Minoris: Practical Out-of-Emulator Kernel Module Fuzzing

Yangxi Xiang, Feng Wang, Yuan Chen, Qiang Liu, *Member, IEEE*, Haoyu Wang, Jiashui Wang, Lei Wu, Chaoyuan Chen, and Yajin Zhou

***Abstract*—Vulnerabilities in the Linux kernel can be exploited to perform privilege escalation and take over the whole system. Fuzzing has been leveraged to detect Linux kernel vulnerabilities during the last decade. However, existing kernel fuzzing techniques highly use QEMU/KVM as the underlying infrastructure, thus suffering from unnecessary costs due to user-kernel context switch and kernel-emulator context switch. This degrades the fuzzing performance. In this paper, we propose a kernel module fuzzing framework named MINORIS. It moves the kernel module under testing (KMUT) out of both real kernel and emulator, thus eliminating unnecessary context switches. However, implementing such a system requires solving the dependency challenges. We solve these challenges by automatically linking kernel module with LKL, and performing initialization functions on-demand to prepare the required status. Besides, a hardware-emulation library is proposed to provide underlying hardware support. Our system not only improves the fuzzing speed but also can easily integrate mature fuzzing techniques, such as user-space memory sanitizer. We evaluate MINORIS on five different KMUTs. Compared with the state-of-the-art solution, MINORIS achieves an average execution speedup from $\times$3.31 to $\times$7.38. It improves the fuzzing throughput ($\times$102.58), explores more code coverage (89.51% more branches), and detects 6 new bugs.**

***Index Terms*—Operating systems, security, fuzzing.**

I. INTRODUCTION

LINUX kernel, as one of the most popular OS kernels, its code size[1] has grown from 6.2 million lines of source code in v2.6.12 (2005) to 35.7 million lines of source code in v6.12 (2024). With the kernel's increased code size and complexity, the number of CVEs in the Linux kernel has grown from 264 in 2005 to 3,109 in 2022. By exploiting these vulnerabilities, an attacker can escalate privileges to perform malicious tasks, e.g., leaking other users' privacy and/or taking over the whole computing infrastructure. The increased number of vulnerabilities and serious consequences urge the need for an automatic Linux kernel bug detection system.

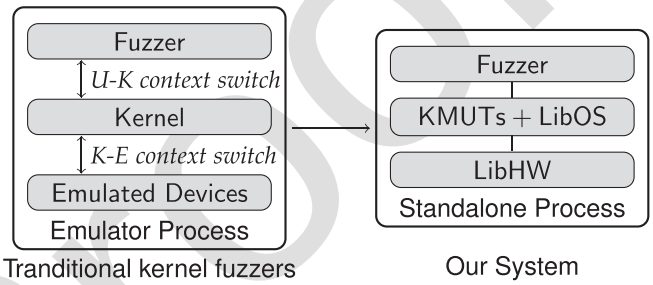


Fig. 1. Typical kernel fuzzers [2], [3], [4] and our system. Our system moves the kernel modules into user space, removing the user-kernel and kernel-emulator context switches.

Fuzzing, an effective testing technique, has been adapted to discover kernel vulnerabilities. Some kernel fuzzing systems focus on specific kernel components like file systems, TCP stacks, and USB driver stacks [5], [6], [7], [8], [9], [10]. Some of them are generally purposed for the whole kernel [11], [12]. Furthermore, sophisticated techniques, such as symbolic execution [12], [13], [14], vulnerability pattern summary [15], [16] and exploitation assessment [17], [18], [19], [20], [21] are applied to find more difficult-to-discover vulnerabilities.

Specifically, Trinity [22], started in 2006, was widely used for testing syscalls with a random but type-aware fuzzing fashion. The testing syscalls are executed in a real kernel. Therefore, it achieves a native performance but might corrupt the host system. With the development of hardware virtualization (e.g., Intel VT-x), software emulators (e.g., QEMU) can improve performance with hardware virtualization acceleration. Accordingly, kernel fuzzing frameworks that combine with emulators (e.g., QEMU/KVM) have been proposed [2], [3], [4]. Syzkaller, a representative state-of-the-art emulator-based kernel fuzzer, has hunted thousands of kernel bugs. Though with hardware virtualization, the emulator-based kernel fuzzing systems still have high-performance overhead. Recently, techniques for improving the performance overhead have been proposed. Some of them reduce the emulation overhead [9], [23], [24], but introduce non-negligible manual efforts and still cannot achieve native performance. Some of them try to complete the hardware dependency [6], [14], [25], [26], but may introduce extra overhead or high false positives on emulation.

Received 27 July 2024; revised 16 September 2025; accepted 16 September 2025. This work was supported in part by the National Key R&D Program of China under Grant 2022YFE0113200 and in part by the National Natural Science Foundation of China (NSFC) under Grant U21A20464. *(Corresponding author: Yajin Zhou.)*

Yangxi Xiang, Yuan Chen, Qiang Liu, Lei Wu, and Yajin Zhou are with the College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang 310027, China (e-mail: xyangxi5@gmail.com; yuanchen96@zju.edu.cn; cyruscyliu@gmail.com; lei_wu@zju.edu.cn; yajin_zhou@zju.edu.cn).

Feng Wang, Jiashui Wang, and Chaoyuan Chen are with Ant Group, Hangzhou 310000, China (e-mail: yanuo.wf@antgroup.com; jiashui.wjs@antgroup.com; yaoguang.cyg@antgroup.com).

Haoyu Wang is with the School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: haoyuwang@hust.edu.cn).

Digital Object Identifier 10.1109/TDSC.2025.3616496

[1] The statistic about the code size of the kernel is reported by CLOC tool [1].

Motivation: Even though hardware virtualization can be leveraged for accelerating OS kernel fuzzing, emulator-based kernel fuzzing still suffers from unavoidable performance overhead on *user-kernel context switch* and *kernel-emulator context switch* (see the left of Fig. 1). For the *user-kernel context switch*, a cross-ring system call will take up additional CPU time for saving and restoring contexts. For the *kernel-emulator context switch*, the emulator introduces overhead when forwarding instructions and I/O requests. The performance overhead is non-negligible. This can be seen from the fact that the community is trying to avoid VM-Exit events in QEMU/KVM raised by the kernel [27]. These two types of overhead become much more significant in kernel fuzzing scenarios since fuzzing involves many system calls and frequent hardware interactions. As a result, the effectiveness of fuzzing is hindered.

Compared to fuzzing kernel modules in emulators, intuitively, it is more beneficial to fuzz a kernel module as a native user-space program on the host computation platform. By doing this, it removes the overhead introduced by *user-kernel context switch* and *kernel-emulator context switch*, which is naturally unnecessary in kernel fuzzing.

Our Approach: In this work, we propose a fuzzing framework named MINORIS to fuzz kernel modules (extracted from the AArch64 and the x86-64 Linux kernel) as user-space programs on the host computation platform (x86-64 architecture). Specifically, we choose to get the kernel module under testing (KMUT in this paper) running with the fuzzer as a standalone user-space process (as shown in the right part of Fig. 1). Additionally, a user-space library OS (i.e., LKL in our design) and a hardware emulation library (i.e., LibHW) are integrated to provide the necessary OS kernel and underlying hardware environment support, respectively. By doing so, we could implement a high-performance kernel fuzzing framework without *kernel-emulator context switch* nor *user-kernel context switch* overhead.

MINORIS brings two benefits. First, it accelerates the kernel fuzzing at native speed without additional requirements. The typical emulator-based kernel fuzzers launch the testing kernel inside QEMU with the KVM acceleration, which demands a homogeneous CPU architecture between the testing and host kernel. In contrast, MINORIS removes such limitations as MINORIS compiles the KMUT to run as a user program on the host platform, regardless of the original architecture hosting the kernel module. Second, fuzzing kernel module as a user-space program enables the easy integration of the off-the-shelf user-space fuzzing techniques. With the popularity of fuzzing, many techniques have been proposed to improve fuzzing effectiveness, such as using symbolic execution or static program analysis to assist the fuzzer in exploring deeper program states. Most of these techniques target user-space programs, and porting them to kernel-fuzzing systems would require considerable manual effort. With MINORIS, the user-space fuzzing techniques can be easily adapted for fuzzing the kernel modules.

Challenges: Despite the benefits, building and fuzzing a kernel module as a user-space program is not as simple as it looks. Technique challenges are faced because the kernel module is not self-contained and has external dependencies. We characterize these challenges into the following three categories.

Challenge I. Symbol Dependency at Compile Time: Due to the inconsistency between the linked user-space kernel library (LKL) and the original kernel source code that the kernel module depends on, compiling and linking a kernel module into a user-space program may bring symbol dependency issues, such as undefined references, multiple definitions, or inconsistent declarations. The symbols include macros, data structures, variables, functions, etc.

Challenge II and III. State Dependency and Hardware Dependency at Runtime: Although LKL is provided as the user-space kernel library, the corresponding kernel runtime state must also be set up properly before fuzzing, especially for the dynamically probed devices. This can ensure that the KMUT is accessible from system call interfaces issued by the fuzzer. Moreover, during fuzzing, the KMUT needs to interact with the underlying hardware to function properly. A lack of hardware interaction would greatly hinder fuzzing execution. We conclude these two runtime issues as state dependency and hardware dependency, respectively.

Our Solution: We solve the above dependencies to fuzz kernel modules as user-space programs. Specifically, we propose SemaLinker to solve symbol dependency at compile time automatically. SemaLinker transforms the source code of the kernel module by resolving symbol dependency issues and linking with LKL to generate an executable. Next, to solve state dependency, MINORIS retrieves the required initialization functions of the KMUT automatically and explicitly makes LKL invoke them so that the KMUT gets initialized properly. Moreover, the necessary system calls are also identified and issued before fuzzing to prepare the user-space states. Lastly, a hardware-emulation library (i.e., LibHW) is coupled to provide underlying hardware support. The interaction between the KMUT and the underlying (emulated) hardware in LibHW is bridged by MINORIS automatically. LibHW is implemented based on QEMU to utilize its rich virtual device assets and mature framework for adding new devices. With the above dependencies solved, we can integrate various user-space fuzzing techniques and fuzz the KMUT with high throughput.

We comprehensively evaluated MINORIS on various targets. We take the x86-64, AArch64, and Android kernels as targets. First, our evaluation shows that SemaLinker can automatically fix the symbol dependency of kernel modules and successfully link with LKL to generate user-space programs. Second, MINORIS can achieve an average execution speedup from $\times$3.31 to $\times$7.38 by removing the user-kernel context switch and kernel-emulator context switch. Third, we compared MINORIS with the state-of-the-art kernel fuzzer syzkaller [2] for fuzzing throughput, code coverage, and bug finding. MINORIS greatly improves the fuzzing throughput ($\times$102.58), has more code coverage (89.51% more branches), and detect 6 new bugs.

Contributions: This work makes the following main contributions.

- We summarize two performance issues for emulator-based kernel fuzzing and propose a new fuzzing framework MINORIS to solve these issues. It tests kernel modules as user-space programs on the host machine to improve the performance and enables the easy integration of user-space fuzzing techniques.

- We locate three challenges when implementing the system, including symbol dependency during compilation, state dependency, and hardware dependency at runtime. We also propose corresponding solutions for these challenges.
- We implemented a prototype and evaluated the scalability of transforming the kernel module into a user-space program, the execution speedup, and the effectiveness. Our evaluation shows that MINORIS can gain $\times$3.31 to $\times$7.38 execution speedup compared with the QEMU/KVM-based kernel fuzzing systems and can greatly improve the fuzzing throughput, achieve high code coverage, and detect more bugs.

II. BACKGROUND

A. Kernel Module

Kernel modules enhance OS functionality by adding features like device drivers. They can be either statically linked into the kernel image (configured with `y`) or dynamically loaded at runtime (configured with `m`).

A loadable kernel module can be loaded using the `modprobe` command in two stages. First, undefined symbols are resolved via the kernel's symbol table (`kallsyms`). Second, the module's memory is initialized, and its initialization function is invoked via the `module_init` entry point. Once loaded, the module runs in the kernel's address space with full system privileges. In contrast, statically linked modules are part of the kernel image and are initialized during system boot using the same initialization functions.

B. Kernel Fuzzing

Fuzzing has become a key technique for uncovering kernel vulnerabilities. Compared to user-space fuzzing, kernel fuzzing poses distinct challenges. First, kernels need to interact with diverse hardware devices. Emulator-based approaches attempt to handle hardware dependencies through custom device emulation but suffer from context-switching overhead and fail to provide the rich hardware dependencies needed for effective testing. Second, generating valid test cases requires synthesizing complex structured inputs, like syscalls, whose specifications are buried in kernel code. Third, the kernel's low-level execution environment complicates state monitoring and result interpretation, and scalability issues grow during extended fuzzing campaigns.

C. Context Switches During Kernel Fuzzing

Kernel fuzzing efficiency is affected by two main kinds of context switch overhead.

User-Kernel Context Switches: Real syscalls require the cross-ring transitions, switching between user mode and kernel mode, except for those optimized calls executed mostly in user space via mechanisms, like `clock_gettime` via vDSO. These transitions, involving saving and restoring CPU and memory states, typically take hundreds of nanoseconds to microseconds per call, depending on hardware. While vital for security and resource management in regular programs, this overhead accumulates noticeably during kernel fuzzing, where real syscalls are frequent.

Kernel-Emulator Context Switches: Hardware operations or privileged instructions cause VM-exit events, triggering context switches to the hypervisor (e.g., QEMU/KVM) for instruction emulation or I/O handling. This adds notable latency, as seen in ongoing efforts to reduce such exits [27]. Fuzzing workloads with frequent hardware interactions are particularly affected by this overhead.

D. Lkl

Our framework relies on LKL to solve symbol dependency for KMUTs. LKL, a LibOS, re-implements a new architecture interface that adapts to the host environment for Linux and therefore makes the OS kernel a user-space library. By linking it, an application can send syscalls to the embedded kernel and gain nearly complete kernel functionality. However, LKL is designed to run a user application and cannot provide hardware dependency for KMUTs.

III. SYSTEM DESIGN

The goal of MINORIS is to provide a framework to fuzz kernel modules (extracted from the AArch64 and the x86-64 Linux kernel) as user-space programs on the host computation environment, i.e., the x86-64 desktop PC. By doing so, the kernel module fuzzing can achieve high throughput while at the same time can directly integrate the off-the-shelf user-space fuzzing techniques, e.g., ASAN and SYMCC [28], [29]. Figure 2 shows the overall flow of how MINORIS works. Specifically, for the kernel module to be fuzzed, our system extracts it from the kernel source tree and compiles it as a separate compilation module, which involves the automatic process of recursively extracting the dependent header files from the kernel source code (❶). We then link it with the LKL and LibHW (a hardware emulation library provided by our system, detailed in Section III-C) to generate an executable (❷). After that, it will be loaded and fuzzed as a normal user program. During this process, the KMUT interacts with LKL for needed kernel functions (❸) and libHW for the underlying hardware environment (❹). Noted that MINORIS fuzzing eliminates both kinds of context switches mentioned in the introduction. The fuzzer and LKL merely call the LKL (❸) and LibHW (❹) functions directly, eliminating the overhead of *kernel-emulator context switch* and *user-kernel context switch*.

The idea of *building and fuzzing a kernel module as a user-space program* looks straightforward. However, it faces several technical challenges, including the *symbol dependency* during compilation, and the *state dependency* and *hardware dependency* at runtime, respectively. First, when compiling and linking a kernel module into a user-space program, the dependent symbols (e.g., macros, definitions of data structures, and global variables) and inconsistency between the linked user-space kernel library (LKL) and the original kernel source code pose a serious challenge. Second, when running KMUT in user space, the kernel module states (initialized by the Linux kernel) and the necessary user-space states (created by the init process and other

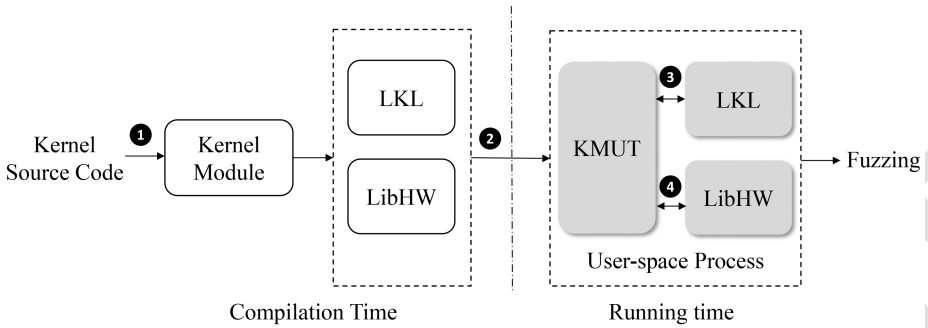


Fig. 2. The overview of MINORIS.

system daemons) are missing. This may cause the KMUT to be inaccessible from LKL. Third, a kernel module usually depends on the underlying hardware to function properly. The lack of underlying hardware will greatly impact the code coverage of KMUT and the effectiveness of fuzzing.

MINORIS proposes corresponding solutions. First, MINORIS automatically analyzes the kernel module to add the necessary symbol declarations or definitions. It also resolves the conflicts between LKL and the original kernel source code to link the kernel module with LKL (and LibHW). Second, we retrieve the initialization functions of the KMUT and make LKL invoke them to prepare the proper kernel module states. Besides, the needed system calls to prepare the user-space states are identified and issued. Third, MINORIS redirects the interaction between the KMUT and the underlying hardware to a user-space hardware emulation library (LibHW). We will elaborate them in Section III-A, Section III-B, and Section III-C in the following.

A. Symbol Dependency

1) Problem Statement: The root cause of the symbol dependencies is *the original kernel source code that the kernel module depends on is different from the user-space kernel library LKL.* Vendor customization, mainline kernel differences, or architecture differences can all introduce differences. Fig. 3 shows four examples of symbol dependency issues. First, an SoC vendor may add header files with SoC-specific declarations missing in LKL (E1). Second, the vendor may modify declarations like adding `dev_socdata` to struct device, incompatible with LKL (E2). Third, since LKL aligns with the mainline kernel at major versions, incompatible declarations (**E3**) could be caused when the kernel module relies on the minor versions of the mainline kernel. Fourth, architecture-dependent macros like `rmb` need to be replaced when extracting ARM modules to an x86-64 host (E4).

To better illustrate the problem, we model the process of compilation and linking the kernel module with the dependent symbols. Specifically, during the compilation, the compiler sees every symbol reference $s =< \mathsf{name}_s, \mathsf{decl}_s, \mathsf{ref}_s >$ or symbol definition $d =< \mathsf{name}_d, \mathsf{decl}_d, \mathsf{def}_d >$. When some symbol reference ref_s is used in the source file, the compiler will generate the instructions and memory allocations according to the declaration decl_s, and the linker will link the reference ref_s to the definition def_s. The compiler and the linker identify a unique symbol entity

E1: header missing

```
#include <soc/samsung/exynos-pd.h>
```

E2: vendor added field

```
struct device {
+  struct dev_socdata   socdata;
   struct fwnode_handle *fwnode;
};
```

E3: shifted enum constant definition

```
enum { // PM_QOS_RESERVED
   RESERVED, CPU_DMA_LAT, NETWORK_LAT,
+  /* 11 enums */, BUS_THROUGHPUT_MAX,
   NETWORK_THROUGHPUT, /* 23 enums */,
   NUM_CLASSES,
}
```

E4: architectural specific macro

```
#define rmb() dsb()
```

Fig. 3. Four concrete examples of unresolved symbol dependencies.

$e \in \{s, d\}$ by its name name_e. We assumes that a kernel module S has the property named *compilation correctness* in its original kernel O, which says that $\forall s \in S, \forall d \in D_s(O \cup S) = \{d \in O \cup S \mid \mathsf{name}_s = \mathsf{name}_d\}, |\mathsf{def}_{D_s(O \cup S)}| = 1 \wedge \mathsf{decl}_s = \mathsf{decl}_d$.

Based on the above model, we accordingly summarize the possible symbol dependencies, when compiling the kernel module with LKL T, as the following three categories.

Case 1: $\exists s \in S, |\mathsf{def}_{D_s(T \cup S)}| = 0$. The compiler complains an *undefined reference* of the symbol. It possibly happens when the dependent modules are not compiled into the T.

Case 2: $\exists s \in S, |\mathsf{def}_{D_s(T \cup S)}| > 1$. The compiler complains about *multiple definitions* for a symbol. This happens when the symbol occurs in both the module S and LKL T.

Case 3: $\exists s \in S, d \in D_s(T \cup S), |\mathsf{def}_{D_s(T \cup S)}| = 1 \wedge \mathsf{decl}_s \neq \mathsf{decl}_d$. The linker can link ref_s to the def_d by name_s, but runtime errors will be raised, such as segmentation fault, due to *inconsistent declaration*.

Our experience when developing MINORIS shows that symbol dependencies are common. Specifically, nearly all the modules contain undefined symbols (Case 1) when extracting kernel modules from AArch64 5.1, Android 5.4. And there are

Algorithm 1: Transform Algorithm.

1: **procedure** Transform
2: input: S represents the kernel module. O represents the original kernel, while T represents LKL.
3: output: adjusted LKL T', adjusted kernel module S'', and header files H used by S''.
4: expand macros in S by T as S'
5: expand rest undefined macros in S' by O as S''
6: **for** $s \in S''$ **do**
7: **if** s is intercepted by fuzzing developers **then**
8: pass
9: **else if** $|\mathsf{def}_{D_s(T \cup S'')}| = 0$ **then**
10: emit a stub definition of decl_s to H
11: **else if** $|\mathsf{def}_{D_s(T \cup S'')}| > 1$ **then**
12: $R \leftarrow R \cup \mathsf{def}_{D_s(T)}$
13: **else if** $\mathsf{decl}_s \neq \mathsf{decl}_d$ for $d \in D_s(T \cup S'')$ **then**
14: **try**
15: replace decl_s with decl_d and compile S''
16: **catch** CompileError caused by replacement
17: patch T to adapt $decl_s$
18: **end try**
19: **end if**
20: **end for**
21: rename R in T to avoid name conflict with S''
22: $(H, T') \leftarrow (H \cup \mathsf{decl}_T, T)$
23: **end procedure**

hundreds of inconsistent symbols (Case 2 and Case 3) when extracting kernel modules from kernels of different architectures or kernel configurations.

2) Our Solution: We propose SemaLinker to solve the symbol dependencies automatically in two steps. First, the symbol semantics of the kernel module, the original kernel, and LKL are collected, respectively. Then, with complete symbol semantics, SemaLinker transforms the kernel module into the KMUT and adjusts LKL accordingly to ensure the KMUT has *compilation correctness* property maintained in the adjusted LKL. We show these two steps in the following.

Step I. Collect Symbol Semantics: SemaLinker runs a collecting procedure on the parsed AST with semantics produced by Clang. It performs a reachability symbol analysis and records the semantic information of symbols, i.e., symbol reference s and symbol definition d, simultaneously. The collected semantics cover several major categories associated with SemaLinker, including macro, function, type, and variable symbols representing preprocessing, control flow, memory protocols, and the state of a program.

SemaLinker checks semantic information at the AST level instead of the IR level because Clang does not emit correct IR if there are syntax or semantic errors in the source files. SemaLinker can handle or ignore errors produced by the Clang parser and analyze the kernel modules from the original kernel, even if the AST contains errors.

Step II. Transform Kernel Modules: Our transform algorithm is shown in Algorithm 1. The algorithm aims to solve three illustrated possible symbol dependencies by taking the kernel module, the original kernel, and LKL with their associated symbol semantics extracted in previous step as inputs. As a result, the adjusted LKL T', adjusted kernel module S'', and header files H containing symbol declarations used by S'', are generated as outputs. The combination of S'' and H forms the KMUT, which has compilation correctness property preserved in the adjusted LKL T'.

First, SemaLinker expands the macros in S. It first expands S by the macro definitions from T (Line 4in Algorithm 1). This solves the issues caused by macros in Case 2 and Case 3. It is correct since the macro definition in the T usually has the same functionality as that defined in the original kernel. SemaLinker then expands the rest of undefined macros by O (Line 5in Algorithm 1), solving Case 1. Our macro expansion strategy is crucial when transforming kernel modules from an ARM kernel, which usually contains architecture-dependent macros.

Then, SemaLinker iterates all symbol references in the adjusted kernel module S'' and solves the symbol dependencies issue when encountered. For an undefined reference (Case 1), SemaLinker generates a stub definition based on the declaration from O to compile the KMUTs (Line 9-10in Algorithm 1). This strategy is previously discussed in EASIER paper [24]. As for Case 2, this happens when the symbol definition occurs in both S and T. At this time, we would like to make the kernel module use its symbol definition. Specifically, SemaLinker collects duplicated definitions in T (Line 12in Algorithm 1) and then renames them to avoid name conflicts with S'' (Line 21in Algorithm 1). Moreover, their corresponding references in T itself are also renamed. When the decl_s of a symbol reference in S'' conflicts with the decl_d of the corresponding symbol definition in LKL (Case 3), SemaLinker prefers to replace decl_s with decl_d at first. If the replacement causes compilation errors when compiling the kernel module with the declarations decl_d, SemaLinker reverts the replacement and tries to patch LKL to adapt decl_s instead. For example, when the vendor adds a new field to the `struct device` (example E2), the issue with Case 3 occurs. If the kernel module does not use the field, the compilation should succeed, and the decl_d is used. Otherwise, the error hints that the kernel module depends on the added field, hence we should add the missing field to LKL by patching the source code of LKL. Since replacing decl_d likely works, this helps stabilize the LKL runtime by avoiding unnecessary patches.

Discussion: The design of Algorithm 1 is based on the assumption that S has compilation correctness property in O. Our algorithm believes that the conflict symbols keep similar semantics so that automatic transformation becomes possible. If the kernel versions differ a lot, the algorithm may fail since there are too many inconsistent declarations. One reason that we chose LKL as our user-space kernel implementation is that LKL is aligned with the mainline kernel at major versions. We can always choose the LKL that is closest to target Linux kernel version.

B. State Dependency

With SemaLinker, the KMUT is ready for linking with LKL. However, to ensure the KMUT is accessible from system calls issued by the fuzzer at runtime, the kernel module states

(initialized by the Linux kernel) and the necessary user-space states (created by the init process and other system daemons) should be prepared before fuzzing. In this section, we illustrate how MINORIS solves the state dependency issue.

Kernel Module State Preparation: In the Linux kernel, kernel modules (including loadable modules) can be directly built into the kernel. If the kernel module is built into the kernel, the initialization vectors (i.e., functions) of the kernel module will be invoked for kernel module state initialization during kernel booting (i.e., `start_kernel`). We observe that the Linux kernel provides a mechanism, named `initcalls`, to allow the kernel module to register its initialization vectors to different boot stages by annotating its initialization vectors. Based on this observation, during compilation, MINORIS automatically identifies the initialization vectors of the KMUT in the source files and inserts them into the corresponding boot stage of LKL. Then, at runtime, the KMUT's initialization vectors are invoked with appropriate timing during the LKL boot process. As a result, the kernel module states of the KMUT are set up properly.

User-space State Preparation: Due to the limited user-space component support of LKL (such as the lack of system daemons), the necessary user-space states of the KMUT are missing. Therefore, MINORIS prepares the necessary user-space states before fuzzing. For example, `mknod` syscalls should be emitted to create the device file entries in the filesystem to run device driver fuzzing, which is originally issued by the user space daemon `udevd`, which does not exist in LKL. Another example is a few virtual network devices that must be created by syscalls to run fuzzing on the network subsystem. For this, we manually write the program to issue the required syscall sequences.

C. Hardware Dependency

At runtime, the KMUT (mainly the device driver) usually depends on the bidirectional interaction with the underlying hardware to function properly. Specifically, the KMUT may need to perform IO access to the device, while the device side could trigger a hardware interrupt targeting the KMUT. However, there is no hardware backend provided by LKL, which will greatly hinder the execution of the KMUT, thus affecting the effectiveness of fuzzing. To solve the above hardware dependency issue, MINORIS integrates a user-space hardware emulation library (LibHW) with the LKL and KMUT to provide a hardware backend. Furthermore, a KMUT-agnostic design is adopted by MINORIS to bridge the bidirectional interaction between the KMUT and the corresponding backend device in LibHW seamlessly. Lastly, the design of linking the kernel (i.e., LKL) and the hardware emulation (i.e., LibHW) together brings additional optimization opportunities for hardware access to achieve better performance and analysis robustness.

LibHW: LibHW is constructed on top of QEMU's hardware emulation framework. Specifically, we customize QEMU to serve as a user-space library and could be invoked for hardware device (including interrupt controller) emulation while leaving QEMU's CPU and memory emulation unused. By doing so, we can utilize its rich virtual device assets and mature framework for adding new virtual devices. In addition, there have been solutions proposed for real device integration [25] and automated virtual device generation [14], [26] for QEMU. They can also be integrated to provide richer hardware device support for LibHW.

Moreover, LibHW leverages device tree configuration to tell the kernel (LKL and the KMUT) how to access the emulated hardware resources. Specifically, at runtime, LibHW is invoked to initialize its hardware resources with the specified hardware configuration (including the devices that the KMUT depends on). A device tree configuration describing the emulated hardware resources by LibHW is generated after the initialization of LibHW. Then, the device tree configuration is passed to LKL, so that LKL can boot with the correct and consistent hardware resource description with LibHW. Since the absence of the bootloader in LKL, we patch LKL with the device tree configuration (i.e., `.dtb`) loading support. For example, for the PCI device, a customized PCI controller kernel module is also added to LKL to support the PCI bus scanning.

Routing the IO Requests from the KMUT: Since the Linux kernel provides a well-defined abstraction for the KMUT to perform IO access, MINORIS is able to intercept IO requests from the KMUT via the well-defined abstraction and redirect them to LibHW.

- *MMIO&PIO:* The `readX`/`writeX` ($X \in \{b, w, l\}$) and `inX`/`outX` ($X \in \{b, w, l\}$) functions are provided by Linux kernel for the KMUT to perform MMIO and PIO operations, respectively. By hooking these low-level IO access functions, MINORIS could capture MMIO/PIO operations issued by the KMUT at runtime and then redirect them to LibHW.
- *DMA:* Regardless of the variations of the kernel DMA APIs, they are eventually transformed into `map_page`, `unmap_page` to access the mapped virtual memory. When MINORIS captures the invocation of `map_page`, it will instruct LibHW to insert the corresponding memory region into the address space emulated by QEMU (i.e., LibHW) and assign a higher priority. In this way, whenever the KMUT accesses the virtual memory or the backend device accesses the emulated address space, they are both reading or writing the same part of physical memory in the host machine. When `unmap_page` is invoked, MINORIS performs an inverse operation of `map_page`. MINORIS removes the memory region for DMA from the emulated address space. Additionally, a pool of memory regions is managed by MINORIS for higher performance on small page mapping.

Routing Interrupts from the Emulated Hardware: We handle interrupts delivered from the LibHW simply and effectively. When delivering an interrupt, LibHW will invoke the interrupt handler inside the LKL directly (since they are inside the same process). This eliminates the dependency on the involvement of CPU emulation for interrupt delivery. We made minor modifications to the interrupt signal-raising logic of QEMU to achieve this, so that our modification is KMUT-agnostic. It's a one-time effort for each system architecture. Correspondingly, LKL is patched with the interrupt controller driver for each architecture to translate architecture-specific hardware interrupt resources to the interrupt abstraction of the Linux kernel.

Algorithm 2: Divirtualized Hardware Access.

```
1: procedure DivirtualizedMMIOReadForE1000
2:   input: addr is the captured input arguments of readX.
3:   output: the return value of readX.
4:   get last accessed MMIO Region M
5:   if M is valid and addr is in range of M then
6:     if QEMUTypeOf(M.parent) = "e1000" then
7:       return e1000_mmio_read(M.parent, addr)
8:     else
9:       return M.mmioRead(addr)
10:    end if
11:  end if
12:  M' ← addressSpaceTranslate(addr)
13:  set accessed MMIO Region M'
14:  return M'.mmioRead(addr)
15: end procedure
```

Divirtualized Hardware Access: Inspired by the divirtualized C++ virtual function dispatching, MINORIS applies an optimization named *divirtualized hardware access* to route the MMIO&PIO requests. We illustrate the divirtualized `readX` for e1000 KMUT by Algorithm 2. When the e1000 KMUT invokes `readX`, MINORIS captures the input arguments and the physical address to access. At first, no memory region is accessed, so MINORIS translates the physical address into the memory region by the address space of QEMU and calls the MMIO handler stored in the memory region. The translation result is saved in a global cache for KMUT, e.g., a thread-local variable. Then, when the KMUT re-accesses the same MMIO region, MINORIS can validate that the physical address located is in the saved memory region and serves the `readX` request in the fast path. Specifically, if `readX` is called in the E1000 driver, the `readX` is specialized to check the type of region's parent to be an E1000 device and call function `e1000_mmio_read` statically if possible.

The *divirtualized hardware access* can also benefit the dynamic analysis for the KMUT. Since the LKL and the LibHW are linked together in the same process, the underlying emulation code for `readX` is in the same process. When routing the `readX` in a fast path, the dynamic analysis can reach the underlying emulation code within a function call. For instance, symbolic execution can benefit from it. We believe that the combination of kernel and hardware emulation in the same process brings additional optimization opportunities.

IV. IMPLEMENTATION

We have implemented a prototype of MINORIS. Specifically, SemaLinker is based on Clang libTooling [30] (15,826 LoC). The fuzzer is based on mutator from syzkaller and libprotobuf-mutator [2], [31] (27,992 LoC). LibOS is based on LKL [32] (200 LoC), and LibHW is based on QEMU [27] (701 LoC). For simplicity, all the primary components of MINORIS work in a single process on the host machine.

A. MINORIS

SemaLinker: SemaLinker implements an analysis crossing different translation units of the kernel module symbols with the help of Clang LibTooling. A translation unit is the root AST, corresponding to an exact source file of the analyzed kernel module. SemaLinker computes a closure on the reachability among the symbols within a single translation unit. Then, SemaLinker classifies whether a declaration or definition of the symbol is required by completing the source file compilation and saves the information as needed.

When the kernel module compilation involves multiple translation units, it is necessary to merge identical symbols and their dependencies in different translation units. Here, SemaLinker implements a string-based digest algorithm to merge the symbols.

LibOS: We implemented a kernel runtime for KMUTs based on LKL. In particular, we integrate some device drivers into LKL, such as the PCI host controller driver for the PCI bus and the ARM GIC driver for ARM Interrupts.

The PCI host controller is implemented based on the generic PCI controllers. It first reads device tree to get the shared MMIO region and hardware IRQ number. It then scans the bus and makes PCI drivers probe the PCI devices correctly.

LibHW: LibHW is a library built on QEMU to provide hardware APIs for KMUTs. LibHW utilizes QEMU's virtual device framework while bypassing its CPU and RAM emulation. First, LibHW accepts the command line options and builds virtual devices according to options by the QEMU framework. Then, to leave CPU and RAM emulation unused, we replace the thread function of QEMU CPU threads with a dummy function so that the CPU threads are idle during fuzzing. As for the QEMU IO thread, because we choose to invoke the LibHW directly as function calls and QEMU CPU threads keep idle, the IO thread is also idle during fuzzing since there is no incoming request for it.

B. Fuzzer

We implemented a user-space kernel module fuzzer based on MINORIS, integrating off-the-shelf fuzzing techniques. The fuzzer employs libFuzzer as the fuzzing engine and syzkaller's mutator for mutation. We use ASAN and Sanitizer Coverage to instrument both KMUTs and LKL for sanitization and coverage collection. To handle LKL's slab allocator, we patched ASAN to support its custom memory management. Furthermore, SymCC [29], a symbolic execution technique, is leveraged by us to assist the fuzzer in reaching a deeper program state, which cannot be easily applied to the OS kernel drivers without our system. Finally, the fuzzer is linked with the KMUT, LKL, and LibHW to run as a user-space program.

V. EVALUATION

In this section, we comprehensively evaluate both the effectiveness of automatically transforming kernel modules into user-space programs and the performance improvement of our framework compared to the state-of-the-art virtualization-based (KVM) kernel fuzzer, syzkaller. All experiments are conducted

TABLE I
THE STATISTICS OF TRANSFORMING KERNEL MODULES UNDER TESTING (KMUTS) BY SEMALINKER. COLUMNS SUMMARIZE STATISTICS ABOUT THE KMUTS COMPILED UNDER ALLYESCONFIG. THE COLUMN "COMPILED KMUTS" SHOWS THE PERCENTAGE OF KMUTS SUCCESSFULLY COMPILED ACROSS ALL ATTEMPTED KMUTS (THE "TOTAL" COLUMN), AND THE PERCENTAGE IN PARENTHESES REPRESENTS THE DIFFERENCE BETWEEN THE NUMBER IN THIS ROW AND THAT IN CORRESPONDING LKL ROW. †: LKL DOES NOT COMPILE UNDER ALLYESCONFIG.

Version	Source Tree	Arch	Number of KMUTs		Compiled KMUTs
			Count	Total	
5.1	Mainline	LKL	-†	9594	65.968%
	Mainline	x86-64	8937		83.771% **(+17.803%)**
	Mainline	AArch64	8684		82.354% **(+16.385%)**
5.4	Mainline	LKL	-	9968	75.251%
	Mainline	x86-64	9273		84.520% **(+9.270%)**
	Android	AArch64	9045		82.253% **(+7.002%)**
5.10	Mainline	LKL	-	10178	77.697%
	Mainline	x86-64	10178		86.933% **(+9.236%)**
5.15	Mainline	LKL	-	10608	78.271%
	Mainline	x86-64	10608		83.522% **(+5.251%)**
6.1	Mainline	LKL	-	11674	76.555%
	Mainline	x86-64	11674		81.960% **(+5.405%)**
6.6	Mainline	LKL	-	11859	74.011%
	Mainline	x86-64	11859		80.546% **(+6.535%)**
6.12	Mainline	LKL	-	12413	75.638%
	Mainline	x86-64	12413		80.649% **(+5.011%)**

on an Ubuntu 22.04 system with an Intel(R) Core(TM) i7-8700 6-core CPU @ 3.20 GHz and 32 GB RAM. Our evaluation addresses the following research questions:

- **RQ1**: Can SemaLinker successfully compile kernel modules as native programs? (Section V-A)
- **RQ2**: Does our system improve the execution speed of Kernel Modules Under Testing (KMUTs) over emulator-based kernel fuzzing? (Section V-B)
- **RQ3**: How does our system's fuzzing performance compare to syzkaller? (Section V-C)

V.-A. *Module Transformation and Compilation (RQ1)*

Experiment Setup: To evaluate the scalability of MINORIS (**RQ1**), we assess SemaLinker (Algorithm 1) by compiling KMUTs with LKL across kernel versions and architectures.

The algorithm requires three inputs: the original kernel O, the target kernel T, and a symbol set S from O. For O, we select mainline x86-64, mainline AArch64, common Android, and LKL kernels, with versions 5.1, 5.4, 5.10, 5.15, 6.1, 6.6, and 6.12—all recent LTS releases. This covers both x86-64 and ARM architectures and includes Android and LKL distributions. For T, SemaLinker uses mainline LKL matching O's major version, as discussed in Section III-A. For S, source files are grouped into KMUTs using `CONFIG` variables, then a symbol set S is extracted for each KMUT.

Additionally, we compile kernels using *allyesconfig* and retain only KMUTs that successfully compiled in the original kernel as the theoretical upper bound of KMUTS to compile. We choose *allyesconfig* because Linux experts use it for regression compilation testing. Table I normalizes these module counts per kernel version using the union set. For example, in mainline v5.1, of 11,483 identified modules, 8,937 KMUTs targeting x86-64 (T_x), and 8,684 KMUTs targeting AArch64 (T_a) are compiled under *allyesconfig*. The union $|T_x \cup T_a| = 9,594$ represents our estimated KMUTs to evaluate SemaLinker for this kernel version.

Finally, Algorithm 1 is executed on each KMUT to determine compilation success.

Transformation Results: Table I presents the transformation results, demonstrating SemaLinker's scalability by compiling kernel modules into user-space programs with substantial success rates. The LKL rows, representing the case of $O = T$ (i.e., trivial transformations), serve as a baseline. For instance, the LKL v5.1 row shows that adjusting `CONFIG` variables per LKL's *defconfig* enables compilation of 65.968% of 9,594 KMUTs. Across LTS versions (v5.4–v6.12), SemaLinker maintains robust performance, consistently achieving over 74.011% success rates and confirming reliability for intra-distro transformations.

The rest rows reveal SemaLinker's value in enabling testing of x86 and ARM kernel modules with LKL. For example, when transforming x86-64 modules from mainline v5.1, SemaLinker achieves 83.771% compilability, improving 17.803% over baseline. That is because SemaLinker resolves additional symbol dependencies, such as undefined references, using information available only in the original kernel. Similarly, SemaLinker compiles 82.354% AArch64 modules from mainline v5.1.

SemaLinker can also handle non-mainline distributions, demonstrated by its 82.253% success rate with Android AArch64 modules from mainline v5.4. This capability is crucial for testing modified or out-of-tree KMUTs, such as those found in Android kernels but absent in the mainline.

Failure Analysis: While SemaLinker achieves substantial transformation success, it does not achieve high bound due to the complexity of Linux's million-line codebase. Two primary failure categories are identified. First, 34.032% of modules from LKL v5.1 fail to compile, due to missing symbol dependencies beyond those included in LKL's *defconfig*. Noted that LKL is not compiled under *allyesconfig*.

Second, approximately 20% of x86-64 and AArch64 modules fail due to SemaLinker-specific issues, primarily involving symbol resolution and code generation. Using x86-64 v6.1 kernel as a representative case (Table II), the most common failure (59.572%, or 10.747% of all modules) comes from unresolved symbols or undefined references during transformation or compilation. Another 22.518% result from malformed C source files generated by transformation, often caused by unhandled corner cases in the complex ordering of declarations, definitions, and macros. Symbol conflicts between kernel and user-space environments contribute 7.743% of failures, while missing source files (3.705%) and miscellaneous issues (6.461%) constitute the

TABLE II
SUMMARY OF FAILURE REASONS OF MODULE TRANSFORMATION BY SEMALINKER

Reason	Count	Percentage
Missing Symbols	1254	59.572%
Malformed Code Generation	474	22.518%
Conflict Symbols	163	7.743%
Others	136	6.461%
Missing Files	78	3.705%

TABLE III
EXECUTION SPEEDUP ACHIEVED WITH LIBHW, WHICH ELIMINATES KERNEL-EMULATOR CONTEXT SWITCHES. KERNEL DRIVERS PERFORM READ (R) OR WRITE (W) OPERATIONS ON HARDWARE ADDRESSES (`hw_addr`) OR C EXPRESSIONS. RESULTS SHOW THE MEAN AND STANDARD DEVIATION OF KERNEL-SIDE I/O REQUESTS PER SECOND OVER 1,000 EXECUTIONS.

Benchmark	Kernel-Side I/O Requests per Second		
	QEMU/KVM	MINORIS	↑
NVMe (hw_addr 0)	w:28.57k ± 7.10%	w:132.38k ± 7.28%	×4.68 ± 0.67
NVMe (hw_addr 4)	w:58.94k ± 6.49%	w:158.25k ± 7.30%	×2.71 ± 0.37
E1000 (hw_addr 0)	r:50.83k ± 12.73%	r:608.69k ± 10.36%	×12.26 ± 2.79
E1000 (hw_addr 8)	r:16.71k ± 24.85%	r:243.53k ± 7.03%	×14.90 ± 4.67
E1000 (hw_addr 32)	r:30.53k ± 18.77% w:28.69k ± 18.45%	r:550.37k ± 6.10% w:297.07k ± 10.62%	×18.30 ± 4.50 ×10.68 ± 3.04
virtio-blk (notify)	w:12.67k ± 6.98%	w:99.53k ± 9.09%	×7.97 ± 1.27
xHCI (db_addr)	w:38.33k ± 2.15%	w:67.96k ± 3.86%	×1.78 ± 0.11
xHCI (op_regs status)	r:66.78k ± 3.45% w:103.66k ± 2.36%	r:902.79k ± 10.43% w:909.26k ± 6.50%	×13.72 ± 1.90 ×8.82 ± 0.78
FTGMAC100 (hw_addr 24)	w:57.43k ± 14.20%	w:209.40k ± 23.52%	×3.99 ± 1.46
Geometric Mean (read)	-	-	×15.13 ± 1.11
Geometric Mean (write)	-	-	×4.95 ± 1.09
Geometric Mean	-	-	×7.38 ± 1.51

TABLE IV
EXECUTION SPEEDUP ACHIEVED WITH LKL, WHICH ELIMINATES USER-KERNEL CONTEXT SWITCHES. BENCHMARK PROGRAMS PERFORM VARIOUS MEANINGFUL SYSCALLS. RESULTS SHOW THE MEAN AND STANDARD DEVIATION OF SYSCALL EXECUTION RATES PER SECOND OVER 1,000 ITERATIONS.

Benchmark	Syscalls per Second		
	QEMU/KVM	MINORIS	↑
access01 (faccessat)	21.28k ± 2.65%	54.69k ± 10.03%	×2.60 ± 0.33
add_key01 (add_key)	88.49k ± 1.48%	126.11k ± 4.58%	×1.43 ± 0.09
adjtimex01 (adjtimex)	314.55k ± 5.68‰	418.40k ± 4.46%	×1.33 ± 0.07
bind01 (bind)	594.23k ± 1.70%	1164.75k ± 3.26%	×1.96 ± 0.10
capset01 (capset)	614.35k ± 1.96%	1647.99k ± 3.75%	×2.69 ± 0.15
dup01 (dup)	622.41k ± 1.20%	5619.28k ± 15.69%	×9.27 ± 1.56
fcntl01 (fcntl)	643.58k ± 8.35‰	5791.10k ± 2.82%	×9.01 ± 0.33
getsockopt01 (getsockopt)	411.08k ± 1.79%	3110.36k ± 8.32%	×7.63 ± 0.77
ioctl_loop01 (ioctl)	262.86 ± 4.46%	2.99k ± 10.17%	×11.81 ± 1.72
keyctl01 (keyctl)	544.46k ± 7.71‰	1753.00k ± 6.32%	×3.23 ± 0.23
listen01 (listen)	670.17k ± 1.56%	7610.56k ± 7.56%	×11.43 ± 1.04
mmap02 (mmap)	227.57k ± 2.12%	518.21k ± 5.26%	×2.29 ± 0.17
nice01 (setpriority)	503.55k ± 6.49‰	2315.38k ± 5.97%	×4.62 ± 0.31
openat02 (openat)	15.11k ± 13.42%	43.69k ± 22.14%	×3.13 ± 1.08
pipe01 (pipe2)	353.76k ± 1.51%	252.04k ± 2.34%	×0.71 ± 0.03
read04 (read)	553.75k ± 1.79%	651.05k ± 23.44%	×1.25 ± 0.31
timer_create01 (timer_create)	283.22k ± 59.65%	708.07k ± 19.51%	×2.90 ± 2.06
ulimit01 (setrlimit)	765.03k ± 1.20%	14292.29k ± 10.29%	×18.91 ± 2.17
vmsplice01 (vmsplice)	246.45k ± 2.13%	352.87k ± 1.56%	×1.43 ± 0.05
write01 (write)	113.82k ± 3.25‰	166.93k ± 1.41%	×1.47 ± 0.03
Geometric Mean	–	–	×**3.31** ± **0.67**

remainder. These failures highlight opportunities for improving symbol analysis and handling of edge cases in future work.

Answer to RQ1 SemaLinker automates the transformation of kernel modules across different versions and architectures, resolving symbol dependencies and linking them with LKL to produce user-space programs for the host machine.

V.-B. Execution Performance (RQ2)

Experiment Setup: To assess the framework's execution performance (**RQ2**), we compare the execution speed of KMUTs from the mainline x86-64 6.6 kernel with the same code executed in QEMU/KVM. The comparison is conducted at two levels: syscall (evaluating the benefit of using LKL) and hardware (evaluating the benefit of using LibHW).

For hardware-level benchmarking, we test E1000 Ethernet and NVMe block driver KMUTs using LibHW (QEMU backend) with workloads from the `libnvme` test suite [33] and E1000 self-test programs. Additional tests for virtio-blk, xHCI, and FTGMAC100 KMUTs are written using the LTP framework. For syscall-level benchmarking, we select 20 syscall test programs from the LTP project (listed in Table IV).

To ensure fair comparison, noise from time sampling and compilers is minimized. Each benchmark program is executed 1,000 times. Kernel code for both MINORIS and QEMU is compiled with `clang-12` using consistent kernel configurations to avoid differences in compilation flags. Benchmarks are executed with CPU resources limited to a single core.

Results: The execution performance comparison between MINORIS and QEMU/KVM is presented in Table III (hardware-level) and Table IV (syscall-level). Program behaviors have been verified to be consistent across both environments, indicating that KMUTs can be run on MINORIS preserving fidelity. From tables, MINORIS shows consistent performance improvements with low variance, as results were gathered over 1,000 iterations using precise hardware timers, e.g., RDTSC, to measure execution cycles.

At the hardware level, MINORIS achieves an average speedup of ×7.38 (geometric mean) with LibHW. Read and write operations to MMIO regions are improved by ×15.15 and ×4.95, respectively. For instance, writes to NVMe hardware addresses achieve speedups of ×4.68 and ×2.71 at addresses 0 and 4, respectively. Furthermore, the FTGMAC100 driver, originally executed in the QEMU ARM emulator, can be compiled into LKL and executed on an x86-64 host, demonstrating MINORIS's versatility. Performance gains are also attributed to LibHW optimizations such as Divirtualized Hardware Access, which caches memory mappings and directly invokes corresponding C functions for devices. These improvements confirm that MINORIS delivers notable hardware-level acceleration.

At the syscall level, MINORIS achieves an average speedup of ×3.31 with LKL. 19 out of 20 syscall benchmarks exhibit improved performance compared to QEMU/KVM. The median speedup, driven by the `mmap` syscall, is ×2.29, with larger gains observed for syscalls such as `ioctl` (loop) and `ulimit`, which achieve speedups of ×11.81 and ×18.91, respectively. Some syscalls, however, show limited or negative performance impacts. For example, `pipe` performs worse in MINORIS due to cache-issuing memory access to the `set_freepointer` function when allocating SLUB objects in LKL. This can be improved from the LKL side in the future. Additionally, memory-intensive syscalls like `add_key`, `read`, and `vmsplice` show marginal speedups as their performance is limited by heavy memory operations. In such cases, KVM mitigates the overhead of memory simulation through shadow page mapping, reducing the relative advantage of MINORIS.

Answer to RQ2 Our system can achieve an average speed up ×3.31 and ×7.38 with LKL and LibHW, eliminating the *kernel-emulator context switch* and *user-kernel context switches*.

C. Fuzzing Performance

Experiment Setup: To demonstrate the fuzzing performance of MINORIS (**RQ3**), we compare it against syzkaller using 5 fuzzing targets: E1000, Netlink, VTY, NVMe, and Cadence GEM. The KMUTs are instrumented. The kernel image used by syzkaller

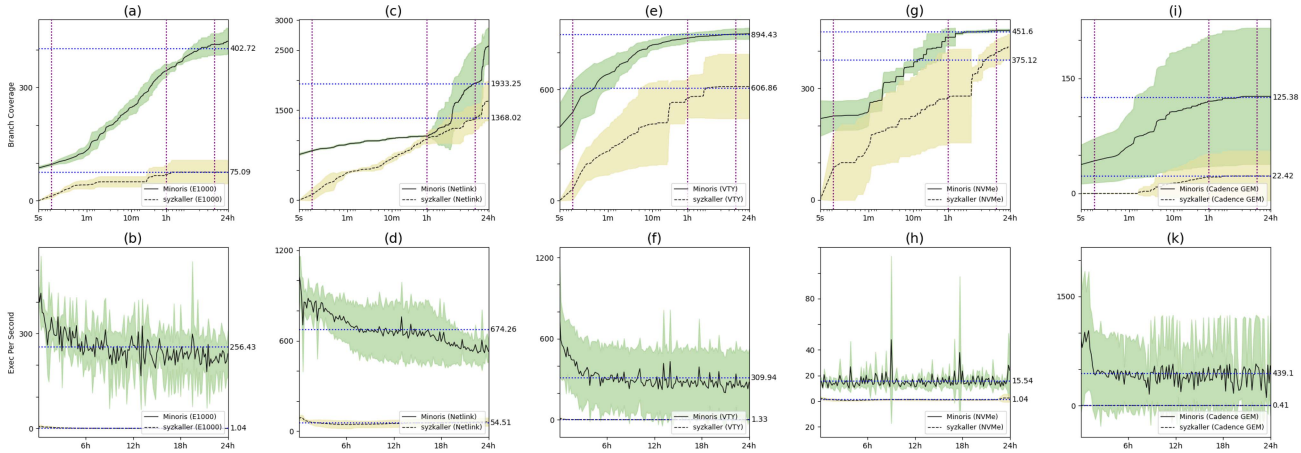


Fig. 4. Branch coverage and fuzzing throughput for five kernel modules under test (KMUTs). Subfigure (a) presents the branch coverage of E1000, and subfigure (b) shows its fuzzing throughput. The remaining subfigures (c)–(k) display the branch coverage and fuzzing throughput for the other four KMUTs: Netlink, VTY, NVMe, and Cadence GEM, respectively. The shaded areas around the lines represent the standard deviation across ten repeated evaluations per target.

is instrumented with KASAN and Kcov. The KMUTs and LKL are instrumented by ASAN and Sanitize Coverage. Then, we deploy the fuzzer applications for the KMUTs in combination with other user space testing techniques, SymCC. All the tests are executed continuously for 24 hours without seed corpora. Both systems are given the same hardware resources, 2 CPU cores and 4 GB memory. Each target is run 10 times repeatedly.

In addition to considering KMUT execution, we also eliminate the differences in the coverage collecting methods for the fuzzing benchmark. Since syzkaller utilizes Kcov and the libFuzzer used by MINORIS utilizes Sanitize Coverage, we collect the corpus during fuzzing periodically and run the corpus on Sanitize Coverage instrumented KMUTs after fuzzing to get final basic block coverage on the code.

Throughput and Coverage: Fig. 4 shows that MINORIS outperforms syzkaller in terms of throughput and code coverage. The average fuzzing branch coverage of MINORIS outperforms syzkaller with QEMU/KVM with the increase of 89.51% , as shown in the first row of Figure 4. The reason is the improvement in execution speed (as shown in RQ2). The current evaluation uses the same fuzzing algorithm as syzkaller. Our system may use more advanced user-space fuzzing techniques (one benefit of MINORIS is integrating user-space fuzzing techniques easily) to further the performance. During the experiment, syzkaller is stuck on the E1000 device driver. The reason is that the network functionality of the testing kernel is broken, but the syzkaller executor uses it to communicate with the fuzzer out of the emulator. We do not fix it and treat the problem as a performance issue of syzkaller, since the fix to avoid some syscall inputs may make fuzzing never touch the related coverage.

The average fuzzing throughput of MINORIS beats syzkaller persuasively with an increase of $\times$102.58, as shown in the second row of Figure 4. The execution per second is KMUT-sensitive. The throughput of Netlink fuzzing has an increase of $\times$12.37. In particular, syzkaller performs 54.51 executions per second, and MINORIS performs 674.26 executions per second.

Stability: The fuzzing adopted by MINORIS did not crash by moving KMUT from kernel to user space, proving the stability of KMUTs. In particular, the MINORIS is even more stable than syzkaller. This is based on the observation that syzkaller's executor did not respond to the fuzzer during the experiment on the E1000 target. We learn that the syscall execution may corrupt the network channel to the fuzzer, which means the fuzzer itself can be affected by the fuzzing target. MINORIS fuzzer itself does not rely on KMUT or LibOS, which helps MINORIS avoid the limitation.

Detected Bugs: MINORIS found 24 bugs during the continuous fuzzing in months, and five of them are new bugs, as shown in Table V. The bugs can be reproduced in the original kernel, and the community has confirmed new bugs.

The bug located in `delete_char` is found to cause use after free, stack buffer overflow, and double free. This proves that our system can leverage *user-space memory sanitizer* ASAN to find `kernel` memory bugs. Another interesting bug is the presence of a UAF in `route4_change`, where routing rules that have been removed remain in the global hash table. Triggering the bug requires creating a large number of system resources through syscalls. Our system can detect such a bug with the assistance of user-space symbolic execution [29].

Answer to RQ3 Our system performs better than syzkaller in fuzzing throughput ($\times$102.58), explores 89.51% more branches, and detects 24 bugs. Among the found bugs, 6 are new bugs.

VI. DISCUSSION

Binary-level SemaLinker: We have implemented SemaLinker at the source-code level, hence it cannot analyze the closed-source loadable kernel modules. The binary-level SemaLinker must overcome challenges about how to identify and patch instructions that access missing hardware or kernel objects inconsistent with LKL. Possible solutions include pattern-based identification of target instructions.

Limitation of LKL: LKL does not have MMU and SMP. It means that the related code cannot be tested currently, such as components directly related to virtual address translation and concurrent execution of the kernel. To solve this limitation, LKL could leverage LibHW to support MMU and reimplement thread scheduling to support SMP in future.

TABLE V
BUGS DETECTED BY MINORIS

Location	Function	Type	New bug
drivers/base/devtmpfs.c	devtmpfs_create_node	Use after free	yes/confirmed/merged
drivers/clk/clk-ast2600.c	aspeed_g6_clk_probe	Use before Initialization	yes
drivers/gpu/arm/mali_kbase.c	set_schedule_time	Divison by zero	yes
drivers/tty/vt/vt.c	delete_char	Data Corruption	yes/confirmed/merged
include/asm-generic/uaccess.h	__strncpy_from_user	Stack buffer overflow	yes/confirmed
net/sched/cls_route.c	route4_change	Double free/Use after free	yes/confirmed
drivers/nvme/host.c	nvme_cancel_request	Null pointer dereference	no
drivers/nvme/host/nvme.h	nvme_trace_bio_complete	Null pointer dereference	no
net/core/net_namespace.c	get_net_ns	Null pointer dereference	no
net/netfilter/ip_set_core.c	ip_set_utest	Null pointer dereference	no
net/netfilter/nft_exthdr.c	nft_exthdr_ipv4_eval	Heap Buffer Overflow	no
net/netfilter/nft_limit.c	nft_limit_init	Integer Overflow	no
net/netfilter/nf_tables_api.c	nft_dump_basechain_hook	Null pointer dereference	no
net/netfilter/nf_tables_api.c	nft_table_validate	Dead Lock	no
net/netfilter/nf_conntrack_netlink.c	ctnetlink_create_conntrack	Memory Leak	no
net/sched/act_ife.c	tcf_ife_init	Null pointer dereference	no
net/sched/cls_api.c	tcf_chain_flush	Infinite loop	no
net/sched/cls_api.c	tcf_exts_destroy	Null pointer dereference	no
net/sched/cls_route.c	route4_bind_class	Use after free	no
net/sched/cls_tcindex.c	tcindex_partial_destroy_work	Out of memory	no
net/sched/cls_tcindex.c	tcindex_set_parms	Out of bound write	no
net/sched/sch_api.c	tc_bind_tclass	Null pointer dereference	no
net/sched/sch_dsmark.c	dsmark_init	Null pointer dereference	no
net/sched/sch_generic.c	qdisc_put	Null pointer dereference	no

Extending Hardware Runtime: MINORIS can support more backends for LibHW, such as real hardware [25] and symbolic hardware modules [14], [26]. With these backends, more code related to specific hardware can be fuzzed by MINORIS. These efforts are orthogonal to MINORIS and can be ported to our framework.

VII. RELATED WORK

*Fuzzing Systems and Frameworks:*Fuzzing is a general testing technique and has identified thousands of zero-day bugs in real-world programs, leveraging fuzzer engines [34], [35], [36] and fuzzing platforms with continuous integration [37], [38], [39]. Zhu et al. [40] conducted an extensive study of fuzzing techniques, providing a roadmap for research in the field. Fuzzing combines with many analysis techniques to improve performance and effectiveness. Instrumentation techniques guide fuzzing, which leverages static analysis results and dynamic behavior detection to collect execution metrics [41], [42], [43] and create oracles [28], [44], [45], [46]. Moreover, fuzzers are often combined with symbolic execution to create general-purpose hybrid fuzzing systems [29], [47], [48], [49], [50].

High Performance OS Kernel Fuzzing: Several approaches have been proposed to improve the performance of OS kernel fuzzing. Agamotto utilizes virtual machine checkpoints to eliminate duplicated executions, but still results in expensive memory consumption with the inevitable overhead of checkpoints [23]. EASIER [24] speed up kernel fuzzing by running a virtual machine snapshot containing the testing module in the dUnicorn, a lightweight CPU emulator. As a result, it outperforms traditional kernel fuzzing using the QEMU emulator. However, it does not solve the hardware dependency of testing modules. Additionally, setting up snapshot introduces overhead.

Solving Hardware Dependency: The kernel modules may be directly or indirectly coupled with some hardware, causing hardware dependency. And it is an open research question to solve the hardware dependency of testing modules. Feng et al. [51] provide a systematic analysis of the challenges and solutions posed by hardware dependencies, particularly in firmware testing.

Some systems redirect hardware requests to real hardware. Charm creates stub devices in QEMU to forward MMIO and interrupt requests between the device driver and real hardware [6]. However, manual efforts are required to transplant device drivers to the hosted kernel in the emulator. It does not support DMA operations as well. Similarly, PeriScope proxies IO requests by page faults [25]. This requires additional overhead to switch the interrupt states.

Other approaches replace certain kernel layers with emulated implementations. Some works adopt automated solutions such as symbolic execution to simulate virtual devices [14], [26], [52], [53]. The hardware models learned by symbolic execution can improve the diversity of virtual devices. However, they are not equivalent to the real implementation and produce false positives.

VIII. CONCLUSION

We propose a new kernel fuzzing framework that can fuzz kernel modules as normal user-space programs to gain nearly native execution performance. We solved a couple of challenges and implemented a prototype named MINORIS. The evaluation shows that compared with the state-of-the-art system, MINORIS can achieve an average execution speedup from $\times$3.31 to $\times$7.38, and improves the fuzzing throughput ($\times$102.58). It can explore more code coverage and detect new bugs.

REFERENCES

[1] A. Danial, "CLOC: Count lines of code," 2009. [Online]. Available: http://cloc.sourceforge.net/
[2] Google, "SYZKaller: SYZKaller is an unsupervised coverage-guided kernel fuzzer," 2015. [Online]. Available: https://github.com/google/syzkaller/

[3] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "{kAFL}:{Hardware-Assisted}feedback fuzzing for {OS} kernels," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 167–182.

[4] D. Jones, "Triforce Linux syscall fuzzer," 2017. [Online]. Available: https://github.com/nccgroup/TriforceAFL/

[5] J. Corina et al., "DIFUZE: Interface aware fuzzing for kernel drivers," in *Proc. 2017 ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 2123–2138.

[6] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian, "Charm: Facilitating dynamic analysis of device drivers of mobile systems," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 291–307.

[7] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, "Fuzzing file systems via two-dimensional input space exploration," in *Proc. 2019 IEEE Symp. Secur. Privacy*, 2019, pp. 818–834.

[8] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim, "Finding semantic bugs in file systems with an extensible fuzzing framework," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 147–161.

[9] H. Peng and M. Payer, "USBFuzz: A framework for fuzzing {USB} drivers by device emulation," in *Proc. 29th {USENIX} Secur. Symp.*, 2020, pp. 2559–2575.

[10] Y.-H. Zou, J.-J. Bai, J. Zhou, J. Tan, C. Qin, and S.-M. Hu, "{TCP-Fuzz}: Detecting memory and semantic bugs in {TCP} stacks with fuzzing," in *Proc. 2021 USENIX Annu. Tech. Conf.*, 2021, pp. 489–502.

[11] E. J. Crowley, G. Gray, and A. J. Storkey, "Moonshine: Distilling with cheap convolutions," in *Proc. 32nd Int. Conf. Neural Inf. Process. Syst.*, 2018, pp. 2893–2903.

[12] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "HFL: Hybrid fuzzing on the Linux kernel," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020.

[13] S. Y. Kim et al., "{CAB-Fuzz}: Practical concolic testing techniques for {COTS} operating systems," in *Proc. 2017 USENIX Annu. Tech. Conf.*, 2017, pp. 689–701.

[14] W. Zhao, K. Lu, Q. Wu, and Y. Qi, "Semantic-informed driver fuzzing without both the hardware devices and the emulators," in *Proc. Netw. Distrib. Syst. Secur. Symp.*2022.

[15] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "RAZZER: Finding kernel race bugs through fuzzing," in *Proc. 2019 IEEE Symp. Secur. Privacy*, 2019, pp. 754–768.

[16] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "KRACE: Data race fuzzing for kernel file systems," in *Proc. 2020 IEEE Symp. Secur. Privacy*, 2020, pp. 1643–1660.

[17] K. Lu, M.-T. Walter, D. Pfaff, S. Nümberger, W. Lee, and M. Backes, "Unleashing use-before-initialization vulnerabilities in the Linux kernel using targeted stack spraying," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017.

[18] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "{FUZE}: Towards facilitating exploit generation for kernel {Use-After-Free} vulnerabilities," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 781–797.

[19] Y. Chen and X. Xing, "SLAKE: Facilitating slab manipulation for exploiting vulnerabilities in the Linux kernel," in *Proc. 2019 ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 1707–1722.

[20] W. Chen, X. Zou, G. Li, and Z. Qian, "{KOOBE}: Towards facilitating exploit generation of kernel {Out-of-Bounds} write vulnerabilities," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 1093–1110.

[21] X. Zou, G. Li, W. Chen, H. Zhang, and Z. Qian, "SYZScope: Revealing high-risk security impacts of fuzzer-exposed bugs in Linux kernel," in *Proc. 31st USENIX Secur. Symp.*, 2022, pp. 3201–3217.

[22] D. Jones, "Trinity: Linux system call fuzzer," 2011. [Online]. Available: https://github.com/kernelslacker/trinity/

[23] D. Song, F. Hetzelt, J. Kim, B. B. Kang, J.-P. Seifert, and M. Franz, "Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 2541–2557.

[24] I. Pustogarov, Q. Wu, and D. Lie, "Ex-vivo dynamic analysis framework for android device drivers," in *Proc. 2020 IEEE Symp. Secur. Privacy*, 2020, pp. 1088–1105.

[25] D. Song et al., "Periscope: An effective probing and fuzzing framework for the hardware-os boundary," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2019.

[26] Z. Ma et al., "PrintFuzz: Fuzzing Linux drivers via automated virtual device simulation," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2022, pp. 404–416.

[27] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX Annu. Tech. Conf., FREENIX Track*, 2005, vol. 41, pp. 10–55.

[28] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "{AddressSanitizer}: A fast address sanity checker," in *Proc. 2012 USENIX Annu. Tech. Conf.*, 2012, pp. 309–318.

[29] S. Poeplau and A. Francillon, "Symbolic execution with {SymCC}: Don't interpret, compile!," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 181–198.

[30] L. Developers, "LibTooling - LibTooling is a library to support writing standalone tools based on clang," 2024. [Online]. Available: https://clang.llvm.org/docs/LibTooling.html

[31] Google, "LibProtobuf-Mutator: Library for structured fuzzing with protobuffers," 2022. [Online]. Available: https://github.com/google/libprotobuf-mutator

[32] O. Purdila, L. A. Grijincu, and N. Tapus, "LKL: The Linux kernel library," in *Proc. 9th RoEduNet IEEE Int. Conf.*, 2010, pp. 328–333.

[33] M. Belanger, "LibNVME: C library for NVM express on Linux," 2024. [Online]. Available: https://github.com/linux-nvme/libnvme/

[34] K. Serebryany, "LibFuzzer - Library for coverage-guided fuzz testing," 2024. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[35] M. Zalewski, "American fuzzy lop," 2020. [Online]. Available: https://lcamtuf.coredump.cx/afl/

[36] R. Swiecki, "HonggFuzz," 2022. [Online]. Available: https://honggfuzz.dev/

[37] K. Serebryany, *OSS-Fuzz - Google's Continuous Fuzzing Service for Open Source Software*. Vancouver, BC, Canada: USENIX Assoc., Aug. 2017.

[38] A. Arya, O. Chang, M. Moroz, M. Barbella, and J. Metzman, "Open sourcing clusterfuzz," Google, Inc., Feb., 2019, Accessed: Oct. 5, 2025. [Online]. Available: https://opensource.googleblog.com/2019/02/open-sourcing-clusterfuzz.html

[39] Microsoft, "OneFuzz: A self-hosted fuzzing-as-a-service platform." 2023. [Online]. Available: https://github.com/microsoft/onefuzz/

[40] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A survey for roadmap," *ACM Comput. Surv.*, vol. 54, no. 11s, pp. 1–36, 2022.

[41] L. Developers, "Clang 19 documentation - Sanitizercoverage," 2024. [Online]. Available: https://clang.llvm.org/docs/SanitizerCoverage.html

[42] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "{ParmeSan}: Sanitizer-guided greybox fuzzing," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 2289–2306.

[43] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, "{APISan}: Sanitizing {API} usages through semantic {Cross-Checking}," in *Proc. 25th USENIX Secur. Symp.*, 2016, pp. 363–378.

[44] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data race detection in practice," in *Proc. Workshop Binary Instrum. Appl.*, 2009, pp. 62–71.

[45] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu, "Context-sensitive and directional concurrency fuzzing for data-race detection," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2022.

[46] L. Developers, "UBSAN: Undefinedbehaviorsanitizer is a fast undefined behavior detector," 2024. [Online]. Available: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[47] C. Cadar et al., "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Symp. Operating Syst. Des. Implementation*, 2008, pp. 209–224.

[48] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Proc. 16th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2011, pp. 265–278.

[49] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 745–761.

[50] J. Chen et al., "{SYMSAN}: Time and space efficient concolic execution via dynamic data-flow analysis," in *Proc. 31st USENIX Secur. Symp.*, 2022, pp. 2531–2548.

[51] X. Feng, X. Zhu, Q.-L. Han, W. Zhou, S. Wen, and Y. Xiang, "Detecting vulnerability on IoT device firmware: A survey," *IEEE/CAA J. Automatica Sinica*, vol. 10, no. 1, pp. 25–41, Jan., 2023.

[52] M. J. Renzelmann, A. Kadav, and M. M. Swift, "{SymDrive}: Testing drivers without devices," in *Proc. 10th USENIX Symp. Operating Syst. Des. Implementation*, 2012, pp. 279–292.

[53] K. Cong, F. Xie, and L. Lei, "Symbolic execution of virtual devices," in *Proc. 13th Int. Conf. Qual. Softw.*, 2013, pp. 1–10.

Yangxi Xiang is currently working toward the PhD degree with the College of Computer Science and Technology, Zhejiang University, and interning with a leading blockchain company, BlockSec. His research interests mainly focus on system security, smart contract security, and DeFi security.

Feng Wang received the bachelor's degree in network engineering from the Nanjing University of Post and Telecommunication, Nanjing, China, in 2016, and the master's degree from the University of Chinese Academy of Sciences from ShanghaiTech University, in 2019, under the supervision of Prof. Fu Song. He is now working with Skyward Lab of Ant Group.

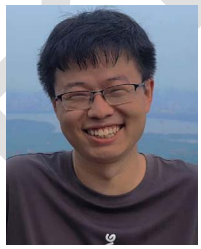
Yuan Chen received the bachelor's degree in computer science and technology from Zhejiang University, in 2019, and the PhD degree in cyberspace security from Zhejiang University, in 2024. His research interests focus on confidential computing, system security, and blockchain security.

Qiang Liu (Member, IEEE) received the PhD degree from Zhejiang University (ZJU), in 2023, under the guidance of Prof. Yajin Zhou. He is a postdoc with EPFL, working with Prof. Mathias Payer with the HexHive Laboratory. His research in cybersecurity focuses on 1) developing prior-to/after-release security enforcement of software based on deep understanding, and 2) building the chain of trust examined by full-chain exploits. His work has been recognized with all the top security conferences: IEEE S&P, Usenix Security, ACM CCS, and ISOC NDSS. He received the Best Paper Awards with USENIX Security'24 and ACM RAID'24. He is also serving on the program committee for IEEE/ACM ASE'25 and USENIX Security'25, and is a reviewer for *ACM Computing Surveys* and *ACM Transactions on Software Engineering and Methodology*.

Haoyu Wang received the PhD degree in computer science from Peking University, in 2016. He is a professor with the School of Cyber Science and Engineering, Huazhong University of Science and Technology. His research covers a wide range of topics in software analysis, privacy and security, eCrime, internet/system measurement, and AI security. More information is available at: https://howiepku.github.io/

Jiashui Wang is the head of Ant Security Skyward Lab and the main founder of Ant Security Light-Year Lab.

Lei Wu received the PhD degree from North Carolina State University, in 2015. He is an associate professor with Zhejiang University and the co-founder of BlockSec, and primarily focuses his research on system security and blockchain security.



Yaoguang Chen is a senior security expert with Ant Group, specializing in cybersecurity defense, mobile security, and AI security. He has extensive practical experience in cybersecurity defense, having discovered hundreds of security vulnerabilities and obtained more than 50 CVE numbers. He has also collaborated with universities to co-author several highquality academic papers.

Yajin Zhou is a professor with Zhejiang University. His research focuses on system security, blockchain security, and cybercrime. He has published more than 50 papers in top-tier venues with more than 10,000 citations and was recognized by AMiner as one of the Most Influential Scholars. He served on the program committees of major security conferences such as ACM CCS, IEEE S&P, and USENIX Security.