KRover: A Symbolic Execution Engine for Dynamic Kernel Analysis

Pansilu Pitigalaarachchillage
pansilu.2020@phdcs.smu.edu.sg
Singapore Management University
Singapore

Xuhua Ding
xhding@smu.edu.sg
Singapore Management University
Singapore

Haiqing Qiu
hqqiu@smu.edu.sg
Singapore Management University
Singapore

Haoxin Tu*
haoxintu.2020@phdcs.smu.edu.sg
Singapore Management University
Singapore

Jiaqi Hong†
hjqgzz@gmail.com
Freelance Researcher
Singapore

Lingxiao Jiang
lxjiang@smu.edu.sg
Singapore Management University
Singapore

ABSTRACT

We present KRover, a novel kernel symbolic execution engine catered for dynamic kernel analysis such as vulnerability analysis and exploit generation. Different from existing symbolic execution engines, KRover operates directly upon a live kernel thread's virtual memory and weaves symbolic execution into the target's native executions. KRover is compact as it neither lifts the target binary to an intermediary representation nor uses QEMU or dynamic binary translation. Benchmarked against S2E, our performance experiments show that KRover is up to 50 times faster but with one tenth to one quarter of S2E memory cost. As shown in our four case studies, KRover is noise free, has the best-possible binary intimacy and does not require prior kernel instrumentation. Moreover, a user can develop her kernel analyzer that not only uses KRover as a symbolic execution library but also preserves its independent capabilities of reading/writing/controlling the target runtime. Namely, the resulting analyzer on top of KRover integrates symbolic reasoning and conventional dynamic analysis and reaps the benefits of their reinforcement to each other.

1 INTRODUCTION

Symbolic execution (SE) is known for its capabilities of precisely describing the concerned data flows by using symbolic expressions and characterizing the control/data flow's dependence on inputs with constraints. Besides its traditional application domain (namely software testing), vulnerability discovery and automatic exploit generation (AEG) are the new applications benefiting from its reasoning and path exploration power, as shown in several techniques for dynamic kernel security analysis [11, 26].

*Haoxin is also a Ph.D. student at Dalian University of Technology, China.
†Jiaqi's contribution was done when she worked as a research fellow at SMU.

ACM Conference on Computer and Communications Security, Nov 27–30 , Copenhagen, Denmark

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

To the best of our knowledge, S2E [7] is the only symbolic execution engine supporting kernel analysis[1]. S2E is anchored at the CPU layer by integrating itself with QEMU [2]. It becomes a software CPU to host all user and kernel threads in a virtual machine. Instructions involving symbolic data are dispatched to its KLEE [4] engine for symbolic computation. The CPU-anchored approach empowers S2E to analyze system-wide code-oriented properties such as performance profiling and driver analysis. However, this approach exhibits several inherent performance and usability limitations when applied to dynamic kernel thread analysis.

S2E may expand the scope of execution more than needed if the analysis is targeted on selected threads. Since QEMU does not dictate which thread to be scheduled to run, S2E processes all threads regardless of their relevance to the analysis goal. Although the out-of-scope execution does not affect the correctness of analysis results, it could take a non-negligible toll on performance and resource consumption. More importantly, the CPU-anchored approach results in a semantic gap between S2E and the kernel since the CPU is agnostic to software semantics. To bridge the gap, S2E requires the target kernel to be instrumented beforehand so that the desired information can be exposed. For kernels with no such instrumentation, the unattended gap hinders effective dynamic analysis concerning software-level semantics. For instance, to count instructions executed within one thread requires identifying the thread's CR3 value as well as monitoring `CR3` switches. Without kernel instrumentation, it is difficult to notify QEMU and get support. Lastly, the user's analysis code lacks direct command and control over the target thread, because it is only executed via user plugins and their access to the target runtime is mediated by S2E and QEMU.

In this paper, we propose KRover, an engine for dynamic symbolic analysis for kernel threads. A kernel analyst can develop her kernel analysis program on top of KRover and invokes the latter as a library. The program benefits from the following KRover features.

- KRover takes a *live kernel thread* as input and functions without (necessarily) relying on the kernel source code. KRover neither instruments the kernel binary nor applies Dynamic Binary Translation (DBT) as in QEMU [2] or Pin [12].
- The analysis program (including KRover) uses the same address space as the target kernel's, with its binary instructions running

[1]SymQEMU [18] is claimed to be applicable for the kernel though no detailed experiments are provided.

on the CPU and referencing kernel memory using kernel virtual addresses e.g., `mov %rax 0xffffffff12345678`.
- The analysis program, like conventional dynamic analysis tools, has access to hardware features to control the target thread, such as using debug registers or `INT3` probs.
- The analysis program is empowered by KRover to direct how the target thread runs: to "slide down" from one program point to another with native execution or to single-step with symbolic execution for close monitoring and analysis. The analysis program orchestrates the interleaving of these two modes.

These features make KRover amicable for those kernel analysis tasks which demand a binary level understanding of an execution flow. For ease of presentation, we collectively call them *binary intimacy*, which has a richer implication than binary code oriented symbolic execution as provided by SymQEMU [18] and QSYM [28].

The advantages of KRover are attributed to our novel system design. The common approach of existing SE engines is to translate the target source/binary code into another representation whose execution subsequently accommodates symbolic operations. Specifically, QSYM [28] makes direct binary-to-binary compilation while all other engines compile/uplift the target's source/binary code to various forms of IR code [4, 5, 7, 17, 18, 24]. The symbolic computation logic is interposed upon the execution/interpretation of the "new-looking" target which is expected to preserve the predecessor's semantics as much as possible. Instead of relying on code translation, KRover weaves symbolic operations into the target's binary execution by using OASIS [8], a KVM-based dynamic software analysis infrastructure supporting address space coalescence and cross-space analysis.

KRover's limitations are also due to its architectural design. KRover is not capable of performing system wide symbolic analysis as it functions upon the targeted threads only. In addition, it cannot handle some hardware related instructions such as `hlt`. Since both the target and KRover runs on the same bare-metal hardware, those instructions could disrupt the underlying environment.

We have built a prototype of KRover on Linux and evaluated its performance and usability[2]. Our performance experiments upon single-path execution of 50 Linux systems call handling show that KRover is 6.8 times faster than S2E on average and 50 times maximum, with 1/10 to 1/4 of S2E's physical memory consumption. We have also run four case studies. The first two are on a buffer overflow vulnerability in a kernel module which can only be triggered after several network I/O operations prepare the necessary kernel state. In the first case study KRover is used to generalize and characterize the vulnerability and in the second we assess if the vulnerability fix is complete. In the third, we analyze a rootkit behavior to attest to the benefits of binary intimacy and the challenges of undertaking the same task with S2E. The last case study showcases that KRover's thread-centric execution is noise-free while S2E's is not. These cases also demonstrate how a user can easily develop a kernel analyzer program that uses KRover as a library and handles binary-level challenges.

Organization. In the next section, we present an overview of KRover including its software and system architecture. In Section 3, we describe the system aspects of KRover including the memory model. Section 4 presents the algorithmic aspects of KRover, including the ways to concretely or symbolically execute kernel instructions. Section 5 reports the performance of KRover, including the time and memory costs taken for one round execution and KRover component overheads, etc. Section 6 presents three practical use cases of KRover for generalization and characterization of a CVE, vulnerability fix validation, rootkit behavior analysis, respectively, and a synthetic case to show noisy execution in S2E. We discuss related work in Section 7 and conclude the paper in Section 8.

[2]The source code of KRover is available for download at https://anonymous-url.

2 OVERVIEW OF KROVER

As a binary symbolic execution engine, KRover runs kernel instructions starting from a live kernel state and explores only one path at a time by default. It outputs the path constraint and the state constraint if any. KRover can also be invoked to explore multiple paths with a given exploration strategy.

2.1 KRover Architecture

System Architecture. As depicted in Figure 1, KRover runs as a library of the user's dynamic onsite analyzer on top of OASIS [8]. With the support of OASIS, it exports the kernel thread from the guest VM to the onsite environment where, on the same vCPU core, the captured kernel thread can run natively (i.e., using the same instructions, data, and addresses as in the guest VM) and KRover can also run in the kernel's virtual address space. To highlight the two modes of execution within the onsite environment, we refer to them as the *native environment* and the *analysis environment*, respectively.

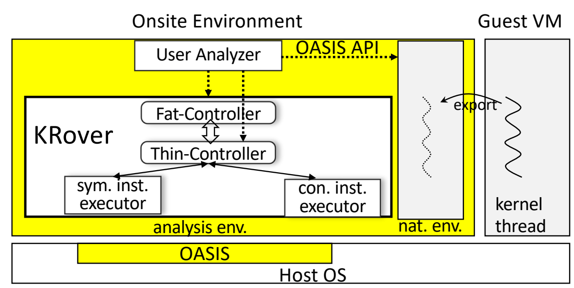

Figure 1: System architecture and main software components of KRover

Software Components. KRover consists of four main software components: two controlling modules: *Fat-Controller* and *Thin-Controller* as well as two execution modules: the Concrete Instruction Executor (CIE) and the Symbolic Instruction Executor (SIE). Fat-Controller manages execution switches between the native environment and the analysis environment, while Thin-Controller dispatches kernel instructions to the CIE or the SIE in the analysis environment, depending on whether symbols are involved. The user analyzer can directly invoke Fat-Controller or Thin-Controller to carry out executions.

Three Execution Modes & One Virtual Memory. At the highest level, Fat-Controller and Thin-Controller command how kernel symbolic execution is carried out. Fat-Controller directs the kernel execution at the function level. It dispatches a kernel function invocation to either the native environment or the analysis environment. In the former, the kernel function runs natively and exits from the environment either when the function returns or when an instruction involving symbols is encountered. In the analysis environment, Thin-Controller directs instruction executions within a function. It dispatches those instructions whose execution involves symbolic data to the SIE for an interpreted execution. Namely, they are interpreted to emulate the effects. Those instructions without symbols are dispatched to the CIE for a CPU execution without interpretation. The concrete execution is to relocate the target instructions and then execute them with the original memory content. Thus, KRover's kernel symbolic execution transits in three execution modes, i.e., *native*, *concrete*, and *interpreted* executions, according to schedules made by the two controllers at runtime. The latter two are collectively called *onsite symbolic execution* as both are on the OASIS analysis environment and Thin-Controller's management.

Empowered by the OASIS framework, interpreted and concrete executions in the analysis environment can make native read/write accesses to the target kernel virtual memory in the same way as native executions in the native environment. Hence, all three modes of executions are conducted upon the target virtual memory, namely directly referencing the target objects with the same virtual addresses (VAs). The only difference is that interpreted execution accesses addresses with symbolic data while the other two do not.

2.2 Onsite Symbolic Execution

Thin-Controller executes kernel instructions one by one using either the CIE or the SIE based on whether symbolic data is involved. It also synchronizes the software CPU context used by the SIE and the hardware CPU context when switching between interpreted and concrete executions.

Symbolic Instruction Executor (SIE). The SIE emulates the CPU execution of symbolic instructions and updates the software CPU context and/or the memory with concrete or symbolic data. If needed, it creates a new symbolic expression according to the instruction's semantics. A challenge in the SIE design is CPU flag handling. Many x86 instructions result in flag changes which affect the subsequent execution of a conditional instruction (e.g., `jnz`). Thus, the SIE must support symbolized flags. We delay flag setting in order to reduce the induced performance toll, which is the same strategy as used by the VEX IR layer in angr [24]. A key difference is that VEX's flag handling is for *every* instruction whereas KRover's flag handling only occurs when executing instructions involving symbolic data. Because of this difference, we can generalize flag operations instead of taking instruction snapshots [24].

Concrete Instruction Executor (CIE). The CIE is designed to leverage its capability of directly referencing the target's virtual memory. It relocates the concrete instruction from the original kernel VA to a new VA in KRover's space. If the memory operand is dependent on the instruction address, it rewrites the operand to reference the original VA. The relocation approach is more efficient than dispatching concrete instructions to the native environment. The speed of native execution of the latter approach cannot compensate for the expensive context switches between native and analysis environments whereas switches between SIE and CIE are merely control flow transfers.

Path Selection Strategies. When handling a conditional transfer instruction whose condition involves a symbol, Thin-Controller selects the branch to explore according to a parameter set by the user analyzer. The analyzer may use the *seeded selection*, i.e., to follow the branch according to the condition evaluation using the symbols' seed values. For example, the user can choose system call parameters used in a known vulnerability proof-of-concept as the seeds to symbolically analyze the vulnerability. The analyzer can guide KRover via path selection heuristics such as a depth-first exploration and a targeted exploration.

2.3 An Illustration of Symbolic Execution

The example in Figure 2 illustrates how KRover handles the target kernel running thread with symbolic operations. In this example, we suppose that the 4-byte data at `-0x4c(%rbp)` is symbolic when KRover attending to the first instruction at `0xffffffff810024bd`.

```
......                                                                  .....
ffffffff810024bd:   e8 ce a6 52 00        callq  0xffffffff8152cb90      (1)
ffffffff810024c2:   48 85 c0              test  %rax,%rax                (2)
ffffffff810024c5:   49 89 c4              mov  %rax,%r12                 (3)
ffffffff810024c8:   0f 84 0d ff ff ff     je  0xffffffff810023db         (4)
ffffffff810024ce:   8b 98 70 02 00 00     mov  0x270(%rax),%ebx          (5)
ffffffff810024d4:   8b 45 b4              mov  -0x4c(%rbp),%eax          (6)
ffffffff810024d7:   85 c0                 test  %eax,%eax                (7)
ffffffff810024d9:   74 40                 je  0xffffffff8100251b         (8)
ffffffff810024db:   49 83 fc 28           cmp  $0x28,%r12                (9)
......                                                                  .....
```

Figure 2: Illustration of KRover's execution. Instructions in the dash line box are natively executed. Instructions in the solid line boxes are interpreted and the rest are concretely executed.

Supposedly a static analysis or heuristics shows that the callee function in Instruction 1 is irrelevant to the symbolic data. Fat-Controller dispatches it to get the entire function call executed natively. Once it returns, Thin-Controller fetches instructions from the kernel virtual memory and executes them. It finds out that Instructions 2 and 3 do not involve symbolic data and hence dispatches them to the CIE for a concrete execution. All branch instructions are interpreted so that the Thin-Controller does not lose control. Since the first `je` depends on concrete data in `RAX`, the control transfer does not involve path selection. Suppose that the `ZF` bit in the `EFLAGS` register is not set after the `test` instruction, there is no transfer and the next instruction (i.e., Instruction 5) is fetched to execute. Thin-Controller finds out that it involves no symbols and then concretely executes it using the CIE. For Instruction 6, it determines that the source is symbolic and then interprets the execution using the SIE. As a result, `EAX` holds symbolic data and forces Instruction 7 to be interpreted as well which results in a symbolic `ZF` bit in `EFLAGS`. Instruction 8 is another conditional control transfer. Different from the first one, it depends on symbolic flags. Hence, Thin-Controller selects a path and adds the corresponding path constraints.

The example shows that three modes of execution are properly choreographed along with the instruction flow and the results from them are seamlessly coalesced into the common virtual memory. It also shows that KRover directly operates upon the running target thread's virtual memory. The runtime binary intimacy cannot be realized in other SE engines.

3 SYSTEM DESIGN

This section presents the system-level details of KRover, which are the premises for software design and implementation.

3.1 Memory Model

KRover models the target kernel's virtual memory as a sequence of bytes referenced by virtual addresses. As shown in Figure 3, it is conceptually composed of two non-overlapping zones: the *concrete zone* and the *symbolic zone*, i.e., the two sets of VA regions holding concrete and symbolic data, respectively. Physically, the concrete zone is embodied by the target kernel's own virtual memory in the guest VM. The symbolic zone is implemented within KRover's own memory in the lower half of the 48-bit space. It is organized in a sorted list of *cells* each representing one VA region in the symbolic zone. A cell holds the region's starting address, the size, and the pointer pointing to the symbolic expression logically residing in the region.

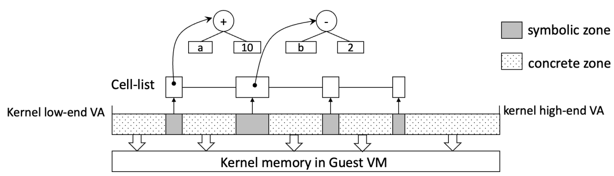

Figure 3: Illustration of the model of the kernel virtual memory: a mixture of concrete and symbolic zones

Given a target kernel's virtual address, KRover either references it directly if it is in the concrete zone; or retrieves the corresponding cell if it is in the symbolic zone. A bitmap is used to facilitate zone identification. Note that although the kernel memory holds (invalid) data at addresses belonging to the symbolic zone, KRover ensures that they are never used by the native or concrete execution.

KRover's memory model is distinct from its counterparts in other SE engines. In an IR-based engine such as KLEE [4], Mayhem [5] and angr [24], the target memory is entirely emulated within the engine's own address space. All target memory objects are physically stored in the engine's memory. In a DBT-based engine such as QSYM [28] and SymQEMU [18], the target memory is modified to accommodate the injected code and data of the engine. In contrast, KRover's memory model is exactly the same as in the target's native execution, which ensures that KRover reasons about the genuine states of the target and the derived input is effective in practice.

3.2 CPU Register Model

KRover models CPU registers to cater for instructions' register accesses. It saves the CPU context including all 64-bit control registers, general-purpose registers (GPR), and the EFLAGS register when exiting from concrete or native execution. Since the kernel may

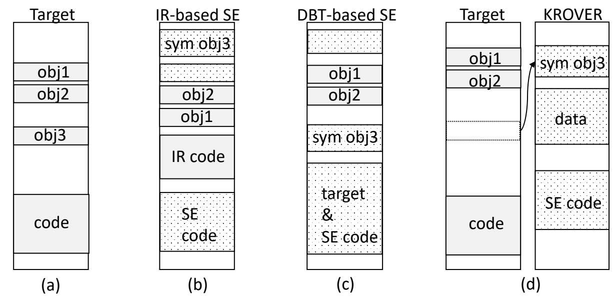

Figure 4: Address space comparison among the target native execution, IR-based SE, DBT-based SE, and KRover. The target's obj3 is symbolic while other objects are concrete.

load registers from memory contents, symbolic expressions could be propagated to registers as well. Moreover, an instruction written to a portion of a 64-bit GPR (e.g., EAX) may lead to a mixture of concrete and symbolic contents within a register. Hence, we model GPRs as a small memory region so that a unified model for memory and GPR registers greatly streamlines data movements between them.

Specifically, a GPR is treated as a 64-bit storage consisting of concrete zones and symbolic zones. Data in concrete zones are retrieved from the corresponding copy in the CPU context image while data in symbolic zones are organized in the cell list as in the memory model. Hence, KRover maintains 16 cell lists for 16 64-bit GPRs. GPRs with shorter widths such as 16 bits do not have a dedicated cell list. Since they refer to a range of bytes in the 64-bit GPR, KRover creates special pointers to the cell matching the offset (if there is a such need at runtime).

KRover currently does not support symbolic control registers (e.g., CR3). Although it can load a symbolic expression to the software copy of control registers, it cannot emulate the hardware behavior for subsequent executions. Hence, when a symbolic expression is loaded to a control register, KRover always concretizes it and adds a constraint. The treatment of the EFLAGS register is more complex as the flags therein can be set directly and indirectly by various instructions. We explain its model and usage in Section 4.4 when describing how the SIE works.

3.3 Execution Mode Switches

Because of the memory model above, neither concrete nor native execution should access any VAs in the symbolic zone as the memory thereof does not hold proper data. Thin-Controller ensures that instructions concretely executed do not use symbolic data since it finds out the referenced memory location. Fat-Controller relies on heuristics derived from static kernel code analysis to predict whether a function encounters symbolic data. To cover inaccuracy in the heuristics, KRover uses data breakpoints to detect accesses to VAs in the symbolic zone during native execution.

Data Breakpoint. Without kernel code instrumentation, KRover relies on hardware support to implement data breakpoints with different granularities. Specifically, Fat-Controller sets up byte-level data breakpoints and page-level data breakpoints before dispatching a function for native execution. The former is based on the CPU's four debug registers. The hardware throws out a debug exception when the CPU attempts to read/write up to eight bytes data from/to a memory buffer whose base address is loaded in one of the debug registers. While it has the finest granularity, the CPU only has four such registers. The page-level data breakpoints are realized by configuring the page table entry of the concerned page so that any access to the page triggers a page fault exception. While there is no limit to such data breakpoints by the hardware, they have coarse granularity and may throw out undesired page faults when the CPU accesses concrete data co-located as symbolic data on one page. Once triggered, a data breakpoint returns the control to Fat-Controller which then dispatches the flow to Thin-Controller.

3.4 Offline Path Exploration

Since one of the objectives of symbolic execution is to find paths leading to a desirable state, it is desirable for KRover to support path exploration, i.e., to start execution from one program point with different paths and terminate either when the goal is met or all paths are tried. Like other offline SE engines (e.g., angr [24] and QSYM [28]), KRover explores the target kernel only one path at a time. Path exploration in kernel space is noteworthily more challenging than in applications. A kernel thread's execution may depend on global data scattered in the vast kernel state. It is infeasible to determine the involved data beforehand. Hence, a consistent path exploration has to ensure the identical kernel state is used in all paths.

We leverage KRover's architectural advantages to tackle the challenge with the copy-on-write strategy as used in KLEE [4], angr [24] and Linux kernel. The user analyzer dictates KRover's multi-path execution by specifying the starting point of exploration as well as termination conditions such as return from a function or writing to a symbolic address. Initially, when KRover's symbolic execution arrives at the starting point, it sets the kernel memory as read-only and saves a copy of the symbolic zone data. The subsequent symbolic execution enters the copy-on-write mode. Whenever a page fault occurs due to writing to a read-only page, KRover allocates a new page with read and write permissions to replace the original one. Note that references to VAs in the symbolic zone are not affected since they are interpreted by the SIE. After terminating one-path execution, those new kernel pages are discarded. A new path exploration starts from the same starting point and the saved copy of the symbolic zone. By default, KRover uses the depth-first strategy to select the new path to explore. The user analyzer can provide a heuristics function to guide KRover's path selection. Figure 5 illustrates a path exploration with two different paths originating from the common initial state.

4 ONSITE SYMBOLIC EXECUTION

In this section, we first present the overall workflow of onsite symbolic execution followed by descriptions about how KRover handles various issues in concrete and interpreted executions.

4.1 The General Workflow

Thin-Controller handles onsite symbolic execution. Following the CPU context prepared by Fat-Controller, it fetches one instruction

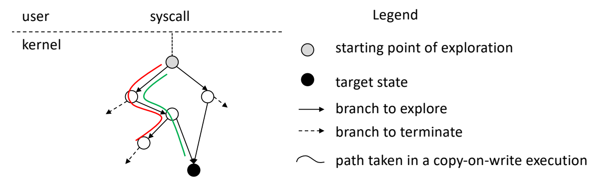

Figure 5: KRover explores multiple paths from the same starting state with copy-on-write execution.

from the kernel memory according to RIP. Thin-Controller parses the instruction to extract the memory address it references (if any) and the involved register. If either is in the symbolic zone, Thin-Controller dispatches it to the SIE for an interpreted execution. Otherwise, it dispatches all non-control transfer instructions to the CIE for a concrete execution and emulates the control transfer instruction by itself, i.e., to continue to fetch the next instruction from the transfer destination.

After the SIE/CIE completes one instruction execution, the control is returned to Thin-Control. Note that all KRover components are in the same address spaces in the analysis environment, and control transfers among them are function calls and returns. Unlike switches across the analysis environment and the native environment, they do not incur any EPT or CPU privilege switch. Nonetheless, entering and exiting the concrete execution requires a software CPU context swap since concrete execution is upon the target kernel's CPU context.

4.2 Concrete Execution

The target kernel instructions for concrete execution have to be relocated because all kernel code pages are deprived of the execution permission in OASIS's analysis environment. Moreover, instruction relocation should not change the memory address referenced by the instruction to preserve semantics and also simplify the procedure.

The performance of concrete execution is critical to KRover's overall performance because there are far more concretely executed instructions than symbolically executed ones. We thus carefully design the CIE to minimize its overhead.

Instruction Relocation. The CIE copies the target instruction into its own memory page as shown in Figure 6 below. The CIE has a modifiable code page dedicated to relocated execution. On this page, there is a 15 bytes long `nop` sled which is the placeholder for the relocated instruction of the maximum binary length. The sled is sandwiched between the CPU context loading and saving instructions.

RIP-independent instructions are directly copied to the new location for execution after the CIE. RIP-dependent instructions require re-writing before relocation. There are two types of dependency on RIP: address dependency due to the RIP-relative addressing mode and data dependency where the RIP value is used as the data. To handle both cases, Thin-Controller rewrites the instruction before dispatching it to the CIE. Specifically, it replaces RIP in the instruction binary with an unused general-purpose register. In the example shown in Figure 6, `0x12(%rip)` is replaced with `0x12(%r15)` and the CIE loads R15 with the original instruction address before running the modified instruction.

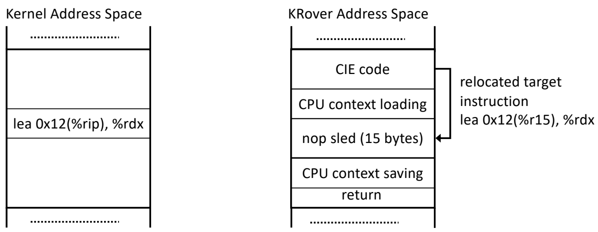

Figure 6: The CIE relocates the target instruction to its own writable code page.

CPU Context. Since the CIE's own execution and the target's concrete execution are in the same control flow, the hardware does not make any context saving or switches. Thus, the CIE deals with three forms of CPU contexts for three purposes: the one for its own execution, the target's context image shared with the SIE, and the context for the target's concrete execution. Note that the last two are kept consistent all the time.

After writing the target instructions to the designated location as in Figure 6, the CIE saves its own CPU context and loads the registers with contents from the saved CPU image. After the concrete execution, the CIE saves all registers (including EFLAGS) from the CPU to the context image so that (if needed) the SIE can carry out the subsequent interpreted execution, and then restores its own CPU context.

4.3 Interpreted Execution

The SIE's interpretation execution of an instruction is largely composed of three steps: to read the source data from the memory or registers, to make the corresponding computation, and to update the destination memory or register. Note that the SIE natively references the target kernel memory using the same virtual address as the kernel and thus memory operations of target instructions can be natively done in general. An exception scenario is sub-cell accesses, i.e., to access only a portion of the bytes represented by a cell. For instance, the symbolic zone has a cell representing eight symbolic bytes at address A and a `mov` instruction loads `RAX` register from address $A + 4$. Both memory and register accesses may encounter this scenario. Since a cell represents the whole region as one symbolic expression, accessing a portion of it requires us to treat read and write scenarios separately.

Mem/Reg Read. If the VA region represented by a cell is entirely within an instruction's read range, the SIE uses the cell as a whole. In the case of a sub-cell read, the SIE generates a new symbolic expression from the symbolic object and creates a new cell for it. Although there is address overlapping between the original cell and the new one, they have consistent expressions. Thus, a read operation incurs no change to the original cell.

Mem/Reg Write. If the VA region represented by an existing cell is entirely within the write range, the cell is updated as a whole. It is either deleted or updated depending on whether the source data is concrete or symbolic. In the case of sub-cell access, the SIE breaks the original cell into two new cells. The one out of the write range holds the new symbolic object derived from the original symbolic data, while the one within the range becomes a new cell which is then written as a whole.

An instruction's operand could involve both concrete bytes and symbolic bytes. While the SIE creates a new symbolic expression for the operand, there is no corresponding cell since it is intermediary data and does not have an address in the kernel memory. All transfer instructions are handled by the SIE, regardless of whether they are concrete or symbolic. This is to ensure that Thin-Controller does not lose control over the target's instruction flow.

4.4 Flag Handling

The bits in the flag register `EFLAGS` determine whether a transfer is taken or not when the CPU executes a conditional control transfer instruction e.g., `jc`. Instruction executions may change `EFLAGS` directly or indirectly. The former is via several flag-controlling instructions such as `clc` and `popf`. Indirect changes are much more common. After an instruction execution, e.g., `add`, the hardware checks if any flag needs to be set or cleared to reflect the state of the execution outcome. Hence, the flag values can be dependent on symbolic data used in a flag-affecting instruction. We call the two types of instructions as flag-controlling and flag-affecting instructions, respectively.

Thin-Controller determines whether a conditional transfer is made or not when `EFLAGS` is symbolic. Once a branching decision is made, the corresponding flags are concretized according to the instruction specification. Nonetheless, it has no use to set the path constraint upon the flag symbol, such as $z = 1$ where z is a symbol representing the zero flag bit since the constraint is already defined by the instruction itself. The challenge is thus how to set the constraint that captures the *causality* of flag setting rather than the flag values themselves.

An intuitive approach is to set the flags with a symbolic expression using operands appearing in the symbolically executed instruction. Nonetheless, due to the large number of flag-affecting instructions and the complexity of flag-setting, this approach not only incurs a higher development burden but also significantly takes a heavy toll on KRover performance. Moreover, the runtime cost is wasteful as one instruction's flag-setting is very likely to be replaced by the subsequent instruction execution.

Similar to the VEX IR used in angr [24], we also use a *lazy* approach for flag evaluation. Our approach is based on the assumption[3] that, between a conditional transfer instruction depending on a flag (e.g., `je`) and its preceding flag-affecting instruction setting the flag (e.g., `test`), the kernel compiler never inserts any flag-affecting instruction that sets other flags. In our approach, the SIE maintains a bit vector, called the *flag vector*, representing the symbolic states of flags in `EFLAGS`. Whenever a flag-affecting instruction is symbolically executed, the SIE sets *all* bits in the flag vector to indicate their symbolic state and also associates them with the symbolic expression created by the instruction. Hence, when the interpreted execution continues, all flags remain symbolic with their associated symbolic data evolving. When switching to concrete execution, the first flag-affecting instruction triggers all bits in the flag vector to be cleared and the association between

[3]Note that the assumption may not hold for certain kernel compilers.

flags and symbolic data to be removed. It means that the subsequent interpreted execution uses the concrete values in the saved EFLAGS.

Compared with flag-affecting instructions, it is relatively straightforward to handle flag-controlling instructions. The SIE just emulates their executions by treating EFLAGS as a general purpose register to assign a symbolic expression to it if needed. Note that only the affected bits of the flag vector are set since the SIE already has the instruction semantics.

Upon interpreting a conditional transfer instruction, the SIE checks the corresponding bit(s) in the flag vector to find out whether the condition is concrete. If so, it evaluates the condition and chooses the next instruction to run accordingly. Otherwise, it concretizes the dependent condition flag(s) according to the path selection and creates the path constraint following the instruction's arithmetic description in the Intel manual, e.g., `jle` implies "if less or equal". Thus, our approach entails a minimal overhead to each instruction execution and the cost is only incurred upon conditional transferring. As a limitation of this approach, it cannot handle scenarios where the flags are used to indicate execution errors.

4.5 Path Selection

The SIE always runs the target kernel execution along one path until the end. When encountering a symbolic branch, i.e., a conditional transfer dependent on symbolic data, it uses heuristics provided by the user analyzer to select the branch.

If no heuristic is provided, the SIE randomly decides whether the branching condition is true or false. It implies a random exploration of the kernel states. For the sake of performance, the SIE does not check the constraint solver to determine whether a branch is reachable since the kernel is highly optimized and is unlikely to have an unreachable branch. Moreover, because a false path selection can be caught by offline constraint solving or a test case replay, the performance benefit outweighs the potential inaccuracy.

A special heuristic supported by KRover is the seeded selection, similar to the one used as used in KLEE [4]. It allows KRover to explore the target following the execution path dictated by user-supplied seed values of symbols. Under the seeded selection heuristics, the SIE evaluates a symbolic conditional branch using the corresponding seed values and takes the branch accordingly.

4.6 Execution Event Detection

KRover works on the binary layer of the kernel where the software semantics (e.g., a data structure or a pointer to an object) all vanishes. As compared with other engines running with the intermediary representation code compiled from the source code, KRover's intimacy with binary has pros and cons in detecting execution events such as errors, specific function calls, etc.

On the downside, KRover does not have the built-in capability of detecting program errors such as buffer overflow/underflow where the memory access is outside of the boundary of an object. This limitation can be mitigated if the target kernel has KASAN [10] enabled at runtime. Since KRover does not change the kernel's address space layout, KASAN still reports memory errors reliably. The strength of KRover error detection is that it can catch execution errors reported by the hardware including page faults and other exceptions. More importantly, binary intimacy ensures that errors reported by KRover are exactly the same as in native executions.

By default, the SIE reports the following errors: (a) arbitrary transfer where the destination of a control transfer is symbolic; (b) arbitrary write where the destination address is symbolic. (c) arbitrary read where the source address is symbolic. In these cases, KRover reports the error to the user analyzer and stops symbolic execution if the analyzer does not concretize the involved symbol.

Note that a normal kernel behavior may also appear like an arbitrary read/write. For instance, the user analyzer symbolizes an I/O system call argument representing the location of a userspace buffer. As a result, the syscall handler's normal execution returns the I/O data to the user-supplied buffer which matches the definition of arbitrary write. To avoid such false positives, KRover introduces a special type of symbol named *buffer symbol*. The user analyzer can symbolize a buffer using a buffer symbol with a concrete size. KRover actually allocates the buffer with the correct size but treats its location as symbolic data. Different from symbolic addresses, reads and writes to a symbolized buffer can be implemented by the SIE using the buffer's concrete location, and therefore they are *not* deemed as arbitrary read or write.

While working on the binary layer, KRover is able to detect specific events in the execution. For instance, KRover has been included with the guest kernel's symbol table (a file) allowing the Thin-Controller to detect calls to certain interesting functions (e.g., kmalloc() and memcpy()) as required by the user analyzer. Also, when required, KRover dynamically acquires the corresponding function call return address before a call to a certain interesting function and uses that information to detect the return of the function call. Once detected, the Thin-Controller passes the control to the analyzer for further action.

5 PERFORMANCE EVALUATION

We develop a prototype of KRover and conduct experiments on a PC with an Intel Core i7-10700 2.90 GHz processor and 32 GB DRAM. The PC is managed by a host Linux with kernel version 5.4.125 supporting KVM. In all experiments, we launch a KVM guest with the same Linux kernel as the host.

5.1 Implementation of KRover

We implement a prototype of KRover on top of the OASIS framework [8]. In addition, we use Microsoft constraint solver Z3 version 4.8.14 [22] for constraint checking/evaluation and Dyninst version 12.0.0[20] for binary disassembly and instruction parsing. The size of the prototype binary is merely 912 KB, compiled from around 12.3K lines of C/C++ code and 250 lines of assembly code in total. A breakdown of each component's code size is shown in Table 1.

Fat-Controller	Thin-Controller	CIE	SIE	Z3-API
1901	5519	834	2526	1807

Table 1: # of SLOC in different components.

At the time of writing, the SIE supports 135 x86-64 instructions covering a wide range of operations including memory movement, arithmetic/logic/bit-wise processing, stack operations, and REP prefixed instructions, etc. We have also added ∼1.7K lines of C code to the OASIS framework [8] to provide new functionality/APIs, such as support copy-on-write execution during path exploration, page table updates, and hardware breakpoint handlers.

To evaluate KRover performance, we conduct three sets of experiments. The first set measure the run-time CPU costs of major components of KRover. These results help us to identify the performance bottlenecks. The second set measure and compare the single-path symbolic execution speed of KRover and S2E. The last one compares the memory utilization of KRover and S2E. In all experiments, we use a utility tool for instruction pre-processing. The tool disassembles the binary code and provides a cache containing instructions and extracted operands. We also use seeded execution for all experiments so that the results are comparable with S2E.

5.2 KRover Component Overhead

The specific overheads of KRover components vary with many factors including the workload, the percentage of symbolic executions, and even instruction types. To generally estimate the overhead distribution, we execute 50 Linux system calls in seeded symbolic execution mode. The results are reported in Table 2. By and large, more than half of the total execution cost is incurred by Thin-Controller. The costs incurred at the CIE and the SIE are mainly dependent on the number of instructions dispatched to them.

Thin-Controller	CIE	SIE
48%-78%	3%-37.6%	2.5%-37%

Table 2: Percentage of major components' CPU time in one round of execution.

5.2.1 *Thin-Controller Execution.*

Thin-Controller analyzes instructions fetched from the cache of the pre-processor and emulates `nop`, and all control transfer instructions regardless of whether they are concrete or symbolic. For other instructions, it dispatches them to either the SIE or the CIE. Table 3 shows the costs of Thin-Controller operations.

Instr. fetch	Instr. Analysis	Instruction Emulation			
		`nop`	`ret`	`call`	branch instruction
323	1184	15	113	1770	2072

Table 3: CPU time (in cycles) of main operations in Thin-Controller.

The instruction analysis mainly consists of the checks performed to determine if an instruction involves any symbolic data. It involves the virtual address evaluation of the memory operand in the instruction. In the seeded execution, emulations of symbolic branch instructions require an invocation of Z3 to evaluate the path constraint using seeds. The cost includes a one-time overhead of 894,826 CPU cycles for bootstrapping Z3 and 133,502 cycles per constraint evaluation.

5.2.2 *CIE Execution.*

The CIE runs in four steps, i.e., instruction relocation, rewriting, execution, and updating symbolic EFLAGS. Table 4 reports the average CPU time spent for each step. In our workload, only 0.06% to 4.90% of instructions dispatched into the CIE require rewriting. Hence the performance impact of rewriting on the overall cost of CIE is not prominent. The cost of the execution step includes the CPU context saving and loading. Note that its large overhead is due to frequent code modification which breaks the execution pipeline and invalidates relevant cache lines. Between 2% and 27% of the instructions dispatched for the CIE are flag-changing instructions incurring the overhead of `EFLAGS` image updating.

Instruction Rewriting	Instruction Relocation	Instruction Execution	EFLAGS Update	**Total**
570	38	1016	704	2328

Table 4: CPU time (in cycles) of key steps in the CIE

5.2.3 *SIE Execution.*

Table 5 presents the cost breakdown of the SIE. The dominant overhead is due to processing the concrete and symbolic operands for interpretation which involve operations on various data objects representing symbols and constants. The operation interpretation involves updating the cells with new symbolic expressions. An additional cost is incurred by the flag-affecting instructions to update the `EFLAGS` image with corresponding symbolic expressions if applicable.

Operand Processing	Operation Interpretation	EFLAGS Update	**Total**
21433	7535	1049	30017

Table 5: CPU time (in cycles) of key steps in the SIE.

5.2.4 *Summary.*

While the SIE's per instruction cost is 10 times the CIE, there are usually much less than 10% instructions dispatched to the SIE. Hence, the interpretation overhead does not dominate the overall symbolic execution cost. The invocation of Z3 is two orders of magnitude higher than the cost of one concrete instruction execution. Hence, it is the main cost contributor, especially for cases with short execution paths. According to our results, it incurs 7.5%-36% of the total execution cost.

5.3 Speed of Symbolic Execution

We benchmark KRover's execution speed against S2E. Both KRover and S2E are tasked to symbolically run a system call handler with one or several symbolic arguments. Both engines operate in seeded execution mode so that they are expected to follow the same path dictated by the common seed(s). Note that S2E in seeded mode does not fork out states for path exploration. For S2E experiments, we have built a guest kernel image with the same kernel version as the KVM guest used by KRover. The S2E guest image does not include the standard S2E plugins or sub-modules provided for analysis, which avoids unneeded executions putting S2E in a disadvantageous position.

Raw Execution Speed. We use 50 test cases each comprising a user space program issuing a distinct system call such as `setpriority`, `mmap`, `brk`, `pipe`, `lseek` and `write`. (See Appendix **??** for the full list of system calls tested). Each system call consists of predefined seed inputs along with a predefined symbolic argument(s). In some cases, the test program includes one or multiple preceding system calls to prepare the kernel state. These preceding system calls do not involve symbolic arguments. We highlight that for both S2E and KRover, the time measurement only includes the execution of the system call with a symbolic argument. We run a separate experiment to show how KRover and S2E differ in terms of handling those preparation system calls.

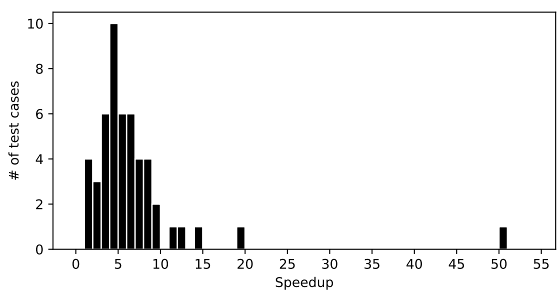

Figure 7: Histogram of the speedup (in times) of KRover's execution to S2E.

KRover outperforms S2E in all test cases with paths ranging from nearly one hundred instructions to a few thousand instructions. The consolidated results on speed-up are shown as a histogram in Figure 7. The speed-up ranges between 1.1 to 50.2 times as fast as S2E and an arithmetic average is 6.8 times. Refer to Appendix **??** for specific speed-up data for each test case.

Late Launch. KRover has the *late launch* capability of natively executing the target program until the kernel cultivates the desired run-time state for symbolic execution. To demonstrate its advantage over S2E, we design nine test cases requiring preceding system calls as the workload. We measure the time taken by KRover and S2E, respectively, to execute the state preparation work. The results are in Table 6. On average KRover is 10 times faster than S2E in preparing the kernel state. Note that in S2E experiments, these setup system calls are actually handled by QEMU as there is no symbol created yet.

Selective Dispatch. Given proper heuristics on function execution, Fat-Controller can dispatch functions without using symbolic data to natively execute. To demonstrate this capability, we use a semi-automatic tool to derive such heuristics upon five system call handlers. We use srcSlice [15] to extract the needed intelligence about a kernel function execution's dependence on variables, including intra-procedure data flows and inter-procedure data flows via function call argument passing. The tool eventually produces a variable-function dependence dictionary in which each entry is indexed by a variable defined in the source code and lists the names of the functions with dependence on them. Based on that, we prudently pick the functions which are predicted not to have symbols with high confidence for each test case. Table 7 below

Test case	Setup sys. call(s)	Execution time KRover	Execution time S2E	KRover's speedup
setpriority	getpriority	32025	140188	4.38
statx	open	71567	745785	10.42
tee	pipe, write	67420	517019	7.67
get_robust-_list	set_robust-_list	32295	137081	4.24
kcmp	getpid	31909	127384	3.99
lseek	open	57564	934760	16.24
write	pipe	56112	393307	7.01
shmat	shmget	54944	1743838	31.34
getppid	getppid	33430	147860	4.42

Table 6: CPU time (in CPU cycles) for executing system calls before symbolic execution.

shows that high-quality heuristics may save up to 90% of execution time owning to native execution. We leave it as future work to develop data flow analysis techniques for such heuristic derivation.

5.4 Memory Usage

We measure the memory footprint of KRover and S2E for single-path symbolic execution of three system call handlers. The system call `personality` handler has around 200 instructions while the `access` and `pipe2` handlers have more than 2.5K instructions each. Note that both the memory occupied by the guest VM image in S2E and the KVM guest memory have been excluded from measurement. The results are shown in Figure 8 where the Time axis is not scaled to the actual time but to indicate the start and end of execution. KRover has a significantly low and rather stable memory consumption over time while S2E uses approximately 4-10 times more memory than KRover.

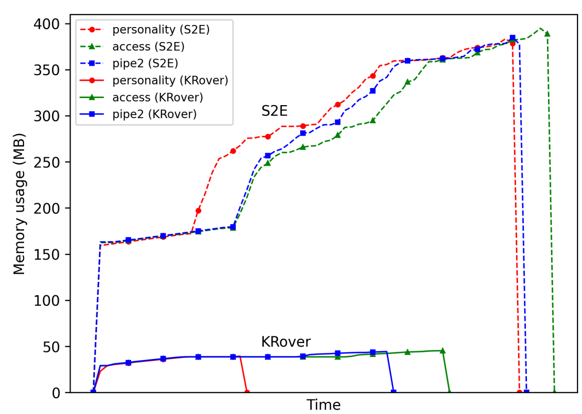

Figure 8: Memory usage of KRover and S2E in single-path symbolic execution

5.5 Path Exploration

We also run experiments to evaluate KRover's performance in path exploration. We test KRover against the `setuid` syscall handler with the `uid` argument being symbolized. The test case uses the

Statistic	**Results**				
	statx	**sched_getaffinity**	**utime**	**getcwd**	**utimes**
Total instructions in execution path	718	671	3172	842	3187
Instructions executed in native environment	631	363	2389	835	2389
Total execution time with selective dispatch	2.757	2.897	3.446	0.412	4.567
Total execution time without selective dispatch	4.938	3.635	9.962	4.417	11.566
% reduction in execution time	**44.17%**	**20.30%**	**65.41%**	**90.66%**	**60.52%**

Table 7: Cost-saving due to heuristic-based selective dispatch. Time is measured in millions of CPU cycles.

depth-first search strategy to explore all possible paths in the handler. When encountering a symbolic transfer instruction, the SIE invokes the constraint solver to determine whether a branch is satisfiable or not. In total, KRover discovered 23 paths[4] and executed 15730 instructions in 1.8 seconds. The longest path has 1052 instructions while the shortest one has 44 instructions. The initial preparation for path exploration, i.e., to freeze the target kernel, takes about 34.4 milliseconds. The subsequent turnaround time, i.e., the switching time from one path to another, is about 0.6 milliseconds. Except for five short paths, executions in most paths trigger 6 to 8 page faults requiring a 4KB-sized page allocation and 3 to 4 page faults requiring a 2MB-sized page allocation.

6 APPLICATIONS OF KROVER

The first two use cases study the CVE-2021-43267 [19] which describes a heap buffer overflow vulnerability in the Transparent Inter Process Communication (TIPC) module [13] in Linux kernels before version 5.14.16. The TIPC module processes a newly received TIPC packet in the kernel heap. The vulnerable code is inside the kernel function `tipc_crypto_key_rcv`[5] which allocates a heap buffer according to the packet header and moves the packet payload to the newly allocated buffer by two `memcpy` calls. The size argument in `memcpy` is derived from the payload without proper checking. Hence, a malformed packet may lead to an overflow at the last `memcpy` invocation. We suppose that the CVE, its POC [16], the kernel's symbol tables and definitions of kernel object structures are accessible, while the kernel source code is not. The third case study is against a third party rootkit in Github and the last one uses our own program.

6.1 Case I: Vulnerability Generalization

By generalizing the vulnerability using symbolic execution, the analyst's objective is to have a systematic way to set relevant bytes in the offensive packet for a particular goal, e.g., to write 0x20 bytes from a particular position of the packet payload outside of the buffer. Specifically, the analyst considers the following questions under the assumption that he has full control over the contents of the TIPC packet.

- Q1. How to control the overflown length? This question is equivalent to the following two subquestions: (a) whether and how the size of the victim buffer can be controlled; (b) whether and how the number of bytes written can be controlled.
- Q2. Which part of the packet corresponds to the overflown data?
- Q3. Whether the packet data is written to the target buffer without change or not?

Note that the victim buffer is dynamically allocated in the heap. Symbolic execution techniques are not mature enough to reason about Fengshui[?] attacks to determine the buffer location. Hence, the control over the victim buffer address is not in the scope of consideration.

At first glance, a symbolic execution engine taking the symbolized TIPC packet contents as the input can easily produce answers to the aforementioned questions in the form of path constraints characterizing the control flow and symbolic expressions characterizing the data flow. Nonetheless, a dive into the problem reveals several hidden challenges when a user analyzer deploys an SE engine for this task.

- It is difficult to reach the proper kernel state that starts to symbolically process the offensive packets as network I/O operations are involved according to the known PoC.
- There are several occasions demanding the analyzer to steer the SE with the target kernel's runtime data and the SE to notify the analyzer reciprocally. For instance, since a symbolized packet cannot be sent through the network I/O, the offensive packet needs to be symbolized in the memory after the I/O. Another occasion is to detect the victim buffer allocation. Although it is not in the analyst's goals to symbolically execute `kmalloc`, we need to answer Q1.a with the parameters passed to `kmalloc`. On the other hand, the SE needs to notify the analyzer when detecting out-of-bound memory read/write.
- The location of the offensive TIPC packet in the kernel heap needs to be symbolized in order to tackle Q2. Otherwise, the execution uses concrete addresses to copy data from the packet. However, a symbolized location entails reading or writing a symbolic address. Most SE engines do not support such operations.

6.1.1 *Symbolic Execution Using KRover.*

We develop a user analyzer of 207 lines of source code for this case. The analyzer (including its KRover component) runs on top of the OASIS infrastructure. The guest VM uses Linux kernel version 5.11-rc7. The analyzer uses the seeded execution mode of KRover. We explain below how the analyzer program tackles the challenges above by leveraging KRover's features.

Kernel State. We run a slightly modified POC [16] in the guest to prepare the needed kernel state. The PoC sends a series of TIPC packets on the loopback interface to prepare the kernel states and injects the vulnerability-triggering packet. We have prepared a loaded

[4]S2E only explored 3 paths under the same depth-first multi-path exploration mode.
[5]The vulnerable source code can be found at https://elixir.bootlin.com/linux/v5.15/source/net/tipc/crypto.c#L2320.

kernel module[6] to hook `tipc_udp_recv` and invoke OASIS[8]'s capability to export itself to the onsite environment when the processing of the last packet reaches the entry of `tipc_udp_recv`. In this way, the difficulty in handling network I/O is circumvented.

Locating TIPC Packet. The analyzer starts to dynamically analyze the captured thread at the entrance of `tipc_udp_recv(struct sock *sk, struct sk_buff *skb)`. The analyzer locates the TIPC packet starting from the target's RSI register since the register holds the address of the `sk_buff` object according to the x86-64 ABI. As shown in Figure 9, the analyzer equipped with kernel object definitions (see Appendix **??**) traverses the runtime kernel memory to locate the TIPC packet in the kernel heap in the same way as the kernel itself.

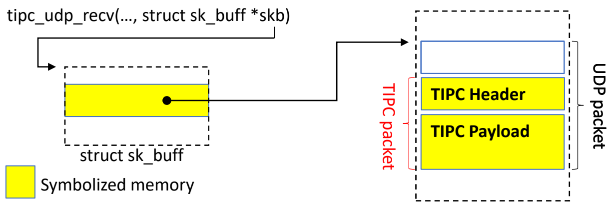

Figure 9: KRover's binary intimacy helps locate the TIPC packet by traversing the run-time kernel memory.

Symbol Declaration and Symbolic Execution Launch. The analyzer symbolizes all bytes in the TIPC packet whose concrete values are used as seeds. In addition, it uses the address of the acquired `sk_buff` object to locate the member pointer pointing to the `UDP packet` encapsulating the TIPC packet by following kernel object definitions. It then sets the UDP packet pointer as a buffer symbol (denoted by Y) which helps answer Q2. As explained in Section 4, KRover supports reads and writes to a buffer symbol. These symbolic memory regions are shown in Figure 9.

After the initial symbolization of memory data, the analyzer starts KRover symbolic execution by calling Thin-Controller with proper parameters. KRover resumes the target thread execution. In the course of running, KRover invokes the analyzer's callback functions to handle concerned events.

Event Detection and Handling. At runtime, the analyzer gets involved in the symbolic execution on two occasions. One is to detect the symbolic execution of `kmalloc` in order to answer Q1.a and also to concretize its symbolic argument as the symbolic execution does not support heap operations. The second is to turn the return value of `kmalloc` as a buffer symbol (denoted as Z) in order to answer Q1.b. Note that the `kmalloc` function is dispatched to native execution. The common memory view allows the analyzer and KRover to use kernel VAs consistently.

Addresses Reasoning with Buffer Symbols. Buffer symbols allow KRover to reason about addresses without affecting their references. As explained above, the analyzer uses two such symbols for the packet location and the victim buffer location, respectively. The symbolic execution encounters the instruction `REP movsb %ds(%rsi),%es(%rdi)`. In this memory copy instruction, the source and destination addresses are expressions of Y and Z, respectively. In addition, the present RCX is also a symbol (denoted by M) that specifies the number of repetitions of the `movsb` operation. Hence, KRover stops execution and passes the control to the user analyzer.

[6]It is also possible to use hardware virtualization to intercept TIPC execution non-intrusively.

6.1.2 Results. The analysis results are shown in Figure 10. The victim buffer size is $p - q$ which is the parameter used on the final `kmalloc` (Q1.a). The TIPC module copies N bytes to the victim buffer. N is located at the offset $q + 0x28$ to the UPD packet base. The source location of copying is the offset $q + 0x2c$ to the UDP packet location Y (Q2) while the destination location of copying is the offset $Z + 0x24$ of the victim buffer Z (Q1.b). The notations of p, q, Y, Z, and N are in Figure 10. As the source address is in the symbolized buffer holding the UPD packet, there is no change in the bytes written to the victim buffer (Q3).

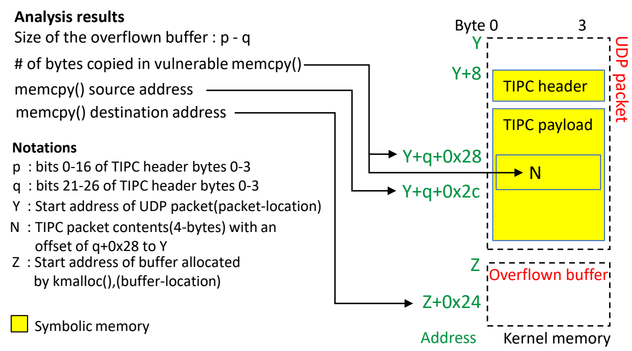

Figure 10: Summary of analysis results

Based on the generalized description of the buffer overflow, the condition to trigger the overflow is:

$$N > p - q - 0x24 \quad (1)$$

in addition to the path constraint reaching the vulnerable program point. Since N, p, q are symbols that are under the analyst's direct control, he can trigger the buffer overflow vulnerability.

6.2 Case II: Vulnerability Fix Completeness Verification

The analyst can use KRover to verify the fix[25] on Linux kernel version 5.15.1 in deed eliminates the vulnerability in CVE-2021-43267. (See Appendix **??** for the details of the fix.) For this purpose, we develop another user analyzer of 156 lines of code. As in Case I, the analyze invokes KRover and runs on top of the OASIS infrastructure. The core idea is to let KRover explore all paths and check whether the ones reaching the vulnerable program point satisfy Equation 1 discovered previously. Similar to Case I, the user analyzer exports the modified PoC after the offensive packet is received. Since Equation 1 does not involve buffer locations, the analyzer does not need to symbolize the packet location or the buffer location. Only the packet contents are symbolized.

The CVE fix is in the vulnerable function `tipc_crypto_key_rcv`. Hence the user analyzer controls KRover to symbolically execute the target thread until reaching its entrance. It then starts path exploration to find all paths within the function reaching the previously vulnerable `memcpy`. Finally, the new path constraints are analyzed to check whether Equation 1 is satisfied or not.

Late Launching of Path Exploration. The user analyzer makes use of KRover's late launch feature to delay the start of the path search until the execution reaches a desired state, i.e., the entry of `tipc_crypto_key_rcv`. Once reached, KRover takes a snapshot of the symbolic state maintained in KRover's internal structures. Note that we do not take snapshots of the guest VM. Then KRover starts the depth-first random path search.

Path Termination. KRover allows the analyzer to define the terminating conditions in the analyzer. Hence, the analyzer defines the return address of the `tipc_crypto_key_rcv` function call as the path termination condition. It also defines the vulnerable `memcpy` as the target state. Both addresses are obtained via kernel virtual memory introspection.

Fix Validation. During path exploration, KRover encounters 4 symbolic branch conditions resulting in a total of 5 paths. Two of them reach the (previously) vulnerable `memcpy`. The symbolic states of these two paths show that they have the same symbolic expressions describing the arguments in the memory copying as in Case I. It shows that the CVE fix does not affect the data flow leading to the memory copy. Nonetheless, the path constraints reported in Case II are not the same as those from Case I. There are three additional path constraints introduced by the fix as follows. The definitions of N, p, and q are as per Figure 10.

$$\texttt{p - q > 0x37} \tag{2a}$$

$$\texttt{N + 0x24 = p - q} \tag{2b}$$

$$\texttt{N <= 0x48} \tag{2c}$$

Thus, Z3 cannot solve the combination of these three constraints Equation 1, because Equations 2b and 1 are conflicting. It is therefore confirmed that the vulnerability triggering condition can never be satisfied in the fixed kernel version.

6.3 Case III: Rootkit Analysis

The third case study is to demonstrate how KRover's binary intimacy helps an analyst tackle challenges arising from an analysis task that demands thorough binary level analysis and reasoning at runtime.

6.3.1 The Analysis Task. The target rootkit is downloaded from Github[7]. Once loaded as a kernel module, it uses ftrace to hook the kernel's `kill` syscall handler so that its code executes before the genuine handler, as shown in Figure 11. It is equipped with multiple malicious functions whose executions are triggered by the signal number delivered to the kernel by its user space accomplice. It is also known that Signal #52 notifies the rootkit to hide itself by modifying the kernel's linked-list of loaded modules.

The Goal. With the aforementioned information, we (as the analyst) are interested in finding out how the rootkit prepares itself for hiding. Specifically, we hope to answer the following questions.

[7]http://github.com/h3xduck/Umbra

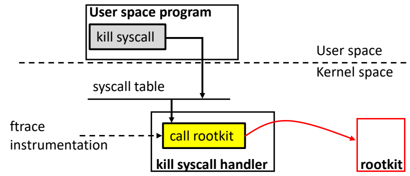

Figure 11: Illustration of the rootkit's high level working.

Q1. How is the rootkit's control flow in Sig#52 handling dependent on kernel states and its own global state?
Q2. How are these memory states shaped by the rootkit's handling of other signals?

Since it is infeasible to statically determine all memory addresses to be accessed during the rooktit's signal handling, we need both dynamic analysis to find them out and symbolic executions to reason about the control flow and data flow dependence.

6.3.2 The KRover Analyzer. With KRover, we develop an analyzer application with only 308 lines of C++ code[8] to accomplish the mission. The rootkit is loaded to a guest VM that uses Linux kernel version 5.4.150. In a nutshell, the analyzer first obtains all memory accesses in a seeded symbolic execution and then applies the results to define new symbols for symbolic path exploration. Figure 12 visualizes the workflow consisting of five steps: entering the rootkit, initialization and state backup, seeded symbolic execution for memory references acquisition, preparation for exploration, and symbolic path exploration.

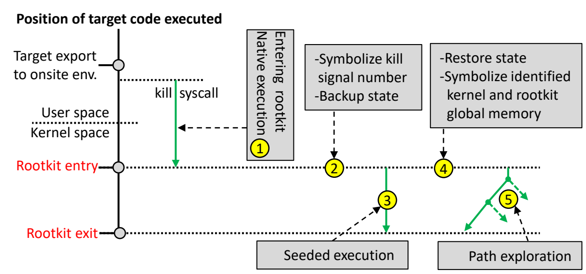

Figure 12: Five steps of the analyzer application execution

To prepare the analysis, we run an application issuing a `kill` system call with Sig#52 in the guest VM and then export the thread to the onsite environment before that system call. The analyzer then runs the five steps below. Note that the analyzer at run-time obtains from kernel symbols tables the addresses of the kill syscall handler (__x64_sys_kill), `printk` function entry and the module structure of the rootkit comprising its base address and size, etc.

Step-1: Entering the rootkit. In this step, the analyzer resumes the target thread's execution to issue the `kill` syscall and intercepts it when the flow is about to enter the rootkit code in the kernel. Because this segment of target execution is for syscall issuance in the libc and the syscall dispatch in the kernel without involving

[8]The source code is available to download from GitHub https://github.com/anacode30/krover-analyzers/blob/main/rootkit-analyze.cpp

the rootkit code, it is not to be analyzed. Thus, the target thread is natively executed. Specifically, the analyzer locates the rootkit entry and installs the INT3 probe. It then instructs KRover to natively execute the target. Once the INT3 exception is triggered by the target, the control is returned to the analyzer which then restores the original instruction at the probe site and proceeds to Step-2.

Step-2: Initial symbolization and backing-up state. Step-2 prepares for the seeded symbolic execution with the symbolized signal number and the seed value 52. Since the symbolization in KRover is directly upon the virtual memory, the analyzer has to find the address (or register) holding the signal number at the present execute state. According to the prior knowledge of the kernel, the signal number is passed to the kernel through register `RSI` which is saved to the `pt_regs` object upon kernel entry. The base address of the `pt_regs` object is then passed by the syscall dispatcher to the handler function via register `RDI`. Thus, the analyzer locates the signal number and its value with two lines of C++ code below.

```
address = &((pt_regs*)(current_rdi))->rsi
value = ((pt_regs*)(current_rdi))->rsi
```

Note that the analyzer's `current_rdi` variable holds the value of `RDI` saved by KRover upon the INT3 exception in Step-1. The analyzer then calls KRover to symbolize the bytes at `address`. Before proceeding to the next step, it backs up the target's CPU state and the symbolic state and enables KRover's copy-on-write feature so that the subsequent symbolic executions use the same starting point.

Step-3: Global memory references acquisition. The analyzer uses KRover's seeded symbolic execution to find out global memory references made by the rootkit instructions. It analyzes each target instruction in the execution flow and symbolizes the memory contents on the fly. To improve efficiency, the seeded execution skips `printk` execution as it has no impact on the task.

To interpose itself upon each instruction execution, the analyzer dispatches the target thread directly to Thin-Controller for a single-stepped execution in which the analyzer's two call-back functions are invoked before instruction dispatching for execution and after instruction execution, respectively. The first call-back function obtains from Thin-Controller the to-be-executed instruction's all memory references including the address and the size pertaining to the memory operand. Recall that these references have been extracted by Thin-Controller to determine whether any symbolic data is involved. The function then checks if the instruction is within the rootkit code section. Only when the instruction is from the rootkit, it further examines whether the involved memory addresses (if any) are from the global memory. It is determined to be in the rootkit's global if it is within the bounds of the rootkit's address layout and the RIP-relative addressing mode is used. It is in the kernel global if it appears in the kernel symbol table or it uses the `GS` segment in addressing. Once a fresh read on the global memory is detected, the analyzer symbolizes its content and flags the instruction as symbolic so that Thin-Controller subsequently dispatches it for the SIE to symbolically execute.

The second call-back function detects two scenarios. One is to check whether `RIP` points to the entry of `printk` after a call instruction execution. In this case, the call-back function skips `printk` execution by emulating the function return. The other checking is about whether the stack is balanced which indicates exiting from the rootkit function. In this scenario, the analyzer terminates the seeded execution and proceeds to Step-4.

Step-4: Preparation for exploration. In this step, the analyzer frees the memory pages written in Step-3 and restores the target state to that of the rootkit function entry with the initial CPU context and symbolic state. It symbolizes memory contents whose addresses and sizes are discovered in Step-3. It invokes Thin-Controller to launch symbolic execution with path exploration of the target (i.e., Step-5). Note that the signal number remains as a symbol.

Step-5: Symbolic path exploration. By exploring different paths in the rootkit handler, the analyzer is able to reason how the global data used by the rootkit handling of Sig#52 is shaped by the rootkit's handling of other signals. Similar to Step-3, the analyzer applies the call-back function after instruction execution to skip `printk` execution and detect the end of a path. The analyzer terminates when there is no more path to explore, and returns the resulting path constraints and symbolic expressions of each global data.

Results. During the seeded symbolic execution (Step 3), seven new symbols (S1-S7) are defined and all of them are found to be the rootkit's global data. No kernel global data accesses are detected. Four of them (S1, S2, S3 and S4) appear in the path constraint derived from the seeded execution. Hence, they are used in tandem with the symbolic signal number for path exploration (Step-5).

In path exploration, KRover explores 31 paths in total. The four bytes symbolized by S4 are written during path exploration while S1, S2, S3 are not. Path constraints of the seeded execution and the paths modifying S4 are shown in Table 8. The results suggest that the rootkit can execute one of the four paths (#4, #12, #19, #27) to make the four bytes (represented by S4) at 0xffffffffc049e240 satisfy the hiding function's path constraint.

Seeded execution	Path exploration		
Path constraint	Path #	Path constraints involving $S4$ and $Sig\#$	State update
$(S1 \leq A)\wedge$ $(S3 > A)\wedge$ $(S1+S2 \leq A)\wedge$ $(Sig\# = 52)\wedge$ $(S4 \neq 0)$	4	$(S4 \neq 1) \wedge (Sig\# = 53)$	$S4 = 1$
	6	$(S4 \neq 0) \wedge (Sig\# = 52)$	$S4 = 0$
	12	$(S4 \neq 1) \wedge (Sig\# = 53)$	$S4 = 1$
	14	$(S4 \neq 0) \wedge (Sig\# = 52)$	$S4 = 0$
	19	$(S4 \neq 1) \wedge (Sig\# = 53)$	$S4 = 1$
	21	$(S4 \neq 0) \wedge (Sig\# = 52)$	$S4 = 0$
	27	$(S4 \neq 1) \wedge (Sig\# = 53)$	$S4 = 1$
	29	$(S4 \neq 0) \wedge (Sig\# = 52)$	$S4 = 0$

Table 8: Results of the analysis. 'A' (0xffffffff81005217) is the return address of the kernel's `kill` syscall handler.

6.3.3 *Benefits from Binary Intimacy.*

All four aspects of binary intimacy are embodied in this case study and greatly simplify the development of the analyzer. Firstly, the analysis is restricted to the target kernel thread serving syscalls from the designated user space program. Secondly, thanks to the same address space setup, our analyzer code effortlessly references the kernel memory using kernel addresses throughout all steps without using any intermediary.

Thirdly, the hardware-supported facility is applied upon the target execution in Step 1. The analyzer can also use the debug register to set a hardware breakpoint on the rootkit entry. Lastly, the analyzer controls how the target runs at different stages of analysis. The target undergoes native execution, seeded symbolic execution and symbolic path exploration and is forced to skip `printk` in Steps 3 and 5. In short, KRover's binary intimacy unites the analyzer's role in dynamic kernel analysis in the conventional sense and the other role in symbolic analysis for data and control flow reasoning and empowers the analyzer to seamlessly switch between them.

It is infeasible to accomplish the task using S2E with an off-the-shelf guest kernel. Even with an S2E instrumented kernel, the task is onerous. The difficulty stems from the task's need for applying fine-grained dynamic kernel analysis to guide symbolic execution. On the one hand, conducting dynamic analysis and symbolic executions in separated sessions faces the address consistency problem because the rootkit is loaded at different addresses in different launches. On the other hand, S2E's sophisticated design and complex system engineering make itself unfriendly to nimble dynamic analysis and difficult to yield the control to user plugins. For example, to the best of our knowledge, plugins cannot change S2E's symbolic execution mode at runtime. Moreover, as the target runs on QEMU, it cannot truly benefit from hardware features e.g., to slide down the path until a breakpoint.

6.4 Case IV: Noise Free Execution

The last case study is to show that KRover inherently confines the symbolic execution to the target thread whereas symbols in S2E are propagated to unrelated threads which (if not properly filtered) results in noisy executions in analysis.

Figure 13 shows our S2E test program. It defines a global variable `flags` and forks out a child process. Within the child process, `flags` is symbolized and passed in the `getpriority` system call (Line 7). The parent process sleeps for 20 seconds to ensure that it continues the execution *after* the child's symbolization step. Upon being woken up, the parent issues the `open` system call taking `flags` as one of the arguments (Line 11).

```
1  int flags = 0;
2  void noise(){
3    if (fork() == 0){
4      printf("at child process\n");
5      //symbolize the global variable 'flags'
6      s2e_make_symbolic(&flags, sizeof(flags), "which");
7      getpriority(flags, 0); // target system call
8    } else {
9      sleep(20);
10     printf("at parent process\n");
11     open("/proc/cpuinfo", flags);
12   }
13 }
14 int main (){ noise(); return 0; }
```

Figure 13: Target program example showing noise execution

Apparently, the parent's `open` system call is to be concretely executed as the symbolization occurs in the child process only. However, the experiment with S2E reports a different outcome. Besides the four kernel states explored for the child, S2E has also explored 2649 kernel states for the parent's `open` for about one hour before it is forced to stop. Supposing that the analysis goal is about the child's `getpriority` handling, S2E's exploration for the parent obviously wastes resources and inundates useful results with a flood of noises.

This outcome is due to a composite effect of the kernel's copy-on-write strategy for process forking and the inherent working of S2E/QEMU. Since neither the parent nor the child process modifies `flags`, the kernel does not allocate a new page for the child's `flags`. As a result, the parent and child access the same physical memory location for `flags` to issue system calls. The `s2e_make_symbolic` execution in the child first labels the memory content of `flags` as symbolic. Then, during the execution of `open`, S2E/QEMU fetches `flags` from the memory and detects the label indicating a symbolic region. It thus symbolically executes the system call.

While our test program is a synthetic one, it does reveal that S2E's memory symbolization has a global effect due to its execution being anchored at QEMU. For monolithic kernels, all threads share the same kernel address space. For instance, a global kernel variable (e.g., `jiffies`) is possibly referenced by any kernel thread. A prudent user of S2E needs to figure out how symbolic kernel data is possibly used. If necessary, S2E or QEMU can be modified to trace `CR3` so that noisy executions and/or outputs are properly filtered. In contrast, when the test program is analyzed using KRover, only the child process is symbolically executed while the parent uses the concrete value of `flags` for `open` invocation.

Caveat. Note that whether the global effect of symbolization is a side effect or a desirable feature is dependent on the scope of symbolic analysis. It becomes a nuance if the analysis focuses on a selected thread. Thus, while S2E is suitable for system-wide analysis, KRover is a better tool for thread-centric analysis.

7 RELATED WORK

Symbolic Execution Engines. In the literature, two prevalent flavors of symbolic execution engines are implemented: IR-based and IR-less [17]. A large portion of the engines fall into the first category. Those engines typically first transform the target program, either from source code or binary code, into IR and then perform the program analysis by interpreting the transformed IR. KLEE [4] and AEG [1] use LLVM bitcode as their IR to analyze the target program with source code. For analyzing binaries, Mayhem [5] transforms the binary to the IR from the BAP platform [3]. S2E [7] leverages both LLVM bitcode and QEMU's IR (i.e., TCG). Angr [24] utilizes the VEX from Valgrind framework [14]. Recent two compilation-based approaches, SymCC [17] and SymQEMU [18] keep the use of IR. However, instead of interpreting IR, they instrument the symbolic analysis capabilities into IR (LLVM bitcode for SymCC and TCG for SymQEMU) and build symbolic execution right into the binary to speed up the execution. The IR-less engines directly execute the target program without involving IR transformation. Triton [23] and QSYM [28] instrument the unmodified machine code at the run time assisted by Intel Pin [12] and directly run the instrumented machine code to support symbolic execution.

Symbolic Execution in Kernel Analysis. Considerable efforts are paid in recent years on improving the security of the kernel by levering the incomparable capabilities of symbolic execution.

In terms of the detection of kernel bugs/vulnerabilities, SymDrive [21] makes device inputs to the driver symbolic to eliminate the need for real devices and allow symbolic execution on the complete range of device inputs to detect potential vulnerabilities in Linux drivers and FreeBSD drivers. HFL [11] performs hybrid fuzzing, i.e., combining symbolic execution and fuzzing, to detect kernel vulnerability. UBITect [29] combines flow-sensitive type qualifier analysis and symbolic execution to perform precise and scalable Use-before-Initialization bug detection. Since not every vulnerability is created equal, many other works are devoted to sorting those vulnerabilities by assessing their exploitability and/or generating exploits for them. FUZE [27] facilitates exploiting kernel Use-After-Free vulnerability by exploring different vulnerability capabilities with fuzzing and leveraging symbolic execution to construct Return-Oriented Programming (ROP). AEM [9] is an automated exploit migration technique to facilitate cross-version exploitability assessment for Linux kernels. Specifically, it symbolizes syscall arguments and enables the symbolic execution to (1) adjust the arguments and (2) collect constraints to compare their exploitability. SyzScope [30] combines fuzzing and static analysis with symbolic execution to evaluate the impact of a given seemingly low risk bug and uncover its potential high risk impact, where symbolic execution is used for validating the feasibility of reaching high-risk impacts and evaluating possible primitives e.g., arbitrary write and constrained write. KOOBE [6] utilizes symbolic execution to facilitate exploit generation of kernel out-of-bounds write vulnerabilities.

Comparison with Other Engines. All existing SEs handle symbolic and concrete executions in a unified fashion. IR-centric engines [4, 5, 7, 24] always interpret IR instructions regardless of whether symbolic data is accessed. While no code modification is incurred at runtime, interpretation-based executions are inherently slower than binary executions. Different from the existing, KRover is a kernel symbolic execution engine catered for dynamic kernel analysis, which directly operates upon a live kernel thread's virtual memory and weaves symbolic execution into the target's native executions. Notably, benefited from waiving of lifting target binary to IR, KRover has shown its significant performance improvement, which enables the fast symbolic execution. KRover's binary intimacy becomes advantageous in overcoming binary-level challenges of kernel analysis. However, it should also be noted that compared to the IR-centric engines KRover's execution results in a larger semantic gap between the executed binary instructions and the source code. Binary-centric engines [18, 28] rewrite all machine instructions and instrument them with symbolic handling logic. These engines outperform their IR-based counterparts. As compared with KRover, they heavily rely on dynamic binary translation (DBT) which is more complicated than the techniques used in KRover. Moreover, DBT reshapes the target's address space, which is likely to introduce analysis inaccuracy. Note that KRover's CIE only relocates the addresses of instruction instead of their memory operands and hence preserves the address space.

8 CONCLUSION

We have presented KRover as a kernel symbolic execution engine for dynamic kernel analysis. Built upon OASIS, KRover conducts symbolic execution directly upon the target kernel's live virtual memory and weaves it into the target's binary execution. KRover is more suitable for thread-centric dynamic kernel analysis due to its binary intimacy, high speed, noise free nature and programable invocation. Our four case studies show that an analysis program can maintain its capabilities of engaging with the target while using KRover as a library for symbolic execution. In short, KRover allows conventional dynamic analysis and symbolic reasoning to be integrated with mutual reinforcements.

ACKNOWLEDGEMENT

We thank anonymous reviewers for their constructive comments and suggestions.

REFERENCES

[1] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. 2011. AEG: Automatic Exploit Generation. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium.*

[2] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX annual technical conference, FREENIX Track*, Vol. 41. Califor-nia, USA, 46.

[3] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. 2011. BAP: A binary analysis platform. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23.* Springer, 463–469.

[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[5] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P).*

[6] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. 2020. KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities. In *Proceedings of the 29th USENIX Security Symposium.*

[7] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E Platform: Design, Implementation and Applications. *ACM Transactions on Computer Systems* 30, 1 (2012).

[8] Jiaqi Hong and Xuhua Ding. 2021. A Novel Dynamic Analysis Infrastructure to Instrument Untrusted Execution Flow Across User-Kernel Spaces. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P).*

[9] Zheyue Jiang, Yuan Zhang, Jun Xu, Xinqian Sun, Zhuang Liu, and Min Yang. 2022. AEM: Facilitating Cross-Version Exploitability Assessment of Linux Kernel Vulnerabilities. In *2023 IEEE Symposium on Security and Privacy (SP).* IEEE Computer Society, 588–603.

[10] The kernel development community. 2022. The Kernel Address Sanitizer (KASAN). (2022). Retrieved Jan 4, 2023 from https://docs.kernel.org/dev-tools/kasan.html

[11] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium.*

[12] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[13] Jon Maloy. 2022. TIPC Programmer's Guide. (oct 2022). Retrieved Jan 4, 2023 from http://tipc.io/programming.html

[14] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices* 42, 6 (2007), 89–100.

[15] Christian D Newman, Tessandra Sage, Michael L Collard, Hakam W Alomari, and Jonathan I Maletic. 2016. srcslice: A tool for efficient static forward slicing. In *Proceedings of the 38th International Conference on Software Engineering Companion.* 621–624.

[16] Peter. 2021. Local PoC exploit for CVE-2021-43267. (nov 2021). Retrieved Jan 04, 2023 from https://haxx.in/files/blasty-vs-tipc.c

[17] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *Proceedings of the 29th USENIX Security Symposium.*

[18] Sebastian Poeplau and Aurélien Francillon. 2021. SymQEMU: Compilation-based symbolic execution for binaries. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium.*

[19] CVE Program. 2021. CVE-2021-43267. (nov 2021). Retrieved Jan 4, 2023 from https://www.cve.org/CVERecord?id=CVE-2021-43267

[20] Dyninst Project. 2021. dyninst. (2021). Retrieved Jan 4, 2023 from https://github.com/dyninst/dyninst/tree/v12.0.0

[21] Matthew J Renzelmann, Asim Kadav, and Michael M Swift. 2012. SymDrive: Testing drivers without devices. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 279–292.

[22] Microsoft Research. 2021. Z3. (2021). Retrieved Jan 4, 2023 from https://github.com/Z3Prover/z3/tree/z3-4.8.14

[23] Florent Saudel and Jonathan Salwan. 2015. Triton: A dynamic symbolic execution framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes*. 31–54.

[24] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.

[25] Max van Amerongen. 2021. tipc: fix size validations for the MSG_CRYPTO type. (oct 2021). Retrieved Jan 4, 2023 from https://github.com/torvalds/linux/commit/fa40d9734a57bcbfa79a280189799f76c88f7bb0

[26] Yan Wang, Chao Zhang, Xiaobo Xiang, Zixuan Zhao, Wenjie Li, Xiaorui Gong, Bingchang Liu, Kaixiang Chen, and Wei Zou. 2018. Revery: From Proof-of-Concept to Exploitable (One Step towards Automatic Exploit Generation). In *Proceedings of ACM CCS*.

[27] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. 2018. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 781–797. https://www.usenix.org/conference/usenixsecurity18/presentation/wu-wei

[28] Insu Yun, Sangho Lee, , Meng Xu, Yeongjin Jang, and Taesoo Kim. 2020. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 2th USENIX Security Symposium*.

[29] Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V Krishnamurthy, and Paul Yu. 2020. UBITect: a precise and scalable method to detect use-before-initialization bugs in Linux kernel. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 221–232.

[30] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian. 2022. {SyzScope}: Revealing {High-Risk} Security Impacts of {Fuzzer-Exposed} Bugs in Linux kernel. In *31st USENIX Security Symposium (USENIX Security 22)*. 3201–3217.