0 ± 0.0	24h	$23h19m \pm 1m$

Table 4: *Effective fuzzing time. U. and B. refer to using unmodified kernel vs. a kernel updated with bowknots. The number of reboots are per hour. Up time is the overall time during which the fuzzer is running including wasted reboot time. Fuzz time (i.e., effective fuzz time) is the time during which the fuzzer is actually fuzzing the kernel of the device.*

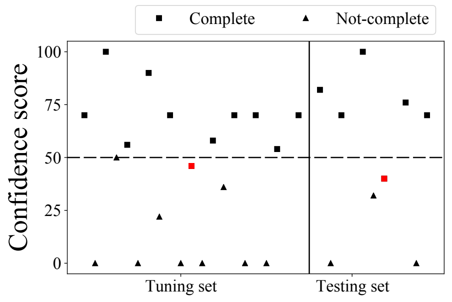

Figure 4: *Hecaton Confidence score prediction for Tuning and Testing sets*

fied. Our measurements show that when we move from one kernel version to the next, on average 115 ± 18 additional function pairs need to be verified, which in our experience takes between 2 to 3 hours.

7.3 Performance Overhead

We measure the overhead of bowknots on the normal performance of the system. To do so, we measure how the performance overhead increases as the number of executed functions with bowknot instrumentation increases. To test the performance overhead of bowknots in our ARM implementation, we use two benchmark applications, "GPU Mark benchmark" that measures the output frame-rate of GPU renderings, and "Tamosoft Throughput Test" that measures the downlink TCP throughput. To test the performance overhead of bowknots in our x86 implementation, we use iPerf tool [1] in Linux kernel to measure the downlink TCP throughput.

Each benchmark results in the execution of many functions in their corresponding kernel components. First, we detect all these triggered functions (410 functions in the Pixel3 GPU driver, 390 functions in the Pixel3 networking stack, and 370 functions in x86 Linux networking stack). We then randomly choose a number of these functions and instrument them with bowknots. For all modules, we either instrument 100, 200, or all available functions in them. We run the benchmarks 10

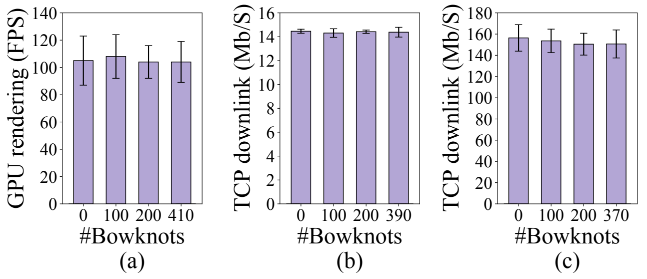

Figure 5: *GPU and TCP performance as the number of executed bowknots increase. (a) Pixel3 GPU , (b) Pixel3 TCP, (c) x86 upstream Linux (running in QEMU) TCP.*

times and show the average±stdev throughput in Figure 5.

The results show that there are no statistically noticeable performance drops even if all executed functions are instrumented with bowknots.

7.4 Use-Case Evaluation

As discussed in §2.2, by neutralizing bug-triggering syscalls, bowknots can help reduce the number of repetitive reboots during a fuzzing session. We evaluate the benefits of bowknots for fuzzing in this section. We fuzzed 13 device drivers and kernel components (camera driver, GPU driver, audio driver, WiFi driver, ION, Binder, and Ashmem) in three smartphones (Pixel3, Nexus 5X, and Samsung S7). Out of these, 5 of them showed repetitive reboots due to easily-triggered bugs. Out of these 5 drivers, 2 of them had easily-triggered bugs that bowknots could effectively mitigate. We show the results for these two drivers: the camera device driver of Pixel3 and the camera device driver of Nexus 5X. We note that bowknots cannot provide any benefits for the other three drivers.

We use the following experimental methodology. We run each fuzzing experiment for 24 hours as suggested by Klees et al. [23]. Moreover, we repeat each experiment 3 times and report averages and standard deviations. To implement this methodology, we faced and solved a practical challenge. More specifically, running 24-hour kernel fuzzing experiments on

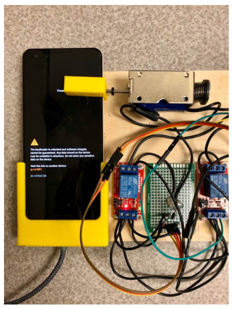

Figure 6: *The setup used in our fuzzing experiments.*

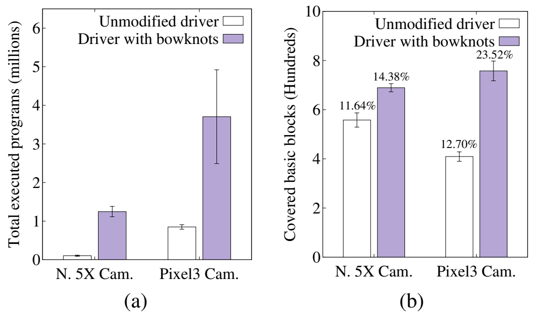

Figure 7: *(a) Total executed fuzzing programs. (b) Covered basic blocks (code coverage percentage is also reported on top of each bar).*

smartphones proved to be challenging due to unreliability of the Android Debug Bridge (ADB). Occasionally, ADB would malfunction and the desktop machine running the fuzzer would lose its connection to the device, disrupting the experiment. This phenomenon happened more frequently when the device was rebooted more often. Our first attempt to address this problem was to restart the experiment from scratch when this issue happened. Given that experiments are 24 hours long, this proved to be a very lengthy process. Therefore, we built a custom hardware-software framework to programmatically and forcefully reboot the device using its power button when the connection to the device was lost. Figure 6 shows this setup. We 3D printed the cover to hold the smartphone in place, used a 45 Newton linear solenoid to press and hold the power button, and used an Arduino Uno board to control the solenoid from the fuzzer.

Increased fuzzing time. Table 4 shows the effective fuzzing time achieved when fuzzing the unmodified driver and the driver with bowknots. As the table shows, bowknots increase the effective fuzzing time by $88.6\% \pm 4.6\%$.

Executed programs. Figure 7a shows the total number of executed fuzzing programs. Bowknots eliminate wasted fuzz time and hence the fuzzer executes more programs. Our results show that we manage to execute $723.5\% \pm 124\%$ more fuzzing programs on average with bowknots.

Code coverage. Figure 7b shows the code coverage in the driver under test. As can be seen, the higher number of executed programs and fewer reboots result in $54.3\% \pm 6.1\%$ higher code coverage.

Comparison with Talos. We compare the effectiveness of our approach in improving the fuzzing efficiency with Talos. To do this, we apply Talos to buggy functions in our fuzzing experiments. Our analysis shows that Talos, as a result, disables a large number of basic blocks, effectively lowering the code coverage. Moreover, our analysis shows that bowknots, when applied to the kernel, allow the fuzzer to cover a large part of the basic blocks that Talos disables. Table 5 shows the results. The results are insightful. Talos' approach disables the code unconditionally resulting in disabling 1290 basic blocks overall. However, bowknots only undo the syscall when they are triggered. Therefore, they allow the code to be executed with good inputs, i.e., those that do not result in triggering the bug. This proves to be critical for achieving good code coverage when fuzzing. As a result, bowknots help cover 66% of the basic blocks disabled by Talos.

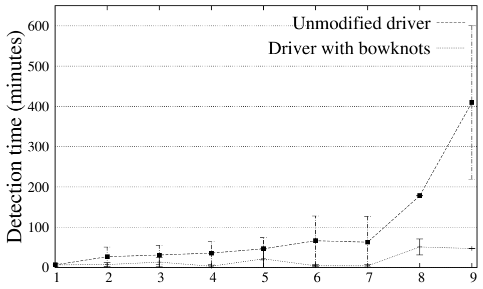

Figure 8: *Time taken for the fuzzer to discover a bug (i.e., trigger a bug for the first time). Each x-axis tick represents a unique bug.The points with no error bars represent bugs only found once during experiments*

Bug triggered by fuzzer	Basic blocks disabled by Talos	Basic blocks disabled by Talos & covered by bowknots
msm_actuator_subdev_ioctl	141	129
msm_camera_io_w_mb	2	2
msm_camera_io_r	2	2
msm_flash_config	91	82
msm_csid_config	37	35
msm_cpp_subdev_ioctl	785	459
cam_ife_mgr_acuire_hw	71	45
cam_sensor_core_power_up	109	67
msm_camera_power_down	52	32

Table 5: *Bowknots vs. code disabling (Talos) for fuzzing.*

Faster and more effective bug detection. By eliminating reboots with bowknots, we manage to find bugs faster. Figure 8 shows the list of all the bugs found in the two drivers. It shows on average the time it takes to find the bug in drivers with and without bowknots. As the results show, bowknots help us find all these bugs faster. On average, we find the same bugs faster by 42.6 minutes. This speed-up varies between 6 minutes to 162 minutes for different bugs.

8 Other Related Work

Automatic fault recovery. FGFT provides fine-grained recovery for faults in device drivers [21]. To do so, it checkpoints the memory and I/O device state on select entry points and restores them when a fault is detected. FGFT's key technique is to checkpoint and restore device state using existing power management code in device drivers. There are two important limitations that make this solution unsuitable to be used as a generic kernel bug workaround solution. First, checkpointing the state of an I/O device using power management facilities is not feasible for all I/O devices. In fact, some of the devices that we tested in our evaluation (e.g., the camera of Nexus 5X smartphone) do not support this. Moreover, a checkpointing solution for the kernel memory is difficult to integrate into existing kernels. Virtual machine checkpointing exists; however, that does not apply to the kernels of real systems. Second, checkpointing the state of the system before every syscall is costly.

ASSURE uses rescue points for automatic recovery from faults in an application [39]. Rescue points are sites within an application that handle known errors. When faced with an unknown error, ASSURE restores the state of the application to a suitable and close rescue point, which then returns an error. However, ASSURE requires checkpointing the state at rescue points, which is expensive for syscalls and not feasible for all the hardware state.

Akeso uses recovery domains to undo a syscall or interrupt upon a fault [24]. Recovery domains log modifications to the kernel state and commit only upon successful execution. This allows the domains to undo the effects when facing a fault. Similar to Hecaton, Akeso can undo a syscall that ends up in a bug trigger. However, Akeso's approach is not suitable for a bug workaround either. First, Akeso has significant performance overhead ($1.08\times$ to $5.6\times$). Second, Akeso does not support "code that write directly to external devices", which includes important parts of device drivers.

RCV automatically recovers from null pointer dereference and divide-by-zero errors [27]. It does so by handling the corresponding signals, repairs the execution by performing a default operation (e.g., return zero to a read from a zero address), monitors the effects of the repair in order to contain its effects within the application process, and detaches from the application when the effects are flushed. RCV is suitable for deployed applications as it helps them survive otherwise fail-stop errors. However, it does change the behavior of the application (even if slightly) and hence is not appropriate as a workaround solution.

Input filtering. Another possible approach to work around a bug in the kernel is to filter those syscalls that trigger it. For example, VSEF uses execution-based filters to detect and then prevent exploits of a known vulnerability [29]. Sweeper monitors the execution of programs to detect attacks, analyzes the attack, deploys an antibody to prevent future exploits, and recovers the execution using the checkpoint/restore mechanism [43]. Vigilante generates a filter for preventing worms from exploiting vulnerable services [16]. However, there are important limitations for this approach to be used as a bug workaround. First, evaluating every syscall against a filter causes performance overhead. Second, discovering the exact condition and inputs under which a syscall triggers a bug is challenging. Third, there is currently no syscall filtering solution that can perform complex checks on the syscall input. Seccomp provides kernel syscall filtering but does not allow to maintain any state nor does it allow to check the arguments passed in memory.

Automated patching. The goal of this line of work is to generate a correct patch for a bug automatically. Recent efforts do so by using simulated genetic processes to fix program faults [46], leveraging static analysis to patch race conditions [20], policing invariants to curb heap buffer overflows and control flow hijacks [33], utilizing the semantic analysis of test suites to correct program states [30], and using code annotations (contracts) to generate patch candidates [45]. In contrast, we focus on a workaround for a bug. Our goal is not to properly patch the bug, rather to provide a temporary solution until a patch is ready. Hence, our work is orthogonal to this line of work.

Hot-patching is a method for changing the behavior of binaries at runtime, commonly used for delivering patches without the need to reboot [41]. Linux kernel and kernel extensions implement hot-patching by modifying the impacted functions and redirecting the execution flows [3] [2]. Recently, the urgent need for delivering security patches to fragmented Android devices has become a hot research topic. KARMA [15], VULMET [47], Instaguard [14], and Embroidery [48] extract rules and specifications from existing patches, and generate hot-patches for the fragmented Android kernel or user space binaries that are poorly maintained. These hot-patching mechanisms work assuming that the patches are available. In contrast, a workaround tries to mitigate a bug before a patch is available. Hence, our work is orthogonal to this line of work.

Error handling analysis. Several efforts have attempted to identify defective error handlers. For example, CheQ [28] locates security checks and error handlers in the kernel by searching certain patterns, and leverages this information to catch unhandled errors and other bugs. APEx [22] identifies the error handlers based on the characteristics of error paths. EPEx [19] symbolically executes the test programs and explores error paths to find the mishandled exceptions. ErrDoc [42] leverages both symbolic execution and function pair matching to identify error handlers, and it automatically detects and then fixes incorrect or missing handlers. Hector [35] walks the control graph to identify the missing release statements in the error handlers based on a list of acquisition-release function pairs. EIO [17] and Rubio-González, et al. [34] present a method that uses data-flow analysis to detect unchecked errors as they propagate in the file system code.

Hecaton identifies function pairs using a method similar to PF-Miner [26] and ErrDoc [42], which utilize string matching and path heuristics. However, there are two differences. First, PF-finder uses Longest Common Substring (LCS) as a metric as opposed to Hecaton's string similarity score discussed in §5.1. Second, PF-finder discards the paired functions with the exact same name, which can result in errors. For example, `regulator_set_voltage` function is used to both turn on and turn off a device.

9 Other Limitations

Undetected corruptions. Bowknots' effectiveness depends on catching the errors before they corrupt the system and undo the effect of the system call that causes the error. In some cases, a crash as a result of a bug (e.g., out of bound write/read to/from a non-allocated address) triggers the execution of bowknots. However, in cases that the same bug does not result in a crash, bowknots rely on kernel sanitizers (e.g., KASAN) to catch the error before it corrupts the kernel. In cases where there is no crash, kernel sanitizers do not catch the error, or they are not enabled in the kernel for performance reasons, the analyst needs to provide the check for triggering the bowknot, otherwise the bowknots might not be secure and effective.
Bugs in clean-up paths. Bowknots are workarounds for bugs designed based on the idea of undoing the effect of partially executed syscalls. However, undoing the effect of syscalls that are themselves designed for clean-up is not possible. Consider a syscall designed to destroy a few kernel objects and free all the allocated memories. If a crash happens in the middle of this syscall, where half of the kernel objects are destroyed, no bowknot could re-create the exact objects and undo the effect of this partially executed syscall. We studied the latest 100 bugs of Linux upstream kernel reported by Syzbot (as of October 2020). Our study showed that 28% of the bugs are located in clean-up paths and hence were not amenable to bowknots.

10 Conclusions

We presented workarounds for kernel bugs, called bowknots, which undo the in-flight syscall that triggers a bug. Bowknots maintain the functionality of the system even when bugs are triggered, are applicable to many kernel bugs, do not cause noticeable performance overhead, and have a small kernel footprint. Moreover, to simplify bowknots generation, we introduced Hecaton, a static analysis tool that generates bowknots automatically. Our evaluations show that bowknots are effective in mitigating bugs and security vulnerabilities and preserve the system functionality in most cases. Moreover, bowknots generated by Hecaton are complete in 64.6% of the cases.

Acknowledgments

The work was supported in part by NSF Awards #1953932, #1953933, #1846230, #1617481, and #1617513. We thank the anonymous reviewers for their insightful comments.

References

[1] iPerf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/.

[2] Livepatch. https://www.kernel.org/doc/Documentation/livepatch/livepatch.txt.

[3] Oracle Ksplice. https://ksplice.oracle.com.

[4] The ultimate, open-source X86 and X86-64 decoder-disassembler library. https://zydis.re/.

[5] american fuzzy lop. http://lcamtuf.coredump.cx/afl/README.txt, 2015.

[6] Android vs iPhone boot times tested: which one is the fastest? https://www.phonearena.com/news/Android-vs-iPhone-boot-times-tested-which-one-is-the-fastest_id69582, 2015.

[7] Qualcomm launches bug bounty program for Snapdragon chips, modems. https://www.zdnet.com/article/qualcomm-launches-hardware-bug-bounty-program/, 2016.

[8] How syzkaller works. https://github.com/google/syzkaller/blob/master/docs/internals.md, 2017.

[9] What's New in Android Security (Google I/O '17) - Video. https://www.youtube.com/watch?v=C9_ytg6MUP0, 2017.

[10] Bugs and Vulnerabilities Founds by Syzkaller in Linux Kernel. https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs.md, 2018.

[11] The Kernel Address Sanitizer (KASAN). https://github.com/google/kasan/wiki, 2018.

[12] syzbot. https://syzkaller.appspot.com/upstream, 2019.

[13] Syzkaller: an unsupervised, coverage-guided Linux system call fuzzer. https://opensource.google.com/projects/syzkaller, 2019.

[14] Y. Chen, Y. Li, L. Lu, Y. Lin, H. Vijayakumar, Z. Wang, and X. Ou. Instaguard: Instantly deployable hot-patches for vulnerable system programs on android. In *Proc. Internet Society NDSS*, 2018.

[15] Y. Chen, Y. Zhang, Z. Wang, L. Xia, C. Bao, and T. Wei. Adaptive android kernel live patching. In *Proc. USENIX Security Symposium*, 2017.

[16] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end Containment of Internet Worms. In *Proc. ACM SOSP*, 2005.

[17] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit. Eio: Error handling is occasionally correct. In *Proc. FAST*, 2008.

[18] Z. Huang, M. D'Angelo, D. Miyani, and D. Lie. Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2016.

[19] S. Jana, Y. Kang, S. Roth, and B. Ray. Automatically Detecting Error Handling Bugs Using Error Specifications. In *Proc. USENIX Security Symposium*, 2016.

[20] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated Atomicity-violation Fixing. In *Proc. ACM PLDI*, 2011.

[21] A. Kadav, M. J. Renzelmann, and M. M. Swift. Fine-Grained Fault Tolerance using Device Checkpoints. In *ACM Proc. ASPLOS*, 2013.

[22] Y. Kang, B. Ray, and S. Jana. Apex: Automated Inference of Error Specifications for C APIs. In *Proc. IEEE/ACM ASE*, 2016.

[23] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating Fuzz Testing. In *Proc. ACM CCS*, 2018.

[24] A. Lenharth, V. Adve, and S. T. King. Recovery Domains: An Organizing Principle for Recoverable Operating Systems. In *Proc. ACM ASPLOS*, 2009.

[25] J. Lettner, D. Song, T. Park, P. Larsen, S. Volckaert, and M. Franz. PartiSan: Fast and Flexible Sanitization via Run-time Partitioning. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, 2018.

[26] H. Liu, Y. Wang, L. Jiang, and S. Hu. PF-Miner: A New Paired Functions Mining Method for Android Kernel in Error Paths. In *IEEE COMPSAC*, 2014.

[27] F. Long, S. Sidiroglou-Douskos, and M. Rinard. Automatic Runtime Error Repair and Containment via Recovery Shepherding. In *Proc. ACM PLDI*, 2014.

[28] K. Lu, A. Pakki, and Q. Wu. Automatically Identifying Security Checks for Detecting Kernel Semantic Bugs. In *Proc. European Symposium on Research in Computer Security*, 2019.

[29] J. Newsome, D. Brumley, and D. Song. Vulnerability-Specific Execution Filtering for Exploit Prevention on Commodity Software. In *Proc. Internet Society NDSS*, 2006.

[30] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proc. IEEE ICSE*, 2013.

[31] S. Pailoor, A. Aday, and S. Jana. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *Proc. USENIX Security Symposium*, 2018.

[32] J. Pan, G. Yan, and X. Fan. Digtool: A Virtualization-Based Framework for Detecting Kernel Vulnerabilities. In *Proc. USENIX Security Symposium*, 2017.

[33] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proc. ACM SOSP*, 2009.

[34] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau. Error propagation analysis for file systems. In *Proc. ACM PLDI*, 2009.

[35] S. Saha, J. Lozi, G. Thomas, J. L. Lawall, and G. Muller. Hector: Detecting Resource-Release Omission Faults in Error-Handling Code for Systems Software. In *Proc. IEEE/IFIP DSN*, 2013.

[36] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Proc. USENIX Security Symposium*, 2017.

[37] S. M. Seyed Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. Amiri Sani, and Z. Qian. Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems. In *Proc. USENIX Security Symposium*, 2018.

[38] H. Shi, R. Wang, Y. Fu, M. Wang, X. Shi, X. Jiao, H. Song, Y. Jiang, and J. Sun. Industry Practice of Coverage-guided Enterprise Linux Kernel Fuzzing. In *Proc. ACM European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019.

[39] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. ASSURE: Automatic Software Self-healing Using REscue points. In *Proc. ACM ASPLOS*, 2009.

[40] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. SoK: Sanitizing for Security. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2019.

[41] A. Sotirov. Hotpatching and the rise of third-party patches. In *Black Hat Technical Security Conference*, 2006.

[42] Y. Tian and B. Ray. Automatically Diagnosing and Repairing Error Handling Bugs in C. In *Proc. ACM ESEC/FSE*, 2017.

[43] J. Tucek, J. Newsome, S. Lu, C. Huang, S. Xanthos, D. Brumley, Y. Zhou, and D. Song. Sweeper: A Lightweight End-to-end System for Defending Against Fast Worms. In *Proc. ACM EuroSys*, 2007.

[44] J. Vander Stoep. Android: Protecting the Kernel. In *Linux Security Summit (LSS)*, 2016.

[45] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proc. ACM ISSTA*, 2010.

[46] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically Finding Patches Using Genetic Programming. In *Proc. IEEE ICSE*, 2009.

[47] Z. Xu, Y. Zhang, L. Zheng, L. Xia, C. Bao, Z. Wang, and Y. Liu. Automatic Hot Patch Generation for Android Kernels. In *Proc. USENIX Security Symposium*, 2020.

[48] X. Zhang, Y. Zhang, J. Li, Y. Hu, H. Li, and D. Gu. Embroidery: Patching Vulnerable Binary Code of Fragmentized Android Devices. In *IEEE ICSME*, 2017.

Appendix

CVE-2019-2293 This vulnerability, which is rated as medium security severity, is caused by a possible null pointer dereference in Qualcomm camera `ife` module. A null pointer dereference might happen because of lack of a proper check on the `isp_resource` length variable before calling `cam_ife_mgr_acquire_hw_for_ctx()`. There are 7 functions in this bug's call stack. Hecaton overall generates 10 undo statements in these functions. Hecaton successfully detects several types of state-mutating statements and their corresponding undo statements including direct function calls, function pointers, and global variable assignments. However, our manual investigation shows that one bowknot does not correctly undo the side effect of its function. In `cam_context_handle_acquire_dev()` function `ioctl_ops.acquire_dev()`, which modifies the state of the camera device driver is called, but it is not paired with its undo function, `ioctl_ops.release_dev()` . Hecaton missed this statement because the original error handling code was not complete and did not call `ioctl_ops.release_dev()`.

After correcting the incomplete bowknot manually, when we run the PoC of this vulnerability on the mitigated kernel, all bowknots in the call stack get executed and successfully undo the side effects of the PoC. The camera device remains functional after this successful undo.

CVE-2019-1999 In function `binder_alloc_free_page()`, there is a possible double-free vulnerability due to improper locking. This vulnerability is rated as high security severity because it could lead to local escalation of privilege in the kernel with no additional execution privileges needed. In 2 functions in the call stack, there are 2 state-mutating statements, which Hecaton automatically detects and uses to generate bowknots. Our manual investigation shows that the generated bowknots are complete. Also, Hecaton-generated bowknots preserve the binder's functionality after recovery. Hence, after the recovery, the system is functional and successfully passes a binder test program that we execute. Our test program consists of two processes, a binder-server and a binder-client. It checks for successful communication between these two processes.

CVE-2019-10529 This is a use-after-free bug that can get triggered with a race condition while attempting to mark the entry pages as dirty using the function `set_page_dirty()`. Use-after-free bugs in the kernel can cause a system crash, put the system in an unexpected state, or be used in privilege escalation exploits. Automated bowknots generated by Hecaton mitigate this vulnerability and preserve the GPU driver's functionality. To test the GPU driver's functionality, we used the "GPU Mark BenchMark" application, which tests the GPU under the stress of rendering. We do not notice any difference in the result before and after Hecaton mitigates this vulnerability. Our manual investigation also shows that bowknots undo worked correctly in this case.

CVE-2019-2000 This is a bug in the binder module of the Pixel3 phone. There are 4 functions in this bug's call stack. Hecaton finds 6 state-mutating statements in these functions and generate the undo code for them in their bowknots. Our experiments show that the binder module remains functional after triggering this bug and executing the bowknots. Our manual investigation confirms that there are no other statements that result in any change in the system's state, which leaks to non-local variables.

CVE-2019-2284: This is a bug in camera driver of Pixel3 phone. There are 4 functions in this bug's call stack. Hecaton finds 10 state-mutating statements in these functions and generates the undo code for them. However, our experiments show that the Camera device loses its functionality after triggering this bug and executing the bowknots. Our investigation shows that 2 out of 4 bowknots Hecaton generates for this bug's functions are incomplete. In `cam_sensor_core_power_up()` function, there is a loop in which it turns on an array of voltage regulators. Although this function has another `for` loop in its error handling path which turns off the same array of the voltage regulators, Hecaton currently does not support multi-statement undo, and only produces a warning for the user. Our investigation shows that the bowknot generated for `cam_sensor_driver_cmd()` is also not complete. In this case, Hecaton fails to generate the complete bowknot because of the incomplete error handling code. Please note that after we manually add the missing undo statements to the mentioned functions, the system and the camera device remain functional after triggering the bug and execution of bowknots.

Syzbot bug a11372b6c9b5fd4abe1c266903bcb27e80e8f2bc This is a bug in the TTY driver of the x86 Linux-Next kernel. There are 5 functions in this bug's call stack. Hecaton locates two state-mutating functions and generates proper undo code for them. It pairs `kmalloc()` with `kfree()` and `console_lock()` with `console_unlock()` in the `con_font_get()` function. The system and TTY module remain functional after triggering this bug and execution of bowknots. Our manual investigation shows that in one of the functions, `fbcon_get_font()`, there are changes to a data structure called `font`, which is not a local variable of `fbcon_get_font()` and is provided as an input variable. Since there is no undo code to revert changes of the `font` data structure, at first glance, it seems that the bowknot does not completely undo the driver's state. However, our further analysis shows that `font` data structure is not a global variable of the driver and is defined as a local variable in `con_font_get()` function, which is the parent function of `fbcon_get_font()`. As a result, changes to the `font` data structure do not leak to the other parts of the kernel before bowknot's execution. Hence, our manual investigation shows that Hecaton-generated bowknots correctly undo the effects of partially-executed system call, which confirms the result of the functionality test.

Syzbot bug 9ad0eb3691bac24fd21ae3d8add8c08014a69f57 This is a bug in the file system of the upstream x86 Linux kernel. There are 10 functions in this bug's call stack. Hecaton finds one state-mutating statement and pairs it with its undo statement. This pair is `blk_start_plug()` and `blk_finish_plug()`, which appears twice in the execution path of this function. The file system functionality tests, including kernel self-tests for the file system, successfully pass after triggering the bug and execution of bowknots. In two functions in the call stack, we observe statements that change the non-local variables of those functions. However, similarly to the previous case, our detailed analysis shows that these non-local variables are not part of the global state of the system or the file system; they are local variables defined in the parent functions in the call stack. There is no change to the system's state, which does not have undo code in the bowknots. As a result, our manual investigation is in agreement with the functionality test.

Syzbot bug d708485af9edc3af35f3b4d554e827c6c8bf6b0f This is a bug in HCI Bluetooth driver of the x86 Linux-Next kernel. There are three functions in the call stack of this bug. Hecaton successfully pairs 4 state-mutating statements with their undo code in these functions' bowknots. We test the functionality of HCI Bluetooth driver with a user-space program that uses this driver and with the network stack self-tests of Linux kernel. The HCI Bluetooth driver and the network stack remain functional after triggering the bug and execution of bowknots. Our manual investigation shows that, in addition to the 4 state-mutating statements that Hecaton finds, there are three other function calls that can potentially change the state of the system. One is `hci_req_cmd_complete()`, which manipulates the `hdev` the driver data structure. However, our further analysis shows that this function does not get executed in the execution path of this bug. As a result, it is not a concern. The two other function calls, which can possibly change the state of system, are `hci_send_to_sock()` and `hci_send_to_monitor()`. Sending data over HCI socket changes the state of system and it is not reversible. However, our deeper analysis shows in the case of triggering this bug, these two functions return at the beginning and do not reach to the point that they change the state of system. As a result, the success of functionality test indicates the correct undo of system state in this case, too.

Syzbot bug f0ec9a394925aafbdf13d0a7e6af4cff860f0ed6 This is a bug in a network driver of the upstream x86 Linux kernel. The bug is located in HCI Bluetooth driver. There are 11 functions in this bug's call stack. Although Hecaton generates complete bowknot for 10 out of the 11 functions in the call stack for this bug, the remaining incomplete bowknot results in unsuccessful recovery. The last function in the call stack of this bug, the `__list_add()` function, is designed to add an entry to a specified location of a doubly linked list in the kernel. It modifies the two nodes that it wants to insert a new node in between. The bug occurs after processing of the first node but before the second node. At this point, the doubly link list is corrupted and there is no code to undo this corruption. We could not fix this problem in the two-hour window that we allow for manual work for each bug.

Syzbot bug 0d93140da5a82305a66a136af99b088b75177b99 This is a bug in a network driver of the upstream x86 Linux kernel. The bug is located in HCI physical layer driver. There are 11 functions in this bug's call stack. Hecaton pairs 5 state-mutating statements with their undo code in these function's bowknots. However, the network self-test result changes after triggering the bug and execution of the bowknots. Hence, the functionality test for the automatically-generated bowknot fails for this function. Our investigation shows that there is one pair of state-mutating and undo functions, which Hecaton missed because of its database's incompleteness. When we manually add `hci_conn_drop()` to the bowknot of the function `hci_phy_link_complete_evt()` to reverse the effect of `hci_conn_hold()`, the bowknots become complete and the functionality test passes successfully.