USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

SoK: Automating Kernel Vulnerability Discovery and Exploit Generation

Anil Kurmus, Andrea Mambretti, and Alessandro Sorniotti, *IBM Research Europe – Zurich;* Vincent Lenders, Damian Pfammatter, and Bernhard Tellenbach, *armasuisse – Cyber-Defence Campus*

https://www.usenix.org/conference/woot25/presentation/kurmus

This paper is included in the Proceedings of the
19th USENIX WOOT Conference on Offensive Technologies.

August 11–12, 2025 • Seattle, WA, USA

ISBN 978-1-939133-50-2

Open access to the Proceedings of the
19th USENIX WOOT Conference on Offensive Technologies
is sponsored by USENIX.

SoK: Automating Kernel Vulnerability Discovery and Exploit Generation

Anil Kurmus, Andrea Mambretti, Alessandro Sorniotti
IBM Research Europe – Zürich

Vincent Lenders, Damian Pfammatter, Bernhard Tellenbach
armasuisse – Cyber-Defence Campus

Abstract

Operating systems (OS) underpin modern IT infrastructure from computers, to smartphones and cloud servers. The OS kernels of these systems are central to their security. Yet their inherent complexity results in a broad attack surface and frequent vulnerabilities, often targeted for denial of service, privilege escalation, or information leakage. While static analysis and fuzzing tools can detect defects in OS kernels, distinguishing exploitable vulnerabilities from benign bugs typically requires manual exploit development, a process that remains labor-intensive. Over the past three decades, attackers have increasingly automated parts of this process, culminating in recent advances in automated exploit generation (AEG) powered by program analysis techniques such as symbolic execution. However, applying these techniques to large complex systems such as OS kernels continues to be challenging. This paper sheds light on the main reasons why it remains challenging to automate exploit generation in OS kernels. We systematize the current knowledge of attacks against kernels in categories, going beyond memory corruption attacks, as well as the relevant threat models and tools used. We categorize existing work along this model to show that gaps exist in many areas. Our analysis helps us identify open problems, in particular the lack of reproducibility across different kernel versions due to the large code base and changing APIs which renders comparisons between different papers difficult. Finally, we propose a set of recommendations for future work in this area.

1 Introduction

Bug exploitation is a widespread and effective way for attackers to compromise software systems. By leveraging one or more bugs, attackers are able to circumvent deployed security mechanisms, and obtain unauthorized control over a system. Reliable *exploits* are highly valuable, often traded on the black market due to their potential to covertly compromise high-value targets [72], fueling an entire underground industry focused on vulnerability discovery and weaponization [3, 5, 29]. Widely used operating systems such as Linux and Windows are one of the favorite targets for attackers, due to the high privilege and access these systems have. Operating systems are vulnerable to a variety of vulnerability classes and offer a large attack surface because of their complexity [4, 52].

While some works focus on the discovery and deployment of defense mechanisms, others deal with the offensive side, thereby attempting to understand strategies to find vulnerabilities as well as to exploit them. The rationale of the latter type of effort is that public discourse over attackers' strategies or trends supports the preemptive creation of adequate defenses.

The typical starting point of an offensive security analyst is the discovery of a *bug* – a defect in which a system responds in a way that was neither foreseen nor intended by the developer. Empirical evidence shows that the process of discovering bugs can be made *efficient* through various methods such as static analysis or fuzz testing. Bug discovery may therefore be considered as *cheap* and *fast* – systems such as syzbot [2] can be efficiently instantiated on commodity hardware and run continuously in a daily search for new bugs. This step can be *efficiently automated*, as evidenced once again by the syzkaller architecture which employs at its core syzbot, which is an automated fuzz testing framework for the Linux kernel.

Not all bugs have a security impact, and understanding whether a bug is a vulnerability involves further steps. The next step requires converting this bug into an exploit that forces the system into behaving in a specific, attacker-chosen manner (e.g., elevate the privileges of an unprivileged, attacker-controlled process). This is *hard to automate* and typically requires a high degree of skilled labour to achieve, which makes it an *expensive* and intrinsically *complex* task.

Attempts to achieve *Automated Exploit Generation* (AEG) started with simple and small user-space programs [8, 38, 39, 82]. These techniques are helpful to (i) quickly analyze large numbers of detected bugs; (ii) prioritize fixes for a set of bugs; (iii) remove the exploitation vectors that are easiest to detect, thus increasing the cost of exploitation; (iv) test the effectiveness of new security mechanisms.

Albeit challenging, AEG for the Linux kernel constitutes a very powerful asset, given the importance of this component. Therefore, understanding where we stand as community and which future direction and open problems we need to take into account in researching this area is paramount to progress in the right direction for kernel AEG.

In this paper, we shed light on the main reasons why automating the exploitation of OS kernel bugs remains challenging. We systematize the current knowledge on attacks against OS kernels into categories, going beyond memory corruption attacks, as well as the relevant threat models, and the tools used. We consider all the body of work that directly, or indirectly, has contributed one or more steps towards achieving AEG for the Linux kernel in the past 12 years. We categorize existing work along this model, to show that gaps exist in many areas. Our analysis helps us identify open problems and recommendations: (i) lack of consideration of non-local threat models, (ii) possible improvements to AEG-inspired bug prioritization approaches, (iii) less-explored AEG vulnerability classes, (iv) lack of portability and reproducibility across different kernel versions due to the large code base and changing APIs which renders comparisons between different papers difficult, (v) ethical considerations for AEG research, and (vi) possible improvements to AEG evaluations.

The rest of this paper is organized as follows: In Section 2, we present background and systematize aspects of kernel exploitation that are relevant. In Section 3, we outline the generic steps required to achieve automated kernel exploitation that we use to classify previous work and their relation to automated kernel exploit generation (AKEG). In Section 4, we survey papers in the area of automated kernel exploitation, and more broadly automated vulnerability finding techniques, as well as new exploitation technique for OS kernels. In Section 5, we discuss the gaps in current AKEG research, our recommendations, and possible future directions.

2 Kernel exploitation

Below we introduce definitions that are used throughout this paper, including many definitions commonly used in OS security. We delineate first the system model in which we consider the kernel, then the various attacker (or threat) models within which kernel security is considered, and finally, what the various attacker (and therefore security) goals are.

2.1 System model and definitions

This paper focuses on modern commodity OS kernels and in particular the Linux kernel. This is because Linux is the target of most of the works we surveyed in this area. It is widely used in industry as well as academia as a target for vulnerability discovery and exploitation projects, because it benefits from being open-source, is widely used in practice on mobile or as part of cloud servers, and has a wide range of high quality program analysis and vulnerability discovery tools available, making it easier to build upon them. Nevertheless, much of the research in this area is also applicable to other modern monolithic OS kernels such as Windows, OS X, iOS, or FreeBSD. We describe here a model of the Linux kernel, defining various concepts and components used throughout this paper and in the literature.

Figure 1 depicts major components of the Linux kernel and the entities it interacts with. *User processes* are processes that run in *userspace*, an area of memory that is considered unprivileged by the CPU. This means the CPU restricts access to certain privileged instructions, but also to *kernelspace* memory, dedicated to the kernel. User processes typically interact with the kernel via *system calls*, an API provided by the kernel to user programs and allowing the kernel to enforce isolation and other security policies. Modern OSes are multi-user systems, with isolation between different user programs and some user programs having distinct *privileges*. These privileges can be implemented in a myriad of ways, including via discretionary access control on special files, mandatory access control via LSM security modules that specify applicable security policies, and Linux capabilities that are assigned to certain programs or users, allowing access to certain system calls (or certain features of a system call). In particular, user programs can run *sandboxed*, meaning that they only have access to a restricted set of system calls. The extent of this access depends on the mechanism and policy that is applied in the sandbox. A plethora of such mechanisms exist for Linux, for example seccomp [1], SELinux/AppArmor [69], and language-based isolation solutions such as JavaScript in browsers, or even software-based isolation (SFI) solutions such as NaCL [96]. Another popular way of running user programs is via *containers*. A container uses the kernel's namespacing feature, such that a set of processes (i.e., a container) sees one set of resources only, those in its namespace, thereby providing isolation. The Linux kernel has also adopted non-monolithic features over the years, namely loadable kernel modules (LKMs) and a

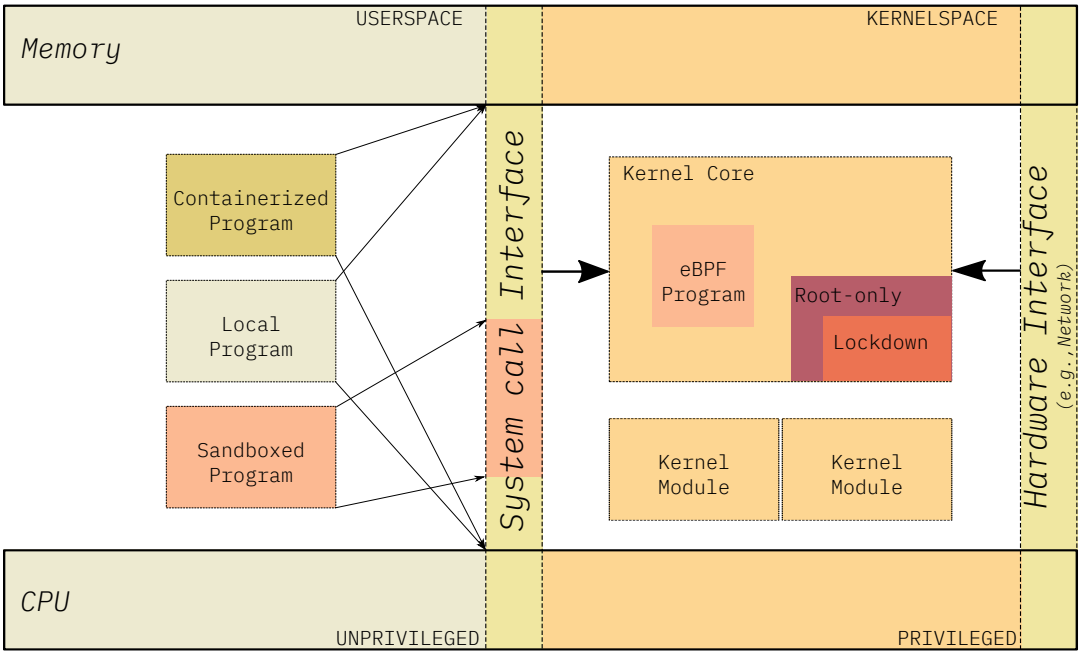

Figure 1: Representation of the Linux kernel components and its security boundaries. The programs running in userspace can be *local*, *sandboxed*, or *containerized*. They interface with the core kernel through the system call interface. These processes can directly access userspace memory, which is segregated from kernelspace memory. Their access of the CPU functionalities is also limited due to the CPU unprivileged mode they are run with. The system call interface is the gateway to access privileged functionalities, and one of the most common ways for attackers to leverage kernel vulnerabilities. The network interface is another important attack surface exposed from the kernel which is used for remote attacks. Similarly, any hardware device that can be under attacker control exposes a new kernel attack surface. Most kernel functionalities are exposed to the regular user through the system call interface, while few are only accessible for the administration user (a.k.a. *root*). Modern kernels also supports a lockdown mode that can restrict the administrator whenever they cannot be fully trusted, for example by removing access to functionalities allowing kernel-modifications like *kexec*.

kernel sandbox (eBPF). LKMs run with the same kernel privileges as the core kernel code, but can be loaded at runtime, possibly on-demand, i.e., when a system call requires it. Loading an arbitrary LKM is always a privileged operation. eBPF code on the other hand runs in a virtual machine, with limited access to the kernel API and features. Arbitrary eBPF code can be loaded by unprivileged users (when the unprivileged eBPF kernel configuration is enabled): a verifier ensures that the eBPF code is well-formed, checking in particular that all accesses are memory-safe, before loading and running it. Finally, the kernel also interacts with hardware devices, typically via interrupts and memory-mapped I/O. In particular, network packets, bluetooth and WiFi packets, are partially processed by the kernel before being delivered either to the device for sending the packet, or the user process responsible with receiving the packet.

Additionally, we use definitions for bugs (or defects), vulnerabilities, primitives, and exploits in line with existing literature [7, 27, 52, 79]. A bug is any deviation from the programmer-intended behavior. A vulnerability is a bug that can manifest a security consequence (in a given attacker model and for a given attacker goal). An exploit is a program that leverages one or more vulnerabilities to achieve an attacker goal. In practice, multiple bugs may need to be combined for an exploit. For example, it is common that a buffer overflow vulnerability may need to be combined with an information leakage vulnerability for an attacker to obtain arbitrary code execution. We therefore also use the term *potential vulnerability* to refer to bugs that may have a security consequence when combined (or by themselves), and further elaborate on such vulnerabilities in Section 3.3. Finally, a primitive is an abstraction corresponding to a functional aspect of an exploit; it corresponds to a common capability or mode of operation attackers will be using across different exploits. For example, attackers refer to a *control-flow hijacking*, an *unlinking* or a *write-what-where* primitive.

2.2 Attacker models

The user-kernel boundary discussed above is a central security boundary in modern operating systems. Kernel code has privileged access to the underlying hardware,

and mediates accesses from userspace mainly via system calls. It thereby ensures isolation and fair sharing of resources between different security domains residing in userspace, traditionally processes running under different user IDs. To bypass this isolation guarantees, userspace attackers may attempt to abuse the user-kernel boundary by typically issuing a sequence of malicious system calls, i.e., a *local exploit*. A pure *remote exploit* abuses external interfaces of the kernel, such as vulnerabilities in network packet parsing. The most commonly used attacker model for kernel exploitation is usually the local attacker model, for local exploits, whereby the attacker has access to unprivileged system calls, with full control over their parameters, in any possible sequence, and with turing complete computation capability in between these system calls. Most papers on kernel exploitation and defenses assume this attacker, either implicitly or explicitly. Modern OSes have however many attacker models that are relevant in practice with respect to the kernel, which we describe here.

- **Local:** Corresponds to an attacker that has access to the system as an unprivileged user and has the ability to interact with the kernel through the system call interface in an unrestricted manner.
- **Local/User sandbox:** A specific case of the local attacker, that corresponds to a sandboxed userspace attacker, i.e., an attacker that has strictly less entry points to the kernel than in the typical local attacker model.
- **Local/Container:** Corresponds to a local attacker in an (unprivileged) container. This is different from the Sandbox attacker, as additional kernel functionality required for containers does open additional kernel attack surface. In particular, namespaces are a Linux kernel feature that allows for partitioning of resources for one or multiple processes. Because such namespacing allows operations traditionally not available to unprivileged users, such as mounting certain filesystems or creating network interfaces, it can open additional kernel attack surface when compared to a local attacker.
- **Remote:** Corresponds to a network attacker with no ability to run code on the target machine. For example: internet attackers that target vulnerabilities in the kernel's TCP/IP stack, LAN attackers that target physical or link layer vulnerabilities, or close proximity attackers targeting WiFi or Bluetooth vulnerabilities.
- **Kernel sandbox (eBPF):** Corresponds to an attacker with the ability to load sandboxed code of their choosing into the kernel. In some configurations, the Linux kernel allows unprivileged users to load eBPF code, opening it to a different attacker model.
- **Lockdown:** Corresponds to a privileged attacker (root) that aims to subvert the kernel and therefore bypass Lockdown [68] restrictions, for example to establish persistence across reboots in the context of Secureboot. Indeed, the Linux kernel provides a Lockdown mode to this effect that introduces a security boundary between root and kernel privileges: under this setting, the administrator (root) is prohibited to execute operations that would subvert the kernel, such as loading kernel modules or executing the `kexec` system call (which allows to load and run a new kernel).
- **Physical:** Corresponds to an attacker with physical access to the target machine, and equipped to perform attacks on the hardware. For example, the attacker may attempt glitching attacks, or use EM-based side channels. We include this attacker model solely for completeness, as the Linux kernel does not attempt to prevent such attacks.

2.3 Attacker goals

The attacker's goal can be formulated as a violation of the kernel's confidentiality, integrity or availability guarantees. This can be via *privilege escalation* (PE), that is obtaining greater privileges than granted, violating confidentiality, integrity and/or availability; *information leakage* (IL), that is, obtaining secret information, violating confidentiality; or *denial of service* (DoS), that is rendering the system temporarily or permanently inoperable, violating availability or fairness.

Many kernel exploits target privilege escalation, which is typically done either by gaining arbitrary code execution in kernel context, or gaining arbitrary read and write capability, allowing to grant the local attacker administrative privileges.

3 End-to-end AKEG

We develop in this section a common language to cluster, compare and identify gaps in existing literature around achieving end-to-end automated kernel exploit generation (AKEG).

3.1 Exploitation steps

We divide as follows the steps an attacker may take in crafting an exploit. The steps correspond well to different stages needed for end-to-end AKEG.

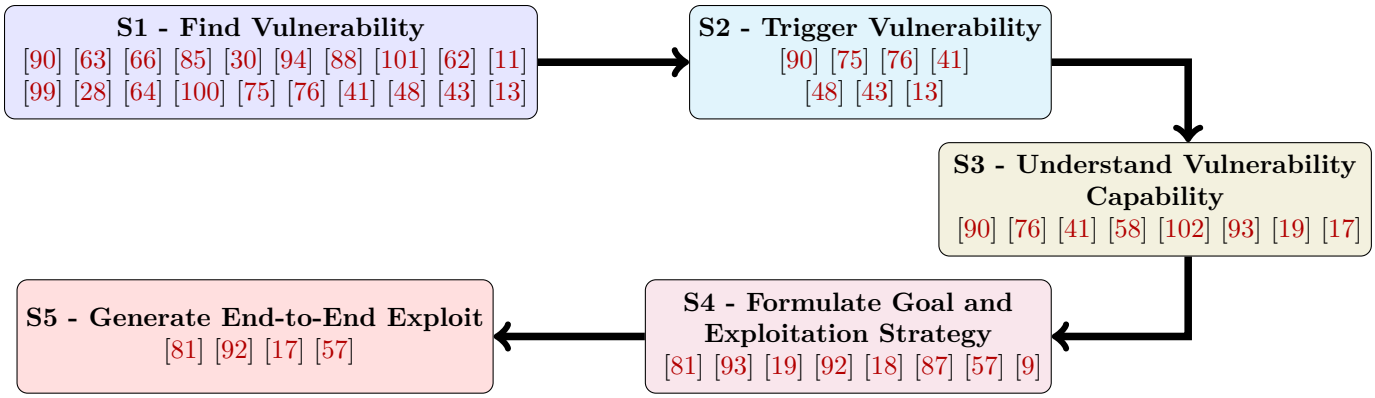

Figure 2: In this figure, we represent the exploitation steps. For each stage, we reference the surveyed papers that worked to automate it.

S_1 **Find vulnerability:** The first step is to find a (potential) vulnerability in the kernel, that is a bug with potential security impact.

S_2 **Trigger vulnerability:** At this step, the attacker finds an input and path starting from a kernel entry point that reaches the vulnerability. The entry points depend on the attacker model, it may for instance be via the system call interface. This, and the previous step, are often achieved via the use of a fuzzer which produces a reproducer capable of triggering the vulnerable code. Static analysis, or manual code analysis, can also be performed to find vulnerabilities, but extra effort must be made to verify the code is reachable by the attacker, and false positives due to the code not being reachable in the relevant threat model are possible.

S_3 **Understand vulnerability capability:** At this step, the attacker infers the full potential of the vulnerability. This means not only the vulnerability class (such as buffer overflow), but also all aspects that are controllable by the attacker (such as how many bytes can be written beyond buffer bounds, and whether those byte values are controlled by the attacker). This is traditionally done manually as it often requires deep understanding of the code base, but sanitizers and symbolic execution engines can automate this process.

S_4 **Formulate goal and exploitation strategy:** At this step, the attacker needs to find an exploitation strategy to achieve the desired goal. This may require using helper exploitation techniques such as heap massaging (e.g., [77, 91]), or bypassing mitigations such as KASLR (e.g., [34]), potentially requiring additional vulnerabilities. In other words, the general idea of this step is to match the vulnerability capability with a target goal, possibly via one or more intermediate steps, which themselves may involve other vulnerabilities.

S_5 **Generate end-to-end exploit:** The last step is to put the previous steps from S_2 to S_4 together in one program, an exploit, that achieves the attacker goal when executed.

In our analysis, we find that all but trivial exploits require these steps for automation. For trivial exploits such as aiming for a local denial-of-service with a memory corruption vulnerability, it would not require the exploitation-related steps S_3, S_4, and S_5 given that the outcome of S_2 may already achieve the attacker's goal (in this case, crashing the kernel). Figure 2 represents the sequence of stages described, and for every step, it reports the corresponding surveyed papers. In Section 4, we provide a detailed description of the paper selection process.

3.2 Defining automation

To classify and survey the work in this area, we define here what is meant by *automation*. This definition depends on each of the S_i steps, and we can represent each by a program taking an input and producing an output. We detail now what these inputs and outputs are.

For S_1, this means a program that takes as input a program (the kernel or part of the kernel), and outputs a set of bugs. For S_2, the input is a bug, and the output a kernel input for the attacker model (e.g., a sequence of system calls in the local attacker model) that triggers the vulnerability when executed. For S_3, taking a vulnerability and trigger as input, we output characteristics of the vulnerability. Characteristics depend on the vulnerability class: for instance, for spatial memory safety vulnerabilities, the attacker would like to know what parts of the input are attacker-controlled, how many

bytes can be written or read out-of-bounds and at which address. For S_4, taking the attacker goal, vulnerability and its characteristics, we output a kernel input (e.g., a sequence of system calls) that either puts the kernel in an exploitable state or produces crucial information needed in an exploit. This step may need to invoke itself S_1-S_5 for another type of vulnerability, for example, for defeating KASLR, which would typically be part of S_4 for a memory corruption vulnerability aiming to gain privilege escalation, the attacker may additionally perform S_1-S_5 to find and exploit a kernel information leak. For S_5, taking all previous outputs, we output a program that achieves the attacker's goal when executed. Note that the input space we specify here for each program is a superset, i.e., automation approaches do not need to handle all possible classes of input at once, but may focus on a non-trivial subset. Similarly, they may fail to produce an output in some cases for those elements in their input set, or their output may contain errors, e.g., the fact that an output is not automatically produced does not imply that it cannot be done by another approach. For example, for static analysis tools (S_1), this remark concerning input and output sets can be reformulated as *soundness*, i.e., the ability of finding a bug if it exists, and *completeness*, i.e., if a bug is identified then it is a real one [73].

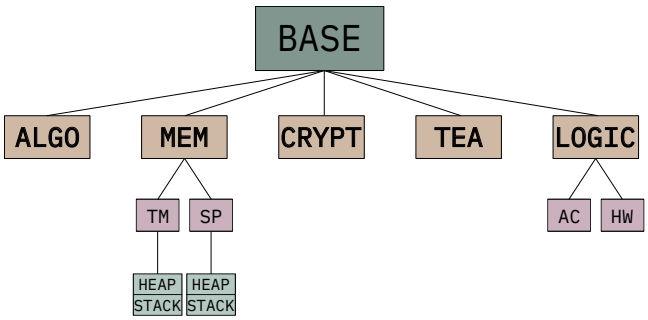

Figure 3: This diagram represents the class of the base vulnerabilities identified for the Linux kernel. A base vulnerability can be caused by an algorithm issue (ALGO), a memory issue (MEM), a crypto issue (CRYPT), a transient execution attack (TEA), or a logic issue (LOGIC). The MEM class can be caused by a temporal (TM) or a spatial (SP) violation, either on the stack or on the heap.

3.3 Vulnerability classes

A 2011 analysis of Linux kernel vulnerabilities shows that a large class of vulnerabilities are memory safety-related [15]. Recent research shows that while this trend remains, there are new related-classes of attacks [12]. We propose three meta-classes of vulnerabilities to combine existing taxonomies for memory safety vulnerabilities as well as other typical vulnerabilities identified in the Linux kernel in a structured way: *base vulnerabilities*, *precursor vulnerabilities* and *helper vulnerabilities*. We make this distinction based on our analysis of vulnerabilities, and rely on two main observations: some classes of vulnerabilities may need to lead to another vulnerability class in order to be exploitable, and some vulnerabilities are not sufficient by themselves for the attacker to reach its goal, but require to be combined with other vulnerabilities.

Base Vulnerabilities Vulnerabilities in this class have the potential to allow the attacker to directly achieve its goal without needing additional vulnerabilities. Although they may still require additional base or helper vulnerabilities in some cases, for example to bypass mitigations, the vulnerability class can be sufficient for the attacker to achieve its goal. In fact, these types of vulnerabilities provide a fundamental primitive to attackers, in the sense that often other vulnerabilities are used to obtain such a primitive. Figure 3 shows the various categories of vulnerabilities in this class. One of the main categories of such vulnerabilities are memory safety vulnerabilities (BASE-MEM), which can lead to privilege escalation, information leakage, or denial of service. The general idea is for the attacker to obtain the capability of reading or writing arbitrary data at arbitrary kernel addresses. These are subdivided into temporal (BASE-MEM-TM) and spatial (BASE-MEM-SP) issues depending on whether a pointer is dangling or out-of-bounds [79], and can also be classified depending on the data structure, e.g., heap or stack vulnerabilities. For example, a prominent class of vulnerabilities often leading to privilege escalation in the Linux kernel are use-after-free vulnerabilities [20], a type of temporal memory safety vulnerability on the kernel heap (BASE-MEM-TM-HEAP). Other categories of base vulnerabilities are algorithmic vulnerabilities (BASE-ALGO) [25], cryptographic and random number generation vulnerabilities (BASE-CRYPT) [49], transient execution attacks (BASE-TEA) [14,43,74,81], and logic vulnerabilities, for instance missing access control checks (BASE-LOGIC-AC) [40] or erroneous use of hardware such as CVE-2018-3665 (BASE-LOGIC-HW).

Precursor Vulnerabilities These potential vulnerabilities may only lead to a security impact via one of the base vulnerabilities, i.e., their impact can be reduced to that of a base vulnerability class. We find this distinction between precursor and base vulnerabilities to be useful: even if techniques for finding such vulnerabilities in code bases may be different (e.g., different static analysis approaches), the analysis of their exploitability can be reduced to that of a base class.

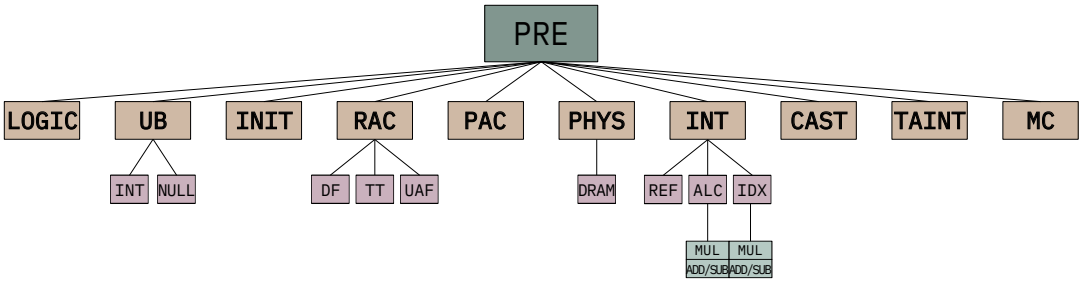

Figure 4: This diagram represents the class of the precursor vulnerabilities. A precursor vulnerability leads to a base vulnerability, and can be a logic issue (LOGIC), undefined behavior (UB), missing initialization (INIT), race condition (RAC), missing pointer access control (PAC), hardware-related (PHYS), integer overflow or underflow (INT), type casting errors (CAST), tainted data (TAINT), or missing return value check (MC).

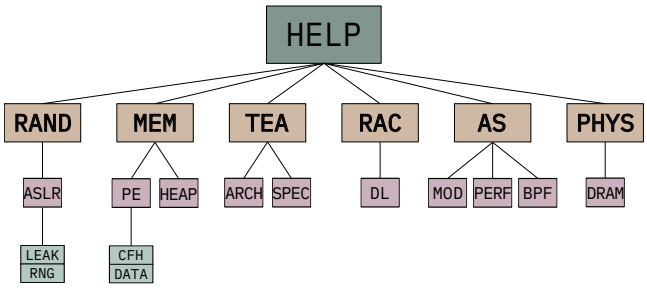

Figure 5: This diagram represents the class of helper vulnerabilities, which are used solely in combination with other vulnerabilities to achieve the attacker's goal. These include attacks on randomization (RAND), privilege escalation techniques (MEM-PE), heap massaging techniques (MEM-HEAP), transient execution attacks (TEA), race conditions exploitation techniques (RAC), attack surface increasing configurations (AS), and hardware attack facilitation techniques (PHYS)

.

Most undefined behaviors in the C programming language fall in this category of potential vulnerabilities. For example, integer overflows (PRE-INT) typically lead to memory safety issues: either via reference counting errors (PRE-INT-REF), leading typically to use-after-free conditions (BASE-MEM-TM), or via out-of-bounds indexing (PRE-INT-IDX), and improper size allocations (PRE-INT-ALC), typically leading to out-of-bounds accesses (BASE-MEM-SP). Another prominent class of precursor vulnerability leading to memory safety issues is missing initialization (PRE-INIT), such as CVE-2010-3296. Signed integer overflow is undefined behavior in the C standard (PRE-UB-INT), and this could lead for instance to the compiler optimising-away important bounds checks, which itself leads to memory safety issues [70]. Similarly, other undefined behavior, such as dereferencing of a NULL pointer (PRE-UB-NULL), could lead to the compiler optimising away crucial checks [23]. Logic precursor vulnerabilities (PRE-LOGIC) broadly include access control and similar issues that may lead to one of the base vulnerabilities, such as CVE-2022-0500. Race conditions can lead to memory safety issues: for instance concurrent use-after-free [80] (PRE-RAC-UAF, and time-of-check-to-time-of-use (PRE-RAC-TT) or double fetch vulnerabilities [76, 85, 86, 94] (PRE-RAC-DF). Pointer access control issues (PRE-PAC), also called user pointer dereference issues, are particularly relevant when passing pointers accross security boundaries, as is the case in the system call interface. Indeed, an attacker can provide a kernel address when a user access is expected, potentially leading to the kernel overwriting its own memory while trying to write to userspace, or conversely leaking memory (BASE-MEM). In fact, such attacks are one of the earliest form of OS kernel memory corruption, known since 1972 [6], but are still relevant today (e.g., CVE-2010-3904, CVE-2010-4258, CVE-2018-20669 and CQUAL [44]). A related issue, type confusion bugs (PRE-CAST), is more generic and also leads to memory safety issue (see for instance Uncontained [51]).

Helper Vulnerabilities These potential vulnerabilities may only lead to an attacker achieving its goal in combination with another vulnerability. For instance bypasses of kernel exploit mitigations fall into this category, such as defeating ASLR (HELP-RAND-ASLR), but also techniques to massage the heap to bring the kernel heap into an exploitable state [20] (HELP-MEM-HEAP), or techniques combining memory safety with speculative execution attacks [31, 67, 74] (HELP-TEA). Techniques to improve reliability of race condition (HELP-RAC) [56, 74], techniques to access additional kernel attack surface (HELP-AS), and techniques to enable attacks via the hardware, such as reverse engineering of DRAM banks necessary for Rowhammer attacks [10, 22] or power measurements via the running average power limit (RAPL) interface for microarchitectural attacks [60].

Note that this meta-class terminology is orthogonal

to that of the commonly used "vulnerability primitive" terminology in prior works: for instance, a control-flow hijacking primitive can be obtained via a number of the base vulnerability classes, but obtaining this primitive may require a helper vulnerability, and using this primitive to achieve privilege escalation may require other helper vulnerabilities as well.

4 Analysis of automation techniques

In this section, we analyze automation techniques pertaining to the exploitation of kernel vulnerabilities in a broad sense, and identify emerging areas and gaps.

4.1 Methodology

Paper selection We select papers published in prominent academic security venues (see Appendix for the detailed selection methodology) that automate some aspect of exploiting vulnerabilities in the Linux kernel. Because exploitation of vulnerabilities is closely related to finding vulnerabilities, we also include relevant papers that automate the task of finding vulnerabilities, but distinguish them explicitly, and do not aim to be exhaustive with papers focusing on finding vulnerabilities. In particular, for static analysis and fuzzing approaches, we refrain from including them unless they specifically propose a new approach to find bugs in OS kernels: a comparison of such tools is outside the scope of this work. In addition, because many manual approaches include some amount of automation, for instance a simple static analysis approach to search for a vulnerable pattern or a particular data structure, we also include papers that do not explicitly focus on automation, but present results on a novel kernel vulnerability class. We refer to the Appendix for additional details on the selection process.

Evaluation criteria We use the various dimensions presented in Section 2 in our analysis, inferring the relevant data when the paper does not clearly state the information, but explicitly stating our inferences. As shown in Table 1, we also note the publication year, whether the tool is open-source if an automation tool is presented, and alternatively whether the attack proof-of-concept is publicly available if the attack is manual. We group papers based on whether they focus on vulnerability finding solely, typically via static analysis (S_1), on vulnerability finding with test case, typically via fuzzing and capability extraction (S_1, S_2, S_3), on automated exploit generation (S_3, S_4, S_5) or show new important attack techniques.

4.2 Analysis

We detail below the results of our analysis, with each paragraph corresponding to a section of Table 1. Table 2 complements this table by showing the vulnerability classes covered by each tool.

Static analysis Static analysis tools mainly target the vulnerability finding phase (S_1). Although the Linux kernel is natively compiled with the GCC compiler, most static analysis tools build on the LLVM compiler framework [53]. This is mainly due to the superior documentation available, as well as the modular architecture of LLVM that makes writing compiler passes significantly easier when compared to GCC. One drawback is that, although great efforts were done to allow the Linux kernel to build with LLVM, only the most recent versions of the kernel code can be compiled with specific versions of LLVM, while older versions cannot be compiled or analyzed easily using it. Beyond that, there may be security-relevant differences between LLVM and GCC builds (for example around undefined behavior [54, 89]). The tools we survey cover a large portion of vulnerability classes, with DR. CHECKER [66] being one of the earliest kernel static analysis papers to cover a wide variety of vulnerability classes, albeit only in kernel drivers due to scaling reasons of interprocedural context-sensitive and flow-sensitive static analysis, which is commonly required to perform accurate static analysis, i.e., close to being sound and complete. Unisan [63] is the first of many checkers detecting uninitialized memory uses [99, 100]. Another class of vulnerabilities well-suited for static analysis are double fetches [85, 94]. Integer overflows and related undefined behaviors are detected by KINT [90].

Fuzzing In recent years, fuzzing tools have been extremely successful in finding bugs, in particular with Syzkaller for the Linux kernel [83]. They are especially useful to developers because they generate test cases, allowing developers to reproduce the defect and debug it (corresponding to S_2). Syzkaller is a coverage-guided, syscall- and mutation-based Linux kernel fuzzer, that found more than 3700 bugs in 3 years [71]. A limitation of Syzkaller is its need for syscall descriptions, that are manually written whenever new features are added, including for each new driver. The lack of good syscall descriptions for a feature will result in low coverage. DIFUZE [24], KSG [78], SyzDescribe [37], and SyzGen++ [16] automate syscall description generation, all using mainly static analysis with the exception of SyzGen++. FuzzNG [13] takes an approach that does not require syscall descriptions, but instead provides specific handling for important kernel mechanisms such as pointer accesses and file descriptor usage. kAFL [75] is

	Work	Year	AM	AG	Steps	Fuzz	San	SA	SMT	DSE	Src	OS
Static Analyzers	KINT [90]	2012	⋆	⋆	S1;S2;S3	–	–	LLVM	Boo	–	✓	Li
	UniSan [63]	2016	⋆	⋆	S1	–	–	LLVM	–	–	–	Li; An
	DR. CHECKER [66]	2017	L	⋆	S1	–	–	LLVM	–	–	✓	Li
	DoubleFetch [85]	2017	L	⋆	S1	–	–	Cocc	–	–	✓	Li; An; Fr
	K-Miner [30]	2018	L	⋆	S1	–	–	LLVM	–	–	–	Li
	Deadline [94]	2018	L	⋆	S1	–	–	LLVM	Z3	–	–	Li; Fr
	LRSan [88]	2018	⋆	⋆	S1	–	–	LLVM	–	–	–	Li
	Pex [101]	2019	L	⋆	S1	–	–	LLVM	–	–	–	Li
	CRIX [62]	2019	⋆	⋆	S1	–	–	LLVM	–	–	–	Li
	KALD [11]	2019	L	⋆	S1	–	–	LLVM	–	–	–	Li
	UBITect [99]	2020	⋆	⋆	S1	–	–	LLVM	–	KLEE	✓	Li
	K-MELD [28]	2021	⋆	DoS	S1	–	–	LLVM	–	–	–	Li
	Goshawk [64]	2022	⋆	⋆	S1	–	–	CSA	Z3	–	✓	Li; Fr
	IncreLux [100]	2022	⋆	⋆	S1	–	–	LLVM	–	KLEE	✓	Li
Fuzzers	kAFL [75]	2017	L	⋆	S1;S2	Own	KA/⋆	–	–	–	✓	Li; Wi; Ma
	DECAF [76]	2018	L	⋆	S1;S2;S3	T; N	–	–	–	–	–	Li;Wi
	RAZZER [41]	2019	L	⋆	S1;S2;S3	Syz	KA/ld/⋆	LLVM	–	–	–	Li
	HFL [48]	2020	L	⋆	S1;S2	Syz	–/⋆	L+G	–	S2E	✓	Li
	GREBE [58]	2022	⋆	⋆	S3	Syz	–	LLVM	–	–	✓	Li
	SyzScope [102]	2022	⋆	⋆	S3	Syz	KA	LLVM	–	S+a	✓	Li
	Kasper [43]	2022	L	IL	S1;S2	Syz	DTA	LLVM	–	–	✓	Li
	FuzzNG [13]	2023	⋆	⋆	S1;S2	Own	–	–	–	–	✓	Li
AEG	Drammer [81]	2016	L	PE	S4; S5	–	–	–	–	–	✓	An
	FUZE [93]	2018	L	⋆	S3;S4	Syz	KA	–	✓	angr	≈	Li
	SLAKE [19]	2019	L	PE	S3;S4	Syz; M	–	L+G	–	–	✓	Li
	KEPLER [92]	2019	L	PE	S4;S5	–	–	Own	–	angr	✓	Li
	KOOBE [17]	2020	L	PE	S3;S5	Syz	KA	–	Kqu	S2E	✓	Li
	ELOISE [18]	2020	L	IL	S4	–	–	LLVM	Z3	–	–	Li; Fr; X
	AlphaEXP [87]	2023	L	PE/IL	S4	Syz	–	KINT;Sf	–	–	–	Li
	K-LEAK [57]	2024	L	IL	S4;S5	Syz	KA	LLVM	–	angr	✓	Li
	SCAVY [9]	2024	L	PE	S4	Syz	KA	LLVM	–	–	✓	Li
Exp. Tech.	Ret2dir [46]	2014	L	PE		–	–	–	–	–	✓	Li
	Prefetch [33]	2016	L	IL		–	–	–	–	–	✓	Li; Wi
	PT-Rand [26]	2017	L	PE/IL		–	–	–	–	–	–	Li; Wi
	Blindside [31]	2020	L	IL		–	–	–	–	–	✓	Li
	ExpRace [56]	2021	L	PE/IL		–	–	–	–	–	✓≈	Li
	Pspray [55]	2023	L	⋆		–	–	–	–	–	✓	Li
	RetSpill [98]	2023	L	PE/IL		–	–	–	–	angr	✓	Li
	SLUBStick [65]	2024	L	PE/IL		–	–	–	–	–	✓	Li
	GhostRace [74]	2024	L	IL		–	–	Cocc	–	–	✓	Li
	CARDSHARK [36]	2024	L	any		–	–	–	–	–	✓	Li

Table 1: Summary table of the survey. Columns: AM: attacker model, AG: attacker goal, Steps: Automated steps, Fuzz: Fuzzer, San: Sanitizer, SA: static analysis, SMT: SMT solver, DSE: dynamic symbolic execution, Src: source code available. Other abbreviations: PE: privilege escalation, IL: information leakage, DoS: denial of service, L: local, ⋆: any, ✓: yes, –: no, ≈: partial, Syz: syzkaller, M: Moonshine, T: Trinity, N: NtCall64 fuzzer, KA: Kernel Address Sanitizer, ld: lockdep race detection, Cocc: coccinelle, L+G: LLVM and GCC, Sf: Soufflé, S+a: S2E and angr, DTA: custom dynamic taint analysis (under San due to space constraints), Kqu: Kquery, Li: Linux, An: Android, Wi: Windows, Ma: MacOS, X: XNU, Fr: FreeBSD.

in comparison to Syzkaller OS-agnostic, works mostly at the hypervisor-level, and uses Intel PT for code coverage, but is limited in the achieved coverage as it uses mostly file-based inputs with a mutation engine similar to AFL. To extend coverage beyond branches with hard-constraints, HFL [48] combines fuzzing with symbolic execution. RAZZER [41] and DECAF [76] specialize in fuzzing concurrency bugs. SyzScope [102] and GREBE [58] both focus on S_3: starting from a fuzzer-found bug, GREBE uses directed fuzzing and taint analysis to find other paths leading to the same bug, in an effort to uncover other potentially more security-relevant behaviors, as reported by a sanitizer, of the same bug. SyzScope additionally guides symbolic execution to evaluate exploitability.

Attack techniques As a response to the introduction of several kernel mitigations, multiple papers contain novel attack techniques, with the goal of demonstrating the limitations of these mitigations as well as the security-relevance of a new vulnerability class. Similarly to user-space attacks, modern OS kernels and architectures prevent execution from user pages while in kernel mode (SMEP on Intel CPUs) as well as direct dereference of user pointers while in kernel mode (SMAP on Intel). Bypassing these mitigations, ret2dir [47] is a technique to redirect either code execution or data pointer accesses into a portion of kernel memory that directly maps user pages into the kernel. PT-Rand [26] demonstrates a particular data-only attack vector (HELP-MEM) that is common to all OSes: modifying page table information to achieve privilege escalation or leaking data. ExpRace [56] presents a technique to help with the exploitation of concurrent data races, in particular to be able to synchronize attack and victim threads. Vulnerabilities on the kernel heap (BASE-MEM-HEAP) are notorious for requiring tailored exploitation techniques, especially in the presence of mitigations and when aiming for reliable exploitation. KHeaps [97] studies that reliability, showing that some known helper techniques (HELP-MEM-HEAP) are not as effective as expected, and proposes a new technique to improve exploit reliability. Pspray [55] proposes a side channel to bypass kernel slab randomization (HELP-MEM-HEAP). RetSpill [98] adapts the pivot-to-heap ROP technique used as part of some control-flow hijacking exploits to work with the kernel stack (HELP-MEM-PE-CFH), thereby achieving greater exploitation flexibility. SLUBStick [65] proposes a generic cross-cache attack technique that bypasses many current mitigations (HELP-MEM-HEAP). Blindside [31] introduces a speculative probing technique (HELP-TEA), which helps defeating a wide range of kernel layout randomization techniques under the assumption of a strong vulnerability allowing overwriting specific pointers (BASE-MEM). Ghostrace [74] combines Spectre [50] branch misprediction attacks (BASE-TEA) with concurrent use-after-free (BASE-MEM-TM-HEAP), as well as a novel race condition exploitation technique (HELP-RAC) showing that architecturally race-free regions of kernel code can be exploited speculatively to achieve arbitrary data leakage. CARDSHARK [36] helps in timing race condition exploits by automatically inferring the necessary delay for succesful exploitation (HELP-RAC).

Automated exploit generation We categorize as automated exploit generation papers that aim to automate S_4 or S_5. KEPLER [92] assumes the attacker has a control-flow hijacking primitive (i.e., controls the instruction pointer) and automatically generates a return-oriented programming (ROP) payload that either creates an arbitrary read primitive via invocation of `copy_to_user` or arbitrary write via `copy_from_user`, achieving partial S_4 in our classification. Control-flow-integrity for the kernel prevents such ROP-based attacks, making data-only exploitation strategies necessary. ELOISE [18] automatically identifies such data-only objects whose length field, if corrupted as consequence of a vulnerability, leads to information leakage from the kernel (partially S_4). KOOBE [17] starts from a fuzzer-found out-of-bounds heap access, and automates the capability extraction (S_3) as well as solving for exploitability in pre-determined cases (partially S_4). SLAKE [19] automates kernel heap manipulation by identifying three primitives required for kernel primitives: allocation, deallocation and dereference. It first identifies candidate objects for each category via static analysis, and uses fuzzing to generate code for each primitive. Together with finding the required sequence of allocation and deallocation, this is used to automatically manipulate the heap into the required state for exploitation of most kernel use-after-free vulnerabilities (partial S_4). AlphaEXP [87] creates knowledge graphs related to kernel objects and functions, with the goal of identifying interesting objects for memory corruption (HELP-MEM-IL, HELP-MEM-PE-DATA and HELP-MEM-PE-IL). K-LEAK [57] statically builds a memory-error-aware data flow graph of the Linux kernel, harnesses capabilities dynamically via KASAN reports, identifies potential infoleak strategies via a search algorithm, and verifies these via symbolic execution. SCAVY [9] identifies kernel objects that can be used as targets of slab memory corruption vulnerabilities, by using a heavily modified Syzkaller implementation simulating corruption of interesting fields and testing for elevated privileges.

Vulnerability class	Papers
BASE-MEM (memory safety)	DR. CHECKER [66], K-Miner [30], K-LEAK [57]
BASE-MEM-TM (temporal)	K-MELD [28]
BASE-MEM-TM-HEAP	Goshawk [64], FUZE [93], SLAKE [19]
BASE-MEM-SP (spatial)	KOOBE [17], Drammer [81]
BASE-TEA (transient execution)	KASPER [43], Ghostrace [74]
BASE-LOGIC-AC (missing access control)	Pex [101]
PRE-CAST (type confusion)	DR. CHECKER [66]
PRE-INIT (uninit. data)	DR. CHECKER [66], UniSan [63], IncreLux [100], UBITect [99]
PRE-INT (integer)	KINT [90]
PRE-INT-ALC (allocation count)	DR. CHECKER [66]
PRE-INT-IDX (index overflow)	DR. CHECKER [66]
PRE-MC (missing check)	CRIX [62]
PRE-PHYS-DRAM	Drammer [81]
PRE-RAC-TT (TOCTOU)	DR. CHECKER [66], LRSan [88]
PRE-RAC-DF (double fetch)	DoubleFetch [85], Deadline [94], DECAF [76]
PRE-RAC (race conditon)	RAZZER [41]
PRE-TAINT (tainted data)	DR. CHECKER [66]
PRE-UB-INT (integer)	KINT [90]
HELP-MEM-HEAP (heap manipulation)	FUZE [93], SLAKE [19], Pspray [55], SLUBStick [65], SCAVY [9]
HELP-MEM-IL-DATA (data-only infoleak)	ELOISE [18], AlphaEXP [87], K-LEAK [57], PT-Rand [26]
HELP-MEM-PE-CFH (control-flow hijack)	KEPLER [92], Ret2dir [46], RetSpill [98]
HELP-MEM-PE-DATA (data-only)	AlphaEXP [87], PT-Rand [26], Ret2dir [46]
HELP-MEM-RAND-ASLR-LEAK	Prefetch [33], KALD [11]
HELP-RAC (race condition)	ExpRace [56], Ghostrace [74], CARDSHARK [36]
HELP-RAC-DL (double lock)	K-Miner [30]
HELP-TEA (transient execution)	Blindside [31]
Agnostic	GREBE [58], SyzScope [102], HFL [48], kAFL [75], RAZZER [41], FuzzNG [13]

Table 2: Distribution of surveyed papers for each class of vulnerability. "Agnostic" refers to papers where the vulnerability class depends on another tool (the chosen sanitizer).

5 Open problems and recommendations

Our systematization identifies a number of gaps around automated kernel exploitation that could be targeted in future work, among which: attacker models and goals tend to only focus on one or two common cases; few vulnerability classes are addressed in an end-to-end manner for AEG; portability and tool reuse and reproducibility is generally limited.

5.1 Attacker model and goal

The attacker model and goal is paramount for prioritization of vulnerabilities. For instance, a system administrator must know in which attacker model a vulnerability exists in order to evaluate whether their system is affected. The same applies for the attacker goal. In most papers, such practical considerations are often secondary. Static analysis papers tackling S_1 are usually attacker-model-agnostic, because they do not consider the reachability of the vulnerable code. This can be beneficial as it covers any potential attacker model, but also tends to lead to large false positives, making developers less likely to act on such bugs. In a sense, fuzzing goes further, and leads one to consider attacker models, given that reachability of code is conditioned on the harness design. Table 1 shows that most papers that do have a specific attacker model focus on the local attacker model. This is reasonable, as it constitutes the most common practical attacker situation, as well as one of the largest attack surface for the kernel [52]. Nevertheless, other attacker models, as shown in Section 2.2, are also relevant from a practical standpoint. This indicates that there is a gap in other attacker models such as the kernel sandbox, kernel secure boot or remote attacker models, with the potential of many undiscovered bugs in the corresponding attack surface, as well as novel exploitation strategies and automated exploit generation approaches. Similarly, most approaches focus on PE or IL as the attacker goal. While these two represent the most impactful attacks,

DoS attacks, especially in remote scenarios, may also have significant practical impact (e.g., CVE-2019-11477). Only one paper explores this area [28], and only in a local setting, making it a potential area for future work.

5.2 Bug prioritization beyond path exploration

Generally, any AEG technique demonstrating exploitability of a bug (e.g., [17,57,87]) helps in prioritizing bugs, as they clearly demonstrate practical security impact. Nevertheless, because AEG approaches are still limited to specific vulnerability classes, this cannot be used widely across all bugs: the inability of an AEG tool to generate an exploit does not imply that the bug is of lower priority, as the bug may still be exploitable with a different approach. Approaches that consider exploitability considerations beyond reachability lead to wider bug prioritization: this is the explicit motivation for SyzScope and GREBE [58, 102], which both explore other paths leading to the same defect, with the idea that some other paths may lead to stronger attacker capability, and therefore higher priority. While they do not demonstrate exploitability, the hint they give to developers with respect to the bug's capability is more accurate than what a standalone sanitizer can indicate. However, we note that bug prioritization also requires exploring other aspects, and not only other paths that may lead to the same bug. First, the assumed attacker model impacts a bug's practical impact: a remote attacker model should get the higher priority, whereas an attack requiring administrative privileges (for example in the Lockdown attacker model in Section 2.2) is a much lower priority. This consideration is not theoretic: Syzkaller is run with administrative privileges, to allow it to achieve as high of a code coverage as possible. This means that it also reports defects that are not reachable in the typical local threat model. Bug prioritization in precise threat models is therefore a large area where further research is needed. Second, the category of bug reported by the sanitizer, such as a stack-based buffer overflow, does not necessarily imply exploitability, and AEG techniques could help in the future in obtaining a more accurate prioritization, for instance in cases where such a buffer-overflow may not be exploitable due to mitigations.

5.3 Vulnerability classes

While Table 1 and Table 2 show that most vulnerability classes are covered, some remain to be tackled, in particular with respect to automated exploit generation techniques. For example, no automated exploitation technique exists for Spectre-type of kernel vulnerabilities. Given that the exploitation (and exploitability evaluation) of such bugs is difficult, this could be an important area of future work. Similarly, Blindside [31] showed that spectre v1-like attacks can be used to help finding an exploit strategy in the presence of mitigations: automating such approaches could be tackled in future work. Another example is with respect to race conditions (PRE-RAC and HELP-RAC), which have not yet been used in the kernel AEG context. In part, this remains challenging because the widely used S2E [21] engine does not support multiple-CPU symbolic execution. For the exploitation of heap vulnerabilities (HELP-MEM-HEAP), while we only mention one technique, there is an abundance of recent techniques with each a different area of applicability [35, 59, 61, 95], depending on the vulnerability and mitigations. In general, while S_1 and S_2 (and sometimes S_3) cover a wide range of vulnerability classes, only few S_4 and S_5 tools cover a wide range of vulnerability classes. This intuitively makes sense: end-to-end automation of exploitation requires making greater assumptions about the specific vulnerability type, the kernel version and the available mitigations.

5.4 Portability

Most automation techniques still heavily rely on templates or domain specific knowledge included in the particular tool. This means that their portability, whether it is across Linux versions or across different operating systems, is low. Combined with versioning limitations as well as API changes for dependencies such as S2E and LLVM, their portability is highly limited and requires very significant engineering work. One possible solution would be to automate porting of exploits across different kernel versions, as proposed by AEM [42]. However, this technique only works for a subset of exploits and kernel version, as more complex kernel changes cannot be handled. Beyond being a practical problem for wider adoption, lack of portability causes significant issues in research: reproducibility problems, making comparisons with prior work particularly difficult. For example, two AEG solutions may only work on two distinct kernel versions, making it particularly challenging to evaluate differences. A possible solution to this problem would be to fix a kernel version for all studies, by analogy with biology research, where studies are conducted on model organisms such as *C. elegans* [45]. These organisms are chosen and bred to be highly similar to one other. This leads to our recommendation in favor of the community agreeing on a fixed Linux kernel version for reproducibility and comparison purposes, for example one of the long-term support (LTS) kernel releases, which could be referred to as a long-term research (LTR) version. While this would lead to research being performed on "outdated" LTR kernels in the long run, we argue this

is largely a problem from the practical industry adoption perspective, a task that would in any case require large amount of additional work, even when working with the latest kernel version. However, this would have tremendous benefits for research, allowing us to compare different approaches in an accurate manner, but also drastically reducing the amount of extraneous engineering required for reusing research artifacts when versions differ. If the community stands behind this idea, defenses would also be developed for this LTR version, making it easier to evaluate and compare them. Therefore, we argue the benefits of the systems community at large fixing a research kernel version for multiple years outweighs its drawbacks.

5.5 Tooling, reuse and reproducibility

In total, among vulnerability-finding papers, we survey 14 papers with static analysis as main tooling, and 8 papers using fuzzing as core technique. For exploitation-oriented papers, we identify 10 manual exploitation technique papers, and 9 papers that do automated exploit generation. Some tools are used across many papers, such as Syzkaller for fuzzing, LLVM for static analysis, and, to a lesser extent, S2E and angr [84] for dynamic symbolic execution. This is highly positive for these tools, as it is a testimony to their robustness and usability. Nevertheless, many approaches could benefit from increased code reuse. For example, many different papers reimplement passes to acquire a precise kernel control flow graph. Similarly, papers using S2E for the kernel often end up re-implementing a number of helper tooling that is similar but not robust enough to be reused across papers. Finally, we notice that while many of the approaches could be combined to achieve better results (for example, KOOBE [17] could be combined with SLAKE [19] to also automate heap manipulation), doing so would be very difficult as the code bases are largely incompatible. This could call for a modular framework that would make such approaches easier, akin to how LLVM made writing compiler passes easier and more modular, and result in end-to-end exploit generation results, which is currently lacking with each paper focusing on demonstrating feasibility for a small part of the problem. Finally, we note that a number of papers do not have source code available, making reproducibility very difficult. However, we notice that papers published in the past few years do have code available, which can likely be attributed to the recent changes in conference policies with respect to artifact availability.

5.6 Ethics of AEG

A common concern is around the ethics of automated exploit generation work: it can be argued that this research makes the attacker's work easier, and therefore increases workload for defenders. Conversely, one can argue that such research makes it possible to better prioritize bug fixing, targetting first exploitable vulnerabilities and leaving only hard to exploit vulnerabilities to attackers. It can also be argued that it makes it easier to systematically evaluate defenses. We believe the crucial differentiator in tipping the balance towards more benefits than harm is in whether AEG research is done publicly. Public AEG research, just like public research on advanced exploitation techniques, improves defender's understanding on what attacks are feasible and automatable, which eventually leads to more robust defenses. Therefore, we recommend to encourage public AEG research, while still ensuring that typical offensive security ethical boundaries are satisfied: for example, ensuring that any exploits generated and published as part of the work target bugs that do have fixes, or where maintainers have been notified well in advance.

5.7 Improving AEG evaluations

Although most AEG papers do discuss limitations of their approach, we notice that this is typically not experimentally evaluated in a measurable way. A recent trend is to evaluate against Google's kernelCTF [32], which provides instances for attackers to demonstrate their exploit, including an instance containing known vulnerabilities to demonstrate mitigation bypasses. However, this only provides anecdotal evidence related to the vulnerabilities (and mitigations) available on the chosen instance. We foresee two possibilities for tackling this issue. The first one, commonly used in other areas, is to establish a benchmark dataset: this could be based on exploits from kernelCTF[1], or a set of bugs identified by a fuzzer, all corresponding to the vulnerability class targeted for the particular AEG approach, with exploits available as ground truth for exploitability. Currently, each paper selects its own set of bugs, making it likely that the dataset is biased, and making comparisons difficult. However, this approach is difficult to implement: given that many AEG papers are the first to tackle a particular class of vulnerability, no relevant benchmark dataset exists, and a new one needs to be created. An alternative would be to evaluate on a set of bugs selected in a provably random manner, for example the top N bugs by git commit id that correspond to the required vulnerability class and attacker model. Beyond enabling

[1] https://github.com/google/security-research/tree/master/pocs/linux/kernelctf

fair comparisons, where future work can demonstrate improvements in a measurable way, this would also give an idea with respect to the proportion of bugs where an exploit can be generated.

6 Conclusion

We systematize knowledge from past work concerning the automation of vulnerability discovery and exploit generation on the Linux kernel. The pervasiveness of deployments of OS kernels such as Linux, with their large attack surface and high privilege level, make a very appealing target for adversaries and therefore a critical component whose security must be very well understood, thus justifying a more thorough analysis of the works in this space. We first systematize the concepts pertaining to kernel exploitation, defining realistic threat models and attacker goals. We break down the steps involved in exploitation of a kernel vulnerability and present the techniques that are leveraged to automate any of these steps. We then proceed to organize the papers in the literature across several dimensions, including to what extent they adopt any of the automation techniques, in which way they do so, whether they are able to combine them and which vulnerabilities they focus on. After the systematization, we highlight a set of open research problems, which we hope will foster discussion and research on the challenging topic of automatic kernel exploit generation.

Acknowledgments

The authors would like to thank our shepherd Kyle Zeng, and anonymous reviewers for their comments on an earlier draft of this paper.

References

[1] Seccomp(2) - Linux manual page. https://man7.org/linux/man-pages/man2/seccomp.2.html.

[2] Syzbot. https://syzkaller.appspot.com/upstream.

[3] Zerodium exploit bounty program. https://zerodium.com/program.html, 2024. [Online; accessed 06-Jun-2023].

[4] R. Abbott, J. Chin, Jed Donnelley, W. Konigsford, S. Tokubo, and D. Webb. Security analysis and enhancements of computer operating systems. page 70, 04 1976.

[5] Lillian Ablon, Martin C Libicki, and Andrea A Golay. *Markets for cybercrime tools and stolen data: Hackers' bazaar*. Rand Corporation, 2014.

[6] James P Anderson et al. Computer security technology planning study. Technical report, Citeseer, 1972.

[7] Ross J Anderson. *Security engineering: a guide to building dependable distributed systems*. John Wiley & Sons, 2010.

[8] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.

[9] Erin Avllazagaj, Yonghwi Kwon, and Tudor Dumitras. SCAVY: Automated discovery of memory corruption targets in linux kernel for privilege escalation. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 7141–7158, Philadelphia, PA, August 2024. USENIX Association.

[10] Alessandro Barenghi, Luca Breveglieri, Niccolò Izzo, and Gerardo Pelosi. Software-only reverse engineering of physical dram mappings for rowhammer attacks. In *2018 IEEE 3rd International Verification and Security Workshop (IVSW)*, pages 19–24. IEEE, 2018.

[11] Brian Belleville, Wenbo Shen, Stijn Volckaert, Ahmed M. Azab, and Michael Franz. Kald: Detecting direct pointer disclosure vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 18(3):1369–1377, 2021.

[12] Herbert Bos. Corruption of Memory: Those who don't know history are doomed to repeat it. Keynote at NDSS'24, 2024.

[13] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. No grammar, no problem: Towards fuzzing the linux kernel without system-call descriptions. In *Network and Distributed System Security (NDSS) Symposium*, 2023.

[14] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 249–266, 2019.

[15] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek.

Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, pages 1–5, 2011.

[16] Weiteng Chen, Yu Hao, Zheng Zhang, Xiaochen Zou, Dhilung Kirat, Shachee Mishra, Douglas Schales, Jiyong Jang, and Zhiyun Qian. SyzGen++: Dependency inference for augmenting kernel driver fuzzing. In *IEEE Symposium on Security and Privacy*, 2024.

[17] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. KOOBE: Towards facilitating exploit generation of kernel Out-Of-Bounds write vulnerabilities. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1093–1110. USENIX Association, August 2020.

[18] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. A systematic study of elastic objects in kernel exploitation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1165–1184, New York, NY, USA, 2020. Association for Computing Machinery.

[19] Yueqi Chen and Xinyu Xing. Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1707–1722, New York, NY, USA, 2019. Association for Computing Machinery.

[20] Yueqi Chen and Xinyu Xing. SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1707–1722, London United Kingdom, November 2019. ACM.

[21] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The s2e platform: Design, implementation, and applications. *ACM Transactions on Computer Systems (TOCS)*, 30(1):1–49, 2012.

[22] Lucian Cojocar, Jeremie Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu. Are we susceptible to rowhammer? an end-to-end methodology for cloud providers. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 712–728. IEEE, 2020.

[23] Jonathan Corbet. Fun with NULL pointers, part 1. https://lwn.net/Articles/342330/, 2009. [Online; accessed 19-Oct-2023].

[24] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. DIFUZE: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2017.

[25] Scott A Crosby and Dan S Wallach. Denial of service via algorithmic complexity attacks. In *12th USENIX Security Symposium (USENIX Security 03)*, 2003.

[26] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. Pt-rand: Practical mitigation of data-only attacks against page tables. In *Proc. of 24th Annual Network & Distributed System Security Symposium (NDSS)*. feb 2017.

[27] Mark Dowd, John McDonald, and Justin Schuh. *The art of software security assessment: Identifying and preventing software vulnerabilities*. Pearson Education, 2006.

[28] Navid Emamdoost, Qiushi Wu, Kangjie Lu, and Stephen McCamant. Detecting kernel memory leaks in specialized modules with ownership reasoning. In *The 2021 Annual Network and Distributed System Security Symposium (NDSS'21)*, 2021.

[29] Nicolas Falliere, Liam O Murchu, Eric Chien, et al. W32. stuxnet dossier. *White paper, symantec corp., security response*, 5(6):29, 2011.

[30] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. K-miner: Uncovering memory corruption in linux. In *NDSS*, 2018.

[31] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative probing: Hacking blind in the spectre era. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1871–1885, New York, NY, USA, 2020. Association for Computing Machinery.

[32] Google. kctf - kctf is a kubernetes-based infrastructure for ctf competitions, 2025.

[33] Daniel Gruss. Software-based microarchitectural attacks, 2017.

[34] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 368–379, 2016.

[35] Ziyi Guo, Dang K Le, Zhenpeng Lin, Kyle Zeng, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, Adam Doupé, and Xinyu Xing. Take a step further: Understanding page spray in linux kernel exploitation. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1189–1206, Philadelphia, PA, August 2024. USENIX Association.

[36] Tianshuo Han, Xiaorui Gong, and Jian Liu. CARDSHARK: Understanding and stablizing linux kernel concurrency bugs against the odds. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 6203–6218, Philadelphia, PA, August 2024. USENIX Association.

[37] Yu Hao, Guoren Li, Xiaochen Zou, Weiteng Chen, Shitong Zhu, Zhiyun Qian, and Ardalan Amiri Sani. SyzDescribe: Principled, automated, static generation of syscall descriptions for kernel drivers. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 3262–3278. IEEE, 2023.

[38] Sean Heelan, Tom Melham, and Daniel Kroening. Automatic heap layout manipulation for exploitation. In *27th USENIX security symposium (USENIX security 18)*, pages 763–779, 2018.

[39] Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang, Han-Lin Lu, and Chung-Wei Lai. Software crash analysis for automatic exploit generation on binary programs. *IEEE Transactions on Reliability*, 63(1):270–289, 2014.

[40] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Analyzing integrity protection in the SELinux example policy. In *12th USENIX Security Symposium (USENIX Security 03)*, 2003.

[41] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 754–768, 2019.

[42] Zheyue Jiang, Yuan Zhang, Jun Xu, Xinqian Sun, Zhuang Liu, and Min Yang. Aem: Facilitating cross-version exploitability assessment of linux kernel vulnerabilities. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2122–2137. IEEE, 2023.

[43] Brian Johannesmeyer, Jakob Koschel, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Kasper: Scanning for generalized transient execution gadgets in the linux kernel. In *NDSS*, volume 1, page 12, 2022.

[44] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, volume 2, page 0, 2004.

[45] Titus Kaletta and Michael O Hengartner. Finding function in novel targets: C. elegans as a model organism. *Nature reviews Drug discovery*, 5(5):387–399, 2006.

[46] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. ret2dir: Rethinking kernel isolation. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 957–972, San Diego, CA, August 2014. USENIX Association.

[47] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking kernel isolation. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 957–972, 2014.

[48] Kyungtae Kim, Dae Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kernel. 01 2020.

[49] Amit Klein and Benny Pinkas. From IP ID to device ID and KASLR bypass. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1063–1080, Santa Clara, CA, August 2019. USENIX Association.

[50] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy*, 2019.

[51] Jakob Koschel, Pietro Borrello, Daniele Cono D'Elia, Herbert Bos, and Cristiano Giuffrida. Uncontained: uncovering container confusion in the linux kernel. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5055–5072, 2023.

[52] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. Attack surface metrics and automated compile-time OS kernel tailoring. In *Proceedings of the 20th Network and Distributed Systems Security Symposium*. The Internet Society, 2013.

[53] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[54] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. Taming undefined behavior in llvm. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 633–647, New York, NY, USA, 2017. Association for Computing Machinery.

[55] Yoochan Lee, Jinhan Kwak, Junesoo Kang, Yuseok Jeon, and Byoungyoung Lee. Pspray: Timing Side-Channel based linux kernel heap exploitation technique. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6825–6842, Anaheim, CA, August 2023. USENIX Association.

[56] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. ExpRace: Exploiting kernel races through raising interrupts. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2363–2380. USENIX Association, August 2021.

[57] Zhengchuan Liang, Xiaochen Zou, Chengyu Song, and Zhiyun Qian. K-leak: Towards automating the generation of multi-step infoleak exploits against the linux kernel. In *NDSS*. Internet Society, 2024.

[58] Zhenpeng Lin, Yueqi Chen, Yuhang Wu, Dongliang Mu, Chensheng Yu, Xinyu Xing, and Kang Li. Grebe: Unveiling exploitation potential for linux kernel bugs. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2078–2095, 2022.

[59] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. Dirtycred: Escalating privilege in linux kernel. In *Proceedings of the 2022 ACM SIGSAC conference on computer and communications security*, pages 1963–1976, 2022.

[60] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. Platypus: Software-based power side-channel attacks on x86. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 355–371. IEEE, 2021.

[61] Danjun Liu, Pengfei Wang, Xu Zhou, Wei Xie, Gen Zhang, Zhenhao Luo, Tai Yue, and Baosheng Wang. From release to rebirth: Exploiting thanos objects in linux kernel. *IEEE Transactions on Information Forensics and Security*, 18:533–548, 2022.

[62] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting Missing-Check bugs via semantic- and Context-Aware criticalness and constraints inferences. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1769–1786, Santa Clara, CA, August 2019. USENIX Association.

[63] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 920–932, New York, NY, USA, 2016. Association for Computing Machinery.

[64] Yunlong Lyu, Yi Fang, Yiwei Zhang, Qibin Sun, Siqi Ma, Elisa Bertino, Kangjie Lu, and Juanru Li. Goshawk: Hunting memory corruptions via structure-aware and object-centric memory operation synopsis. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2096–2113, 2022.

[65] Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. Slubstick: Arbitrary memory writes through practical software cross-cache attacks within the linux kernel. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

[66] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. DR. CHECKER: A soundy analysis for linux kernel drivers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1007–1024, Vancouver, BC, August 2017. USENIX Association.

[67] Andrea Mambretti, Alexandra Sandulescu, Alessandro Sorniotti, William Robertson, Engin Kirda, and Anil Kurmus. Bypassing memory safety mechanisms through speculative control flow hijacks. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 633–649. IEEE, 2021.

[68] Linux man page. kernel_lockdown(7). https://man7.org/linux/man-pages/man7/kernel_lockdown.7.html, 2023. [Online; accessed 19-Oct-2023].

[69] Frank Mayer, Karl MacMillan, and David Caplan. *SELinux By Example: Using Security Enhanced Linux*. Prentice Hall PTR, Englewood Cliffs, NJ, USA, 2006.

[70] Paul McKenney. Signed overflow optimization hazards in the kernel. https://lwn.net/Articles/511259/, 2012. [Online; accessed 19-Oct-2023].

[71] Dongliang Mu, Yuhang Wu, Yueqi Chen, Zhenpeng Lin, Chensheng Yu, Xinyu Xing, and Gang Wang. An in-depth analysis of duplicated linux kernel

bug reports. In *Network and Distributed Systems Security (NDSS) Symposium 2022*, 2022.

[72] PacketLabs. Demystifying the market for zero-day software exploits. https://packetlabs.net/posts/demystifying-the-market-for-zero-day-software-exploits/, 2024. [Online; accessed 06-Jun-2024].

[73] Mauro Pezzè and Michal Young. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

[74] Hany Ragab, Andrea Mambretti, Anil Kurmus, and Cristiano Giuffrida. GhostRace: Exploiting and mitigating speculative race conditions. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 6185–6202, Philadelphia, PA, August 2024. USENIX Association.

[75] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted feedback fuzzing for OS kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, Vancouver, BC, August 2017. USENIX Association.

[76] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, page 587–600, New York, NY, USA, 2018. Association for Computing Machinery.

[77] Alexander Sotirov. Heap feng shui in javascript. *Black Hat Europe*, 2007:11–20, 2007.

[78] Hao Sun, Yuheng Shen, Jianzhong Liu, Yiru Xu, and Yu Jiang. KSG: Augmenting kernel fuzzing with system call specification generation. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 351–366, Carlsbad, CA, July 2022. USENIX Association.

[79] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[80] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 405–419, 2017.

[81] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1675–1689, New York, NY, USA, 2016. Association for Computing Machinery.

[82] Julien Vanegue, Sean Heelan, and Rolf Rolles. SMT solvers in software security. *WOOT*, 12:9–22, 2012.

[83] Dmitry Vyukov. Syzkaller, 2015.

[84] Fish Wang and Yan Shoshitaishvili. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9. IEEE, 2017.

[85] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. How Double-Fetch situations turn into Double-Fetch vulnerabilities: A study of double fetches in the linux kernel. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1–16, Vancouver, BC, August 2017. USENIX Association.

[86] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. How {Double-Fetch} situations turn into {Double-Fetch} vulnerabilities: A study of double fetches in the linux kernel. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1–16, 2017.

[87] Ruipeng Wang, Kaixiang Chen, Chao Zhang, Zulie Pan, Qianyu Li, Siliang Qin, Shenglin Xu, Min Zhang, and Yang Li. AlphaEXP: An expert system for identifying Security-Sensitive kernel objects. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4229–4246, Anaheim, CA, August 2023. USENIX Association.

[88] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. Check it again: Detecting lacking-recheck bugs in os kernels. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1899–1913, New York, NY, USA, 2018. Association for Computing Machinery.

[89] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Undefined behavior: What happened to my code? In *Proceedings of the Asia-Pacific Workshop on Systems*, APSYS '12, New York, NY, USA, 2012. Association for Computing Machinery.

[90] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Improving integer security for systems with KINT. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 163–177, Hollywood, CA, October 2012. USENIX Association.

[91] Yan Wang, Chao Zhang, Zixuan Zhao, Bolun Zhang, Xiaorui Gong, and Wei Zou. {MAZE}: Towards automated heap feng shui. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1647–1664, 2021.

[92] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. KEPLER: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1187–1204, Santa Clara, CA, August 2019. USENIX Association.

[93] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards facilitating exploit generation for kernel Use-After-Free vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 781–797, Baltimore, MD, August 2018. USENIX Association.

[94] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in os kernels. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 661–678, 2018.

[95] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 414–425, 2015.

[96] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM*, 53(1):91–99, 2010.

[97] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. Playing for K(H)eaps: Understanding and improving linux kernel exploit reliability. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 71–88, Boston, MA, August 2022. USENIX Association.

[98] Kyle Zeng, Zhenpeng Lin, Kangjie Lu, Xinyu Xing, Ruoyu Wang, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. Retspill: Igniting user-controlled data to burn away linux kernel protections. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 3093–3107, New York, NY, USA, 2023. Association for Computing Machinery.

[99] Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V. Krishnamurthy, and Paul Yu. Ubitect: A precise and scalable method to detect use-before-initialization bugs in linux kernel. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 221–232, New York, NY, USA, 2020. Association for Computing Machinery.

[100] Yizhuo Zhai, Yu Hao, Zheng Zhang, Weiteng Chen, Guorern Li, Zhiyun Qian, Chengyu Song, Manu Sridharan, Srikanth V. Krishnamurthy, Trent Jaeger, and Paul Yu. Progressive Scrutiny: Incremental Detection of UBI bugs in the Linux Kernel. In *Proceedings of the 2020 ISOC Network and Distributed Systems Security Symposium (NDSS)*, February 2022.

[101] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M. Azab, and Ruowen Wang. PeX: A permission check analysis framework for linux kernel. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1205–1220, Santa Clara, CA, August 2019. USENIX Association.

[102] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian. SyzScope: Revealing High-Risk security impacts of Fuzzer-Exposed bugs in linux kernel. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3201–3217, Boston, MA, August 2022. USENIX Association.

Appendix

Paper selection methodology

For exploit generation papers, we have used the following search terms on Google Scholar to select paper: "kernel automated exploit generation", "kernel exploitation" and limited ourselves initially to papers published in top-tier security venues (USENIX Security, ACM CCS, NDSS, and IEEE Security and Privacy) that automate some aspect of kernel exploitation, initially limiting ourselves to papers published from 2013 to 2023. In a second step,

we explored papers cited in the related work sections of these papers, and included papers relevant to finding vulnerabilities in the kernel as well as kernel-specific novel exploitation techniques, even when they were not automated. We reiterated the process we those added papers. Finally, we added papers that were pointed out to us by reviewers of an earlier draft of this paper.