Enforcing Kernel Security Invariants with Data Flow Integrity

Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, Wenke Lee
Georgia Institute of Technology

***Abstract*—The operation system kernel is the foundation of the whole system and is often the de facto trusted computing base for many higher level security mechanisms. Unfortunately, kernel vulnerabilities are not rare and are continuously being introduced with new kernel features. Once the kernel is compromised, attackers can bypass any access control checks, escalate their privileges, and hide the evidence of attacks. Many protection mechanisms have been proposed and deployed to prevent kernel exploits. However, a majority of these techniques only focus on preventing control-flow hijacking attacks; techniques that can mitigate non-control-data attacks either only apply to drivers/modules or impose too much overhead. The goal of our research is to develop a principled defense mechanism against memory-corruption-based privilege escalation attacks. Toward this end, we leverage data-flow integrity to enforce security invariants of the kernel access control system. In order for our protection mechanism to be practical, we develop two new techniques: one for automatically inferring data that are critical to the access control system without manual annotation, and the other for efficient DFI enforcement over the inference results. We have implemented a prototype of our technology for the ARM64 Linux kernel on an Android device. The evaluation results of our prototype implementation show that our technology can mitigate a majority of privilege escalation attacks, while imposing a moderate amount of performance overhead.**

I. Introduction

The operation system (OS) kernel is often the de facto trusted computing base (TCB) of the whole computer system, including many higher level security solutions. For example, the security of an application sandbox usually depends on the integrity of the kernel. Unfortunately, kernel vulnerabilities are not rare in commodity OS kernels like Linux, Windows, and XNU. Once the kernel is compromised, attackers can bypass any access control checks, escalate their privileges, and hide the evidence of attacks. Among all kernel vulnerabilities, the ones related to memory corruption are the most prevalent because all commodity kernels are implemented in low-level unsafe language like C and assembly. Memory corruption bugs are also the most dangerous ones because they can grant attackers great capabilities. For these reasons, most kernel attacks exploit memory corruption vulnerabilities.

Existing solutions to this problem can be classified into two main categories: off-line tools and runtime protection mechanisms. Off-line tools [17, 32, 38, 62, 66] try to identify potential kernel memory safety violations so that developers can fix them before deployment. Although these tools have successfully found many vulnerabilities in the kernel, they have limitations. First, most bug-finding tools tend to generate lots of false positives, which makes it hard for developers to filter out the real bugs. Second, tools that can prove an implementation is free of memory safety issues usually do not scale well. This means they can only be applied to small, self-contained kernel extensions [11, 47] or micro-kernels [35].

For runtime protection mechanisms, a majority focus on protecting the control flow. Many mechanisms are proposed to protect code integrity and stop malicious kernel extensions from loading [55]. Others focus on preventing control-flow hijacking attacks, such as `ret2usr` [33, 51] and return-oriented programming (ROP) [39]. More recently, researchers have demonstrated the feasibility of enforcing control-flow integrity (CFI) in kernel space [22, 65]. However, in addition to problems discovered in CFI [14, 27],a more fundamental problem is that, because OS kernels are mainly data-driven, CFI can be easily bypassed by non-control-data attacks [20]. For example, to bypass discretionary access control (DAC), attackers just need to overwrite the subject's identity of the current process with the one of the root/administrators.

Some technologies are capable of preventing non-control-data attacks. For example, software fault isolation (SFI) [16, 25, 43, 63] can be used to isolate small "untrusted" modules from tampering the core kernel components. However, a recent study on Linux kernel vulnerabilities [18] discovered that vulnerabilities in the core components are as common as vulnerabilities in third-party drivers. The secure virtual architecture [23] is able to provide full memory safety for the kernel, but its performance overhead is too high to be deployed in practice.

The objective of this work is to provide a defense system that is both *principled* (i.e., cannot be easily bypassed by future attacks) and *practical* (i.e., with reasonable performance overhead). We achieve the first goal by utilizing data-flow integrity (DFI) [15] to enforce kernel security invariants against memory-corruption-based attacks. Similar to CFI, DFI guarantees that runtime data-flow cannot deviate from the data-flow graph generated from static analysis. For example, data from a string buffer should never flow to the return address on stack (control-data), or to the `uid` (non-control-data). Utilizing this technique, we can enforce a large spectrum of security invariants in the kernel to defeat different attacks. For instance, to prevent rootkits from hiding malicious processes, we can enforce that only the process scheduler can modify

NDSS '16, 21-24 February 2016, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2016.23218

the linked list of active processes. In this work, we focus on enforcing invariants that are related to kernel access control mechanisms (a.k.a. reference monitors) so as to defeat privilege escalation attacks. Specifically, assuming a reference monitor is implemented correctly, its runtime correctness relies on two high-level security invariants [4]:

I. **Complete mediation**: attackers should not be able to bypass any access control check; and
II. **Tamper proof**: attackers should not be able to tamper with the integrity of either the code or data of the reference monitor.

While this approach sounds intuitive at high level, enforcing it with practical runtime performance overhead is very challenging. First, unlike kernel extensions, which are usually self-contained and use dedicated data structures, access control checks are scattered throughout the kernel, and related data are mixed with other data. Therefore, the protection technique must be deployed kernel-wide. Moreover, without hardware support, software-based DFI implementation can be very expensive, e.g., the original enforcement technique from [15] imposed an average 104% overhead for user-mode CPU benchmarks. Since OS kernels tend to be more data intensive than those CPU benchmarks, we expect the same technique will be even more expensive for kernel protection.

We propose a system called KENALI that is both principled and practical. KENALI consists of two key techniques. Our first technique, INFERDISTS, is based on the observation that although the protection has to be kernel-wide, only a small portion of data is essential for enforcing the two security invariants. For ease of discussion, we refer to this set of data as *distinguishing regions* (formally defined in §IV-A). Hence, instead of enforcing DFI for all kernel data, we only need to enforce DFI over the distinguishing regions. Implementing this idea must solve one important problem—*what data belongs to the distinguishing regions*. To enforce INVARIANT I, we must enforce CFI, so all control-data should be included. The real challenge is non-control-data. As mentioned above, access control checks are scattered throughout the kernel, and data involved in these checks are usually mixed with non-critical data in the same data structure. One solution, as used by many kernel integrity checkers [10, 53, 54], is to identify these data based on domain knowledge, execution traces, or analysis of previous attacks. However, because these approaches are not sound, attackers may find new attack targets to bypass their protection. Alternatively, one can manually analyze and annotate distinguishing regions. However, this approach does not scale and is error-prone. Our technique, INFERDISTS, overcomes these challenges with a new program-analysis-based approach. Specifically, by leveraging implicit program semantics, INFERDISTS is able to infer security checks without any manual annotation. After this, by considering both data- and control-dependencies of data that can affect each security check, as well as sensitive pointers [28], INFERDISTS generates a *complete* set of distinguishing regions.

Our second technique, PROTECTDISTS, is a new technique to enforce DFI over the distinguishing regions. In particular, since distinguishing regions only constitutes a small portion of all kernel data, PROTECTDISTS uses a two-layer protection scheme. The first layer provides a coarse-grained but low-overhead data-flow isolation that prevents illegal data-flow from non-distinguishing regions to distinguishing regions. After this separation, the second layer then enforces fine-grained DFI over the distinguishing regions. Furthermore, because the access pattern to most data in the distinguishing regions is very asymmetric—read accesses (checks) are usually magnitudes more frequent than write accesses (updates)—instead of checking the data provenance at every read instruction, PROTECTDISTS employs the opposite but equally secure approach: checking if the current write instruction can overwrite the target data, i.e., write integrity test (WIT) [3]. Combining these two techniques, namely, INFERDISTS and PROTECTDISTS, KENALI is able to enforce the two security invariants without sacrificing too much performance.

We implemented a prototype of KENALI that protects the Linux kernel for the 64-bit ARM architecture (a.k.a. AArch64). We chose AArch64 Linux for the following reasons. First, INFERDISTS requires source code to perform the analysis, and thanks to the success of Android, Linux is the most popular open-sourced kernel now. Second, due to the complexity of the Android ecosystem, a kernel vulnerability usually takes a long cycle to patch.This means kernel defense techniques play an even more critical role for Android. Third, among all the architectures that Android supports, ARM is the most popular. Moreover, compared to other architectures like x86-32 [67] x86-64 [24] and AArch32 [68], there is little research on efficient isolation techniques for AArch64. Despite this choice, we must emphasize that our techniques are general and can be implemented on other hardware architectures and even other OS kernels (see §VII).

We evaluated the security and performance overhead of our prototype implementation. The results show that KENALI is able to prevent a large variety of privilege escalation attacks. At the same time, its performance overhead is also moderate, around 7-15% for standard Android benchmarks.

To summarize, this paper makes the following contributions:

- **Analysis technique**: We devised a systematic, formalized algorithm to discover distinguishing regions that scales to Linux-like, large commodity software (§IV-B).
- **Design**: We presented a new data-flow integrity enforcement technique that has lower performance overhead and provides identical security guarantees as previous approaches (§IV-C).
- **Formalization**: We formalized the problem of preventing memory-corruption-based privilege escalation and showed that our approach is a sound solver for this problem (§IV).
- **Implementation**: We implemented a prototype of KENALI for the ARM64 Linux kernel, which utilizes a novel hardware fault isolation mechanism (§V).
- **Evaluation**: We evaluated the security of KENALI with various types of control-data and non-control-data attacks (e.g., CVE-2013-6282) that reflect the real kernel privilege escalation attacks used for rooting Android devices (§VI-C). We also evaluated the performance overhead of our prototype with micro-benchmarks to stress KENALI's runtime component, and end-to-end benchmark to measure the performance overheads similar to what an end-user might experience. Our evaluation showed that KENALI imposes acceptable overheads for practical deployment, around 7% - 15% for

standard Android benchmarks (§VI-D).

II. Problem Scope

In this section, we define the problem scope of our research. We first discuss our threat model and assumptions. Then, we provide motivating examples of our techniques and show how they can bypass state-of-the-art defense techniques like CFI.

A. Threat Model and Assumptions

Our research addresses kernel attacks that exploit memory corruption vulnerabilities to achieve privilege escalation. We only consider attacks that originate from unprivileged code, such as user-space application without root privilege. Accordingly, attacks that exploit vulnerabilities in privileged system processes (e.g., system services or daemons) are out-of-scope. Similarly, because kernel drivers and firmwares are already privileged, we do not consider attacks that originate from kernel rootkits and malicious firmwares. Please note that we do *not* exclude attacks that exploit vulnerabilities in kernel drivers, but only attacks from malicious kernel rootkits. Other kernel exploits such as denial-of-service (DoS), logical/semantic bugs, or hardware bugs [34] are also out-of-scope. We believe this is a realistic threat model because (1) it covers a majority of the attack surface and (2) techniques to prevent the excluded attacks have been proposed by other researchers and can be combined with KENALI.

However, we assume a *powerful* adversary model for memory corruption attacks. Specifically, we assume there is one or more kernel vulnerabilities that allow attackers to read and write word-size value at an arbitrary virtual address, as long as that address is mapped in the current process address space.

Since many critical non-control-data are loaded from disk, another way to compromise the kernel is to corrupt the disk. Fortunately, solutions already exist for detecting and preventing such attacks. For example, secure boot [6] can guarantee that the kernel and KENALI are not corrupted on boot; SUNDR enables data to be stored securely on untrusted servers [40]; ZFS enforces end-to-end data integrity using the Merkle tree [12]; and Google also introduced a similar mechanism called DMVerity [5] to protect the integrity of critical disk partitions. For these reasons, we limit the attack scope of this paper to memory-corruption-based attacks and exclude disk-corruption-based attacks.

B. Motivating Examples

We use a real vulnerability, CVE-2013-6282 [46], as a running example to demonstrate the various types of attacks feasible under our threat model, and to illustrate how and why these attacks can bypass the state-of-the-art defense techniques like CFI and ad-hoc kernel integrity protection. Given this vulnerability, we begin with an existing attack against the Linux kernel. Then, step-by-step, we further demonstrate two additional attacks showing how this original attack can be extended to accomplish a full rooting attack.

1) Simple rooting attacks. CVE-2013-6282 allows attackers to read and write arbitrary kernel memory, which matches our adversary model. The corresponding rooting attack provides a good example of how most existing kernel privilege escalation exploits work:

a) Retrieving the address of `prepare_kernel_cred()` and `commit_creds()`. Depending on the target system, they can be at fixed addresses, or obtainable from the kernel symbol table (`kallsyms_addresses`);

b) Invoking `prepare_kernel_cred()` and pass the results to `commit_creds()`, then the kernel will replace the credential of the current thread with one of root privilege.

Step b can be done in several ways: attackers can overwrite a function pointer to a user mode function that links these two functions together (i.e., `ret2usr`). Alternatively, attackers can also link them through return-oriented programming.

2) Bypassing CFI with non-control-data attacks. The above attack can be prevented by kernel-wide CFI [22]. But CFI can be easily bypassed by non-control-data attacks: by locating the `cred` structure and overwriting the `euid` field, attackers can still escalate the privilege to the root user. The `cred` structure can be located in many ways: (1) if `kallsyms` is available and contains the address of `init_task`, we can easily traverse the process list to locate the `task_struct` of the current process, then `task_struct->cred`; (2) if there is a vulnerability that leaks the stack address (e.g., CVE-2013-2141), attackers can directly obtain the address of the `thread_info` structure, then follows the links to locate the `task_struct`; and (3) with arbitrary memory read capability, attackers can also scan the whole kernel memory and use signature matching to identify the required data structures [41].

3) Bypassing CFI with control-data attacks. In this example, we designed a new attack to demonstrate another limitation of CFI. Specifically, to prevent root privilege from being acquired through compromising system daemons, Android leverages SELinux, a mandatory access control mechanism, to further restrict the root privilege [57]. Therefore, disabling SELinux is a necessary step to gain the full root privilege. This can be achieved through control-data attacks that do not violate CFI. In particular, SELinux callbacks are stored in a dispatch table that has a special initialization phase:

```
1  // @security/capability.c
2  void security_fixup_ops(struct security_operations *ops) {
3    if (!ops->sb_mount)
4      ops->sb_mount = cap_sb_mount;
5    if (!ops->capable)
6      ops->capable = cap_capable;
7    ...
8  }
9  static int cap_sb_mount(const char *dev_name, ...) {
10     return 0;
11 }
```

Basically, if a Linux Security Module (LSM) does not implement a hook, its callback function will be set to the default one (e.g., `cap_sb_mount`). Therefore, the default callback functions are also valid control transfer targets, but they usually perform no checks before directly returning 0. Based on this observation, SELinux can then be disabled by setting every callback function pointer to its default one.

4) Diversity of non-control-data attacks. Some existing kernel integrity protection mechanisms also try to protect non-control-data [10, 53, 54], such as `uid`. However, these approaches are inherently limited because there can be many

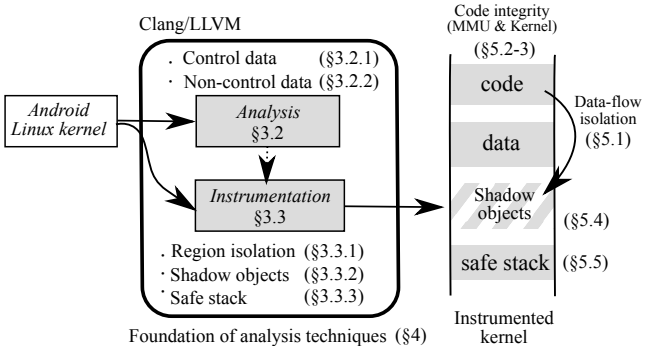

Fig. 1: Overview of our approach, which consists two major steps. In the first step, we use off-line program analysis to infer memory regions that must be protected. In the second step, we use a lightweight runtime protection mechanism to guarantee data-flow integrity for regions discovered in the first step.

different non-control-data involved in access control. For example, from the target kernel we evaluated, we found that 2,419 data structures contain critical data. Here, we use a concrete non-control-data attack to demonstrate this limitation of previous work.

Specifically, the above two steps only grant attackers temporary root privilege until the next reboot (a.k.a. tethered root). To acquire permanent root privilege (untethered root), the de facto way is to install the `su` utility. To do so, however, there is one more protection to bypass: read-only mounting. To protect critical system files, the system partition on most Android devices is mounted as read-only. Existing rooting attacks achieve this goal by remounting the partition as writable, but we achieve this goal through another non-control-data attacks:

```
1 // @fs/namespace.c
2 int __mnt_is_readonly(struct vfsmount *mnt) {
3     if (mnt->mnt_flags & MNT_READONLY)
4         return 1;
5     if (mnt->mnt_sb->s_flags & MS_RDONLY)
6         return 1;
7     return 0;
8 }
```

As we can see, by overwriting data fields like `mnt_flags` and `s_flags`, attackers can bypass the read-only mount and overwrite the system partition.

III. Technical Approach

A. Overview

Figure 1 provides an overview of our technical approach, which consists of two steps. In the first step, we use a novel program analysis technique InferDists to systematically infer a complete and minimized set of distinguishing regions. In the second step, we employ a lightweight runtime protection technique, ProtectDists, to enforce DFI over the inference result.

B. Inferring Distinguishing Regions

In this subsection, we present our approach to the inference problem. The challenge is that for security, our solution must be sound (i.e., no false negatives), but for performance, we want the size of the inference result to be as small as possible.

1) Control-Data: As discussed in §I, Invariant I can be violated via control-data attacks. Therefore, all control-data must be included as distinguishing regions so as to enforce CFI. Control-data in the kernel has two parts: general control-data and kernel-specific data. General control-data (e.g., function pointers) can be identified based on the type information [36]. Kernel-specific data, such as interrupt dispatch table, have been enumerated in [22]. Since our approach to infer these data does not differ much from previous work, we omit the details here.

2) Non-Control-Data: The challenge for inferring distinguishing non-control-data is the soundness, i.e., whether a proposed methodology can discover *all* distinguishing regions. To address this challenge, we developed InferDists, an automated program analysis technique. The key idea is that access controls are implemented as security checks, and while a kernel may have many security checks scattered throughout different components, they all follow one consistent semantic: *if a security check fails, it should return a security related error codes*. For example, a POSIX-compatible kernel returns `-EACCES` (permission denied) [60] to indicate the current user does not have access to the requested resources. Similarly, Windows also has `ERROR_ACCESS_DENIED` [59]. Leveraging this observation, InferDists is able to collect security checks without manual annotation. Then, distinguishing regions can be constructed via standard dependency analysis over the conditional variables of security checks. Next, we use Example 1 as a running example to demonstrate how InferDists works. The formal model of InferDists and proof for its soundness are provided in §IV.

```
1  int acl_permission_check(struct inode *inode, int mask) {
2    unsigned int mode = inode->i_mode;
3    if (current_cred->fsuid == inode->i_uid)
4      mode >>= 6;
5    else if (in_group_p(inode->i_gid))
6      mode >>= 3;
7    if ((mask & ~mode &
8        (MAY_READ | MAY_WRITE | MAY_EXEC)) == 0)
9      return 0;
10   return -EACCES;
11 }
```

Example 1: A simplified version of the discretionary access control (DAC) check in the Linux kernel.

In step **(1)**, InferDists collects the return instructions `R` that may return an error code that we are interested in.

> In Example 1, the function returns `-EACCES` at line 10, so InferDists include this instruction into `R`.

In step **(2)**, InferDists collects the branch instructions `I` that determine if a run either executes a return instructions in `R` or performs a potentially privileged operation. The conditional variables `B` of `I` will be distinguishing variables of this function.

> In Example 1, the condition of the `if` statement at line 7 is the distinguishing conditional variable of the function. Note, the `if` statement at line 3 is not included because it is post-dominated by the `if` statement at line 7.

It is worth mentioning that we need to handle some special cases in step **(2)**. First, in some cases, when a security check fails, it will not directly return an error but will continue to another alternative check, e.g., in the `setuid` system call, three checks are performed:

```
1 if (!nsown_capable(CAP_SETUID) &&
2     !uid_eq(kuid, old->uid) && !uid_eq(kuid, new->suid))
3     return -EPERM;
```

In this case, when the first check `nsown_capable` failed, the control flow will go to the second, and possibly the third check. Although these are all security checks, in the control flow graph (CFG), only the branch instruction for the last `uid_eq` check will lead to an error return. If we only consider this branch, then we will miss the first two checks. Therefore, we must also consider branch(es) that dominate a security check. However, naively including all dominators will introduce many false positives. To reduce false positives, INFERDISTS conservatively excludes two cases: (a) a branch can lead to non-related error return (e.g., `-EINVAL`) and (b) a branch instruction is post-dominated by either a security check or checks in (a), i.e., diamond-shaped nodes.

In step **(3)**, INFERDISTS collects and returns all memory regions that have dependencies on conditional variables in `B`. For completeness, we consider both *data- and control-dependencies*, and the analysis is inter-procedural and iterative, i.e., we perform the analysis multiple times until there is no new data get included.

> In Example 1 the `if` statement on line 7 has two conditional variables, `mask` and `mode`. Data-dependency over `mode` would include `i_mode` (line 2) and control-dependency would include `i_uid`, `fsuid` (line 3), and the return value of `in_group_p` (line 5). Because our dependency analysis is inter-procedural, we will also include `i_gid`.

3) Sensitive Pointers: Pointers to sensitive regions must be protected as well; otherwise, attackers can *indirectly* control the data in distinguishing regions by manipulating these pointers [28]. For instance, instead of overwriting the `euid` field of a `cred` structure, an attacker could overwrite the `task_struct->cred` pointer to point to a `cred` structure whose `euid == 0`.

Hence, after collecting control- and non-control-data, the analysis collects the *sensitive pointers* of a program, and includes such pointers in distinguishing regions. A pointer is sensitive if it points to a data structure that contains distinguishing regions. There are two points worth noting: (1) sensitive pointers are defined recursively and (2) even with the existence of generic pointers, we still can collect a sound over-approximation set of sensitive pointers using a static program analysis, so no false negatives will be introduced.

C. Protecting Distinguishing Regions

After collecting the distinguishing regions, the next step is to enforce DFI over the inference result. The challenge for this step is how to minimize the performance overhead on commodity processors that lack support for fine-grained data-flow tracking. To address this challenge, our key observation is that, after (conceptually) separating the memory into distinguishing and non-distinguishing regions, there could be three types of data-flow: (1) within non-distinguishing regions, (2) between two regions, and (3) within distinguishing regions. Our goal is to prevent attackers from introducing *illegal data-flow* to compromise distinguishing regions. Obviously, it is not possible to compromise distinguishing regions based on the first type of data-flow, but most legal data-flows actually belong to this type. Therefore, if we purely rely on DFI to vet all data-flows, there will be a huge number of unnecessary checks. Based on this observation, we design a two-layer scheme: the first layer is a lightweight data-flow isolation mechanism that prevents illegal data-flow of the second type; then we use the more expensive DFI enforcement to prevent illegal data-flow of the third type. With this two-layer approach, we can reduce the performance overhead without sacrificing security guarantees.

1) Data-Flow Isolation: There are two general approaches to enforce data-flow isolation: software-based and hardware-based. Software fault isolation is not ideal because it requires instrumenting a majority of write operations. For example, in our prototype implementation, only 5% of write operations can access distinguishing regions; thus, relying on SFI would end up with instrumenting the remaining 95% of write operations. At the same time, not all hardware isolation mechanisms are equally efficient. Since distinguishing regions are actually not continuous, but interleaved with non-distinguishing regions, it would be ideal if the hardware could support fine-grained memory isolation. Unfortunately, most commodity hardware does not support this but only provides coarse-grained isolation mechanisms. Based on their corresponding overhead (from low to high), the available options on commodity hardware are: the segmentation on x86-32 [67], the execution domain on ARM-32 [68], the WP flag on x86-64 [24, 65], hardware virtualization [52, 56], and TrustZone [9].

In this work, we explored the feasibility of hardware-based data-flow isolation for AArch64, which is becoming more and more popular for mobile devices, but has not been well studied before. For AArch64, most of the aforementioned features are not available, except TrustZone[1]; but world switch for TrustZone is usually very expensive because it needs to flush the cache and sometimes the TLB (translation look-aside buffer) too. To solve this problem, we developed a novel, virtual address space-based isolation mechanism. Specifically, to reduce the overhead of context switching between different virtual address spaces, modern processors usually tag the TLB with an identifier associated with the virtual address space. Utilizing this feature, we can create a trusted virtual address space by reserving an identifier (e.g., ID = 0). By doing so, context switching between the untrusted context and trusted context becomes less expensive because it requires neither TLB flush nor cache flush (see §V for more details).

2) Write Integrity Test: In addition to preventing illegal data-flow from non-distinguishing regions to distinguishing regions, we use DFI to prevent illegal data-flow within distinguishing regions. However, instead of checking data provenance at read, we leveraged the write integrity test (WIT) technique [3]. We chose this technique for the following reasons. First, we found that the memory access pattern for distinguishing regions is very asymmetric, e.g., reading `uid` is prevalent, but updating `uid` is very rare. Thus, by checking write operations, we can reduce the number of checks that need to be performed. Second, WIT reasons about safe and unsafe write operations and only instruments unsafe writes, which matches another observation—the majority of writes to the distinguishing regions are safe and do not require additional checks. Finally, compared to memory

[1]Hardware virtualization extension is defined, but a majority of processors do not implement it; and for those who have this feature, the bootloader usually disabled it.

safety enforcement techniques [23], WIT is less expensive because it does not require tracking pointer propagation. Because of page limitation, we omit the details of WIT; please refer to the original paper for more details. However, in order to apply this technology to the kernel, we made one change. In particular, the original WIT implementation used a context-sensitive field-insensitive point-to analysis, but since OS kernels usually contain a lot of generic pointers and linked lists, we replaced the point-to analysis with a context-sensitive and field-sensitive analysis that is tailored for the kernel [13].

3) Shadow Objects: Shadow objects is a work-around for the lack of fine-grained hardware isolation mechanism. Specifically, as a hardware protection unit (e.g., page) may contain both distinguishing and non-distinguishing regions, once we write-protect that page, we also have to pay additional overhead for accessing non-distinguishing regions. One solution to this problem is to manually partition data structures that contain mixed regions into two new data structures [58]. However, this approach does not scale and requires heavy maintenance if the data structure changes between different kernel versions. Our solution to this problem is *shadow objects*, i.e., if a kernel object contains both regions, then we will create two copies of it—a normal copy for the non-distinguishing regions and a shadow copy for the distinguishing regions. Shadow memory may consume up to two times the original memory, but because commodity OS kernels usually use memory pools (e.g., `kmem_cache`) to allocate kernel objects, it allows us to reduce the memory overhead by dedicating different pools to objects that need to be shadowed. This nice feature also allows us to eliminate maintaining our own metadata for shadow objects allocation/free; and to perform fast lookup—giving a pointer to normal object, its shadow object can be acquired by adding a fixed offset.

4) Safe Stack: Similar to heap, stack also needs a "shadow" copy for the lack of fine-grained isolation. However, stack is treated differently for its uniqueness. First, stack contains many critical data that are not visible at the source code level, such as return addresses, function arguments, and register spills [21]. Second, the access pattern of these critical data is also different: write accesses are almost as frequent as read accesses. For these reasons, we leveraged the safe-stack technique proposed in [36], with a few improvements to make this technique work better for the kernel. First, we used more precise inter-procedural analysis to reduce the number of unsafe stack objects from 42% to 7%. Second, as the number of unsafe stack objects is very small, instead of using two stacks, we keep only one safe stack and move all unsafe objects to the heap to avoid maintaining two stacks.

IV. Formal Model

To demonstrate that our technical approach is correct, we formalize the problem of preventing privilege escalation via memory corruption and describe an approach to solve the problem. In §IV-A, we formulate the problem of rewriting a monitor with potential memory vulnerabilities to protect privileged system resources. In §IV-B, we formulate the sub-problem of inferring a set of memory regions that, if protected, are sufficient to ensure that a monitor protects privileged resources. In §IV-C, we formulate the problem of rewriting a program to protect a given set of memory regions. In §IV-D, we show that our approaches to solve the inference and protection problems can be composed to solve the overall problem.

A. Problem Definition

1) A language of monitors: A monitor state is a valuation of data variables and address variables, where each address is represented as a pair of a base region and an offset. Let the space of *machine words* be denoted Words and let the space of *error codes* be denoted $\mathsf{ErrCodes} \subseteq \mathsf{Words}$. Let the space of *memory regions* be denoted $\mathsf{Regions}$, and let an *address* be a region paired with a machine word, i.e., $\mathsf{Addrs} = \mathsf{Regions} \times \mathsf{Words}$.

Let the space of *control locations* be denoted $\mathtt{Locs}$, let the space of *protection colors* be denoted colors, let the space of data variables be denoted $\mathtt{D}$, and let the space of address variables be denoted $\mathtt{A}$.

Definition 1: A monitor *state* consists of (1) a control location, (2) a map from each address to its size, (3) a map from each address to its protection color, and (4)–(6) maps from $\mathtt{D}$, $\mathtt{A}$, and Addrs to the word values that they store. The space of monitor states is denoted Q.

A monitor *instruction* is a control location paired with an operation. The set of operations contains standard read operations $\mathtt{read\ a, d}$, conditional branches $\mathtt{bnz\ d}$, returns $\mathtt{ret\ d}$, arithmetic operations, and logical operations. The set of operations also contains the following non-standard operations:

(1) For each protection color $c \in \mathsf{colors}$, each address variable $\mathtt{a} \in \mathtt{A}$, and each data variable $\mathtt{d} \in \mathtt{D}$, $\mathtt{alloc}[c]\ \mathtt{a, d}$ allocates a new memory region $R \in \mathsf{Regions}$, sets the size of R to be to the word stored in $\mathtt{d}$, and stores the address $(R, 0)$ in $\mathtt{a}$. The space of allocation instructions is denoted Allocs.

(2) For each protection color $c \in \mathsf{colors}$, each data variable $\mathtt{d} \in \mathtt{D}$, and each address variable $\mathtt{a} \in \mathtt{A}$, $\mathtt{write}[c]\ \mathtt{d, a}$ attempts to write the value stored in $\mathtt{d}$ to the address stored in $\mathtt{a}$. Let $\mathtt{a}$ store the address (R, o). If in the current state, the color of R is c, then the write occurs; otherwise, the program aborts. If $\mathtt{a}$ stores an address inside of its base region, then the write is *safe*; otherwise, the write is *unsafe*. The space of write instructions is denoted Writes.

Definition 2: For each instruction $\mathtt{i} \in \mathsf{Instrs}$, the *transition relation* of $\mathtt{i}$ is denoted $\sigma[\mathtt{i}] \subseteq Q \times Q$.

We refer to σ as the *unrestricted* transition relation, as it places no restriction on the target address of a write outside of a region (in contrast to *restricted* transition relations, defined in Defn. 5). The formal definitions of the transition relations of each instruction follow directly form the instruction's informal description and thus are omitted.

Definition 3: A *monitor* is a pair $(\mathtt{I}, \mathtt{A})$, where $\mathtt{I} \in \mathsf{Instrs}^*$ is a sequence of instructions in which (1) each variable is defined exactly once and (2) only the final instruction is a return instruction; $\mathtt{A} \subseteq \mathsf{Allocs}$ are the allocation sites of *privileged regions*. The space of monitors is denoted $\mathsf{Monitors} = \mathsf{Instrs}^* \times \mathcal{P}(\mathsf{Instrs})$.

A *run* of M is an alternating sequence of states and instructions such that adjacent states are in the transition relation of the unrestricted semantics of the neighboring instruction; the runs of $\mathtt{M}$ under the unrestricted semantics are denoted $\mathsf{Runs}(\mathtt{M})$.

The *safe* runs of $\mathtt{M}$ are the runs in which each write instruction only writes within its target region.

A monitor $\mathtt{M}' \in \mathsf{Monitors}$ is a *refinement* of $\mathtt{M}$ if each run of $\mathtt{M}'$ is a run of $\mathtt{M}$. $\mathtt{M}'$ is a *non-blocking refinement* of $\mathtt{M}$ if (1) $\mathtt{M}'$ is a refinement of $\mathtt{M}$ and (2) each safe run of $\mathtt{M}$ is a run of $\mathtt{M}'$.

2) Monitor consistency: A monitor *inconsistency* is a pair of runs (r_0, r_1) from the same initial state, where r_0 accesses a privileged region and r_1 returns an error code.

A monitor is *weakly consistent* if for each *run*, the monitor exclusively either accesses privilege regions or returns an error code. That is, monitor $\mathtt{M} = (\mathtt{I}, \mathtt{A}) \in \mathsf{Monitors}$ is weakly consistent if there is no run $r \in \mathsf{Runs}(\mathtt{M})$ such that (r, r) is an inconsistency. A core assumption of our work, grounded in our study of kernel access control mechanisms, is that practical monitors are usually written to be weakly consistent.

A monitor is *strongly consistent* if for each *initial state*, the monitor exclusively either accesses privileged regions or returns an error code. That is, $\mathtt{M}$ is *strongly consistent* if there are no runs $r_0, r_1 \in \mathsf{Runs}(\mathtt{M})$ such that (r_0, r_1) is an inconsistency.

3) The consistent-refinement problem: Each strongly consistent monitor is weakly consistent. However, a monitor $\mathtt{M}$ may be weakly consistent but *not* strongly consistent if it contains a memory error that prohibits $\mathtt{M}$ from ensuring that all runs from a given state either access privileged regions or return error codes. The main problem that we address in this work is to instrument all such monitors to be strongly consistent.

Definition 4: For monitor $\mathtt{M}$, a solution to the *consistent-refinement problem* $\textsc{Refine}(M)$ is a non-blocking refinement of M (Defn. 3) that is strongly consistent.

We have developed a program rewriter, KENALI, that attempts to solve the consistent-refinement problem. Given a monitor $\mathtt{M}$, KENALI first infers a set of *distinguishing allocation sites* $\mathtt{A}$ for which it is sufficient to protect the integrity of all regions in order to ensure consistency (§IV-B). KENALI then rewrites $\mathtt{M}$ to protect the integrity of all regions allocated at $\mathtt{A}$ (§IV-C). The rewritten module $\mathtt{M}'$ is a non-blocking refinement of $\mathtt{M}$, and all potential inconsistencies of $\mathtt{M}'$ can be strongly characterized (§IV-D).

B. Inferring distinguishing regions

1) Problem formulation: We now formulate the problem of inferring a set of memory regions that are sufficient to protect a monitor to be strongly consistent. We first define a semantics for monitors that is parameterized on a set of allocation sites $\mathtt{A}$, under which the program can only modify a region allocated at a site in $\mathtt{A}$ with a safe write.

Definition 5: For allocation sites $\mathtt{A} \subseteq \mathsf{Allocs}$ and instruction $\mathtt{i} \in \mathsf{Instrs}$, the *restricted semantics of* R $\sigma_{\mathtt{A}}(\mathtt{i}) \subseteq Q \times Q$ is the transition relation over states such that: **(1)** if $\mathtt{i}$ is not a write, then $\sigma_{\mathtt{A}}[\mathtt{i}]$ is identical to $\sigma[\mathtt{i}]$; **(2)** if $\mathtt{i}$ is a write, then for an unsafe write, the program may write to any region *not allocated at a site in* $\mathtt{A}$

For each monitor $\mathtt{M} \in \mathsf{Monitors}$, the runs of $\mathtt{M}$ under the restricted semantics for R are denoted $\mathsf{Runs}_R(\mathtt{M})$.

The distinguishing-site inference problem is to infer a set of allocation sites $\mathtt{A}$ such that if all regions allocated at sites in $\mathtt{A}$ are protected, then the monitor is consistent.

Definition 6: For each monitor $\mathtt{M} = (\mathtt{I}, \mathtt{A}) \in \mathsf{Monitors}$ and set of regions, a solution to the *distinguishing-site inference* problem $\textsc{Dists}(\mathtt{M})$ is a set of allocation sites $\mathtt{A}' \subseteq \mathsf{Allocs}$ such that $\mathtt{M}$ is consistent under the restricted semantics $\mathsf{Runs}_{\mathtt{A}'}(\mathtt{M})$. We refer to such a set $\mathtt{A}'$ as *distinguishing sites* for $\mathtt{M}$.

2) Inferring distinguishing sites: In this section, we present a solver, INFERDISTS, for solving DISTS. INFERDISTS proceeds in three steps: **(1)** INFERDISTS collects the return instructions $\mathtt{R}$ that may return an error code. **(2)** INFERDISTS collects the condition variables $\mathtt{B}$ of the branch instructions that determine if a run either executes a return instructions in $\mathtt{R}$ or accesses *any* memory region. **(3)** INFERDISTS returns the dependency sites of all condition variables of $\mathtt{B}$. We now describe phases in detail.

a) Background: data-dependency analysis: For monitor $\mathtt{M}$, data variable $\mathtt{x} \in \mathtt{D}$, and allocation site $\mathtt{a} \in \mathsf{Allocs}$, $\mathtt{a}$ is a *dependency site* of $\mathtt{x}$ if over some run of $\mathtt{M}$, the value stored in some region allocated at $\mathtt{a}$ partially determines the value stored in $\mathtt{x}$ (the formal definition of a data dependency is standard) [2]. The data-dependency-analysis problem is to collect the dependency sites of a given set of data variables.

Definition 7: For a monitor $\mathtt{M} \in \mathsf{Monitors}$ and data variables $\mathtt{D} \subseteq \mathtt{D}$, a solution to the *data-dependency-analysis* problem $\textsc{Deps}(\mathtt{M}, \mathtt{D})$ is a set of allocation sites that contain the dependency sites of all variables in $\mathtt{D}$.

Our solver for the DISTS problem, named INFERDISTS, constructs instances of the DEPS problem and solves the instances by invoking a solver, INFERDEPS. The implementation of INFERDEPS used by INFERDISTS is a field-sensitive and context-sensitive analysis based on previous work [13].

b) Phase 1: collect error-return instructions: Phase 1 of INFERDISTS collects an over-approximation $\mathtt{E}$ of the set of instructions that may return error codes. While in principle the problem of determining whether a given control location returns an error code is undecidable, practical monitors typically use simple instruction sequences to determine the codes that may be returned by a given instruction. Thus, INFERDISTS can typically collect a precise set $\mathtt{R}$ using constant folding, a standard, efficient static analysis [2].

c) Phase 2: collect distinguishing condition variables: After collecting an over-approximation $\mathtt{E}$ of the instructions that may return error codes, INFERDISTS collects an over-approximation $\mathtt{C}$ of data variables that determine if a program returns an error code or accesses a sensitive resource; we refer to such data variables as *distinguishing condition variables*. INFERDISTS collects $\mathtt{C}$ by performing the following steps: **(1)** INFERDISTS computes the post-dominator relationship $\mathsf{PostDom} \subseteq \mathsf{Instrs} \times \mathsf{Instrs}$, using a standard efficient algorithm. **(2)** INFERDISTS computes the set of pairs $\mathsf{ImmPred} \subseteq \mathsf{Instrs} \times \mathsf{Instrs}$ such that for all instructions $\mathtt{i}, \mathtt{j} \in \mathsf{Instrs}$, $(\mathtt{i}, \mathtt{j}) \in \mathsf{ImmPred}$ if there is a path from $\mathtt{i}$ to $\mathtt{j}$ that contains no post-dominator of $\mathtt{i}$. $\mathsf{ImmPred}$ can be computed efficiently from $\mathtt{M}$ by solving standard reachability problems on the control-flow graph of $\mathtt{M}$. **(3)** INFERDISTS collects the set of branch instructions $\mathtt{B}$ such that for each branch instruction

$\mathtt{b} \equiv \mathtt{bnz\ x, T, F} \in \mathtt{B}$, there is some error-return instruction $\mathtt{e} \in \mathtt{E}$ and some access $\mathtt{a} \in \mathtt{accesses}$ such that $\mathsf{ImmPred}(\mathtt{b}, \mathtt{T})$ and $\mathsf{ImmPred}(\mathtt{b}, \mathtt{F})$. **(4)** INFERDISTS returns all condition variables of instructions in B.

d) Phase 3: collect data dependencies of conditions: After INFERDISTS collects a set of distinguishing condition variables C, it collects the dependency sites $\mathtt{A}'$ of C by invoking INFERDEPS (Defn. 7) on M and C.

C. Protecting distinguishing regions

1) The region-protection problem: For monitor M and set of allocation sites $\mathtt{A} \subseteq \mathsf{Allocs}$, the region-protection problem is to color M so that it protects each region allocated at each site in A.

Definition 8: For each monitor $\mathtt{M} \in \mathsf{Monitors}$ and set of allocation sites $\mathtt{A} \subseteq \mathsf{Allocs}$, a solution to the *distinguishing-site-protection problem* $\textsc{DistsProt}(\mathtt{M}, \mathtt{A})$ is a monitor $\mathtt{M}' \in \mathsf{Monitors}$ such that each run of $\mathtt{M}'$ under the unrestricted semantics (Defn. 2) is a run of M under the restricted semantics for A (Defn. 5).

2) Background: writes-to analysis: For module $\mathtt{M} \in \mathsf{Monitors}$, the writes-to analysis problem is to determine, for each `write` instruction w in a monitor, the set of regions that w may write to in some run of M.

Definition 9: For monitor $\mathtt{M} \in \mathsf{Monitors}$, a solution to the *writes-to analysis problem* $\textsc{WrTo}(\mathtt{M})$ is a binary relation $R \subseteq \mathsf{Writes} \times \mathsf{Allocs}$ such that if there is some run of M in which a write-instruction $\mathtt{w} \in \mathsf{Writes}$ writes to a region allocated at allocation site $\mathtt{a} \in \mathsf{Allocs}$, then $(\mathtt{w}, \mathtt{a}) \in R$.

Our implementation of PROTECTDISTS uses a writes-to analysis SOLVEWRTO, which is built from a points-to analysis provided in the LLVM compiler framework [42].

3) A region-protecting colorer: Given an input monitor $\mathtt{M} \in \mathsf{Monitors}$ and a distinguishing set of allocation sites $\mathtt{A} \subseteq \mathsf{Allocs}$, PROTECTDISTS solves the protection problem $\textsc{DistsProt}(\mathtt{M}, \mathtt{A})$ using an approach that is closely related to the WIT write-integrity test [3]. In particular, PROTECTDISTS performs the following steps: **(1)** PROTECTDISTS obtains a writes-to relation W by invoking SOLVEWRTO on the writes-to analysis problem $\textsc{WrTo}(\mathtt{M})$. **(2)** PROTECTDISTS constructs the restriction of W to only allocation sites in A, denoted $W_{\mathtt{A}}$. **(3)** PROTECTDISTS collects the set $\mathcal{C}$ of connected components of W, and for each component $C \in \mathcal{C}$, chooses a distinct color c_C. **(3)** PROTECTDISTS replaces each allocation instruction $\mathtt{alloc}[0]\ \mathtt{a}, \mathtt{d}$ of M in component $C \in \mathcal{C}$ with the "colored" instruction $\mathtt{alloc}[c_C]\ \mathtt{a}, \mathtt{d}$. **(4)** PROTECTDISTS replaces each write instruction $\mathtt{write}[0]\ \mathtt{a}, \mathtt{d}$ of M in component $D \in \mathcal{C}$ with the "colored" instruction $\mathtt{write}[c_D]\ \mathtt{a}, \mathtt{d}$.

PROTECTDISTS may thus be viewed as a "paramaterized WIT" that uses WIT's technique for coloring write-instructions and regions to protect only the subset of the regions that a monitor may allocate to determine whether to access a privileged region or return an error code.

D. Protected monitors as refinements

In this section, we characterize the properties of modules instrumented by solving the distinguishing-site inference (§IV-B) and protection problems (§IV-C). Let the module instrumenter KENALI be defined for each monitor $\mathtt{M} \in \mathsf{Monitors}$ as follows:

$$\textsc{Kenali}(\mathtt{M}) = \textsc{ProtectDists}(\mathtt{M}, \textsc{InferDists}(\mathtt{M}))$$

KENALI does not instrument a monitor to abort unnecessarily on runs in which the monitor does not perform an unsafe write.

Theorem 1: For each monitor $\mathtt{M} \in \mathsf{Monitors}$, $\textsc{Kenali}(\mathtt{M})$ is a non-blocking refinement of M.

While the modules generated by KENALI may have access-control inconsistencies, each inconsistency can be characterized by the results of the points-to analysis used by PROTECTDISTS.

Theorem 2: For each monitor $\mathtt{M} \in \mathsf{Monitors}$, let W be the writes-to relation of M found by SOLVEWRTO (Defn. 9). If (r_0, r_1) is an inconsistency of $\textsc{Kenali}(\mathtt{M})$, then r_0 is of the form $q_0, \ldots, q_n, (\mathtt{w} \equiv \mathtt{write\ a, d}), q_{n+1}$, and there are a pair of regions $R_0 \neq R_1 \in \mathsf{Regions}$ allocated at sites $\mathtt{A}_0, \mathtt{A}_1 \in \mathsf{Allocs}$ such that (1) R_0 is the target region of w, (2) R_1 is the region written by w, and (3) $\mathtt{A}_0$ and $\mathtt{A}_1$ are in the writes-to set for w in W.

Our practical evaluation of our approach (§VI-B) indicates that the accuracy of state-of-the-art points-to analyses greatly restricts the potential for inconsistencies in rewritten programs. Proofs of all theorems will be provided in an extended version of this article.

V. A PROTOTYPE FOR ANDROID

In this section, we present the prototype implementation for KENALI. As a demonstration, we focus on the AArch64 Linux kernel that powers Android devices. We start this section with a brief introduction of the AArch64 virtual memory system architecture (VMSA) and how we leveraged various hardware features to implement data-flow isolation (§V-A). Next, we describe the techniques we used to enforce MMU integrity (§V-B). Then we discuss the challenges we had to overcome to implement the shadow objects technique on top of the Linux SLAB allocator (§V-C). Finally, we describe our implementation of safe stack for this particular prototype (§V-D).

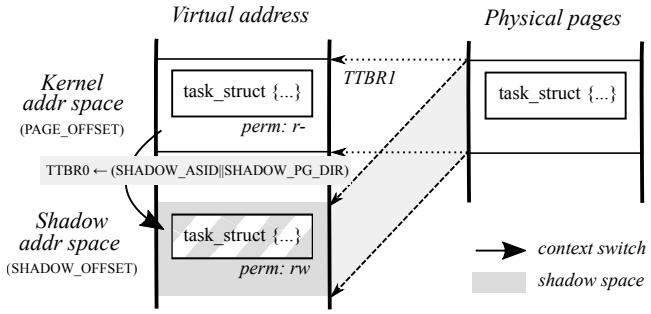

Fig. 2: Shadow address space of KENALI. Under this protection scheme, the same physical memory within different address spaces has different access permissions. Our target platform stores ASID in TTBR0, so we use the bottom half of the VA as our shadow address space and dedicate ASID 0 to this address space. Note that, although the bottom half of the VA is also used for user space memory, because the ASID of the shadow address space is reserved, it does not conflict with any user address space.

A. Data-flow Isolation

For this prototype, we developed a new virtual address space isolation-based technique that leverages several nice features of the AArch64 virtual memory system architecture (VMSA) [7].

1) AArch64 VMSA: The AArch64 supports a maximum of 48-bit virtual address (VA), which is split into two parts: the bottom part is for user space, and the top part is for kernel space. VA to PA (physical address) translation descriptors are configured through two control registers: `TTBR0` (translation table base register) for the bottom half and `TTBR1` for the top half. To minimize the cost of context switch, the AArch64 TLB has several optimizations. First, it allows global pages (mainly for kernel), i.e., translation results are always valid regardless of current address space. Second, each process-specific VA is associated with an ASID (address space identifier), i.e., translation results are cached as (ASID + VA) $\Rightarrow$ PA.

2) Shadow Address Space: Our data-flow isolation implementation leverages the ASID tagging feature and is based on shadow address space, as illustrated in Figure 2. Under this isolation scheme, the same physical page is mapped into two different address spaces with different access permissions. Although the shadow address space technique is not new, the key advantage of our approach is that it does not require TLB flush for context switch. More specifically, most previous systems map the same physical page at the *same VA*. However, because kernel memory is mapped as global, context switching always requires TLB flush to apply the new permission. Our approach instead maps the same physical memory at two *different VAs*. This allows us to leverage the ASID tagging feature to avoid TLB flush for context switch. Another benefit of this approach is that since it does not change the access permissions of global pages, it is *not* subject to multi-core-based attacks, i.e., when one core un-protects the data, attackers can leverage another core to attack.

3) Atomic Primitive Operations: Apparently we cannot keep the shadow address space always available; otherwise attackers can just write to the shadow address. Therefore, we only switch to the shadow address space whenever necessary and immediately switch back to the original (user) context after the operation is done. Moreover, since the kernel can be preemptive (as our target kernel), we also need to disable interruption under the shadow address space to prevent the operation from being interrupted. For these reasons, we want to make every operation under the shadow address space atomic and as simple as possible. Currently, we support the following atomic primitive operations: write a single 8-, 16-, 32-, and 64-bit data, `memcpy`, and `memset`. Example 2 gives an example of performing an atomic 64-bit write operation in the shadow address space.

```
1  ; performing *%addr = %value
2  mrs     x1, daif                          ; save IRQ state
3  msr     daifset, #2                       ; disable IRQ
4  mrs     x2, ttbr0_el1                     ; save current ttbr0
5  msr     ttbr0_el1, %shadow_pgd_and_asid   ; context switch
6  str     %value, %addr                     ; update shadow object
7  dmb     ishst                             ; store barrier
8  msr     ttbr0_el1, x2                     ; restore ttbr0
9  isb                                       ; instruction barrier
10 msr     daif, x1                          ; restore IRQ
```

Example 2: An example of performing an atomic 64-bit write operation in the shadow address space.

B. MMU Integrity

Since our isolation is based on virtual address space, we must guarantee that attackers cannot compromise our isolation scheme. We achieve this goal by enforcing three additional security invariants:

III. **MMU isolation**: We enforce that only MMU management code can modify MMU-related data, including hardware configuration registers and page tables.

IV. **Code integrity**: We enforce that attackers cannot modify existing kernel code or launch code injection attacks.

V. **Dedicated entry and exit**: We enforce that MMU management code is always invoked through dedicated entries and exits, so that attackers cannot jump to the middle of an MMU function and launch deputy attacks.

1) MMU Isolation: Our enforcement technique for INVARIANT III is similar to HyperSafe [65] and nested kernel [24]. First, we enforce that only MMU management code can modify MMU-related configuration registers. (1) Because AArch64 uses one single instruction set that is fix-sized and well aligned, we statically verify that no other code can modify MMU configuration registers. (2) For configuration registers whose values do not need to be changed, such as `TTBR1`, we enforce that they are always loaded with constant values. (3) For registers that can be re-configured like `SCTLR`, we enforce that the possible values either provide the same minimal security guarantees (e.g., enabling and disabling `WXN` bit does not affect the protection because page tables can override) or crash the kernel (e.g., since the kernel VA is much higher than PA, disabling paging will crash the kernel).

The second step is to enforce that all memory pages used for page tables are mapped as read-only in logical mapping. This is done as follows. (1) We break down the logical mapping from section granularity (2MB) to page granularity (4K), so that we can enforce protection at page-level. (2) All physical pages used for initial page tables (i.e., logical mapping, identical mapping and shadow address space) are all allocated from a dedicated area in the `.rodata` section. This is possible because the physical memory for most mobile devices cannot be extended. (3) After kernel initialization, we make all initial page tables as read-only in the logical mapping. (4) We enforce that physical pages used for these critical page tables can never be remapped, nor can their mapping (including access permissions) be modified. This is possible because kernel is always loaded at the beginning of the physical memory according to the ELF section order, so physical pages used for these critical data will have their frame number smaller than the end of the `.rodata` section. (5) Any memory page allocated by the MMU management code is immediately marked as read-only after receiving it from the page allocator.

2) Code Integrity: Enforcing kernel code integrity is essential for enforcing DFI; otherwise attackers can disable our protection by either removing our instrumentations or injecting their own code. We achieve this goal by enforcing two page table invariants. First, similar to page tables, we enforce that the kernel code (`.text`) section is always mapped as read-only. Second, we enforce that except for the `.text` section, no memory can be executable with kernel privilege, i.e., always has `PXN` (privilege execution never) bit [7] set.

3) Dedicated Entries and Exits: Enforcing dedicated entries and exits of MMU management code is trivial, as KENALI protects all code pointers, i.e., its capability of defeating control flow hijacking attacks is equivalent to CPI [36], which is equivalent to fine-grained CFI [1].

C. Shadow Objects

Shadow object support includes three parts: (1) modifications to the SLUB allocator [37], (2) shadowing global objects, and (3) analysis and instrumentation to utilize the above runtime support.

1) SLUB Allocator: In our target kernel, most distinguishing regions are allocated from the SLUB allocator. There are two general types of slabs, i.e., named ones and unnamed ones. Named slabs are usually dedicated to a single data structure, if not merged, while the unnamed ones are for `kmalloc`. Our implementation for shadow objects follows the same design philosophy: we make read access as efficient as possible at the expense of increasing the cost for write operations. Specifically, when SLUB allocates page(s) for a new slab, we allocate a shadow slab of the same size and map it as read-only at a fixed offset (4GB) from the original slab. By doing so, shadow objects can be statically located by adding this fixed offset. Similar to `kmemcheck`, we added one more field in the `page` structure to record the PFN of the corresponding shadow page. Writing to shadow objects requires an additional operation: given a pointer to a shadow object, we (1) subtract the fixed offset to locate the page for its corresponding normal object; (2) retrieve the `page` structure of the normal object and find the PFN for its shadow page; and (3) calculate the VA for the shadow object in the shadow address space, performs a context switch, and write to the VA.

Because recent Linux kernel merges slabs with similar allocation size and compatible flags, we also added one additional flag for `kmem_cache_create` to prevent slabs used for distinguishing regions from merging with other slabs. Finally, since `kmem_caches` for `kmalloc` are created during initialization, we modified this procedure to create additional caches for distinguishing regions allocated through `kmalloc`.

2) Global Objects: While most distinguishing regions are dynamically allocated, some of them are statically allocated in the form of global objects, such as `init_task`. Shadow objects for global objects are allocated during kernel initialization. In particular, we allocate shadow memory for the entire `.data` section, copy all the contents to populate the shadow memory and then map it in the same way as described above, so that we can use a uniformed procedure to access both heap and global objects.

3) Analysis and Instrumentation: Since distinguishing regions contain thousands of data structures, we use automated instrumentation to instruct the kernel to allocate and access shadow objects. Specifically, we first identify all allocation sites for objects in distinguishing regions and modify the allocation flag. If the object is directly allocated via `kmem_cache_alloc`, we will also locate the corresponding `kmem_cache_create` call and modify the creation flag. Next, we identify all pointer arithmetic operations for accessing distinguishing regions and modify them to access the shadow objects (for both read and write access). Finally, we identify all write accesses to distinguishing regions, including `memcpy` and `memset`, and modify them to invoke our atomic operations instead. Our analysis and instrumentation do not automatically handle inline-assembly; fortunately inline-assembly is rare for the AArch64 kernel. There are only 299 unique assembly code snippets, most of which are for accessing system registers, performance counters, and atomic variables. Besides, only a few distinguishing regions are accessed by inlined-assembly, so we handle them manually in a case-by-case manner.

D. Kernel Stack Randomization

Since we use virtual address space for data-flow isolation, performing a context switch for every stack write is not feasible. Thus, we used a randomization-based approach to protect the stack. However, all randomization-based approaches must face two major threats: lack of entropy and information disclosure [26]. We address the entropy problem by mapping kernel stack to a unused VA above the logical map (top 256GB). Because kernel stacks are small (16KB), we have around 24-bit[2] of entropy available.

We contain the risk of information leak as follows. First, when performing safe stack analysis, we mark functions like `copy_to_user` as unsafe, so as to prevent stack addresses from leaking to user space. The safe stack also eliminates stack address leaked to kernel heap, so even with the capability of arbitrary memory read, attackers would not be able to pinpoint the location of the stack. As a result, there are a few special places that can be used to locate the stack, such as the `stack` pointer in the `task_struct` and the page table. To handle the formal case, we store the real stack pointer in a redirection table and replace the original pointer with an index into the table. To protect this table, we map it as inaccessible under normal context. Similarly, to prevent attackers from traversing page tables through arbitrary memory read, page tables for the shadow stacks (i.e., top 256GB) are also mapped as inaccessible under normal context. Accessing these protected tables is similar to writing distinguishing regions, which disables interrupt, performs a context switch, finishes the operation, and restores the context and interrupt. Please note that because we un-map memory pages used for kernel stack from their original VA in logical mapping, attackers can acquire the PFN of the memory by checking un-mapped page table entries. However, this does not reveal the randomized VA of the re-mapped stack and thus cannot be used to launch attacks. Finally, because AArch64 TLB does not hold a TLB entry on both translation error and access flag error [7], TLB-based side channel attacks [29] are also not feasible.

VI. EVALUATION

To evaluate our prototype, we designed and performed experiments in order to answer the following questions:

- How precise is the region-inference algorithm INFERDISTS (§VI-B)?
- How effective is our protection mechanism, PROTECTDISTS, in blocking unauthorized attempts to access distinguishing regions through memory corruption (§VI-C)?
- How much overhead is incurred by our protection mechanism (§VI-D)?

[2] 38-bit (256GB) for unused space above kernel, minus 14-bit size.

A. Experimental setup

We use Google Nexus 9 as our testing device, which embeds a duo-core ARMv8 SoC and 2GB memory. We retrieved the kernel source code from the Android Open Source Project's repository (flounder branch, commit lollipop-release), and applied patches from the LLVMLinux project [61] to make it compatible with LLVM toolchain (r226004). Besides these patches, we modified 64 files and around 1900 LoC of the kernel.

All of our analysis passes are based on the iterative framework from KINT [64], with our own call graph and taint analysis. We found this framework to be more efficient than processing a single linked IR. Our point-to analysis is based on [19], which we extended to be context-sensitive with the technique proposed in [13], and ported to the iterative framework. The total LoC for analysis, excluding changes to the point-to analysis, is around 4400. And our instrumentation pass includes around 500 LoC.

B. Distinguishing Regions Discovery

1) Control data. For the Nexus 9 kernel, our analysis identified 6192 code pointers. Among them, 991 are function arguments and 11 are return values. With safe stack protection, these pointers do not require additional instrumentation. For the rest of the code pointers, 1490 are global variables and 3699 are fields over 783 data structures.

2) Non-Control data. The error codes we used were `EPERM`, `EACCES`, and `EROFS`. Overall, our analysis identified 526 functions as capable of returning permission-related errors; 1077 function arguments, 279 global variables and 1731 data fields over 855 data structures as security-critical.

Next, we measure the accuracy of our analysis. For measuring false positives, we manually verified if the reported data regions are actually involved in the access control decision. Among the 1731 data fields, our manual verification identified 491 fields over 221 data structures as not sensitive, so the empirical false positive rate is about 28.37%. However, most of the false positives (430 data fields over 196 data structures) are introduced by one single check in function `dbg_set_powergate`, which invokes a generic power management function `rpm_resume`; this generic function in turn, invokes power management functions from different drivers through callback function pointers. Since our call graph analysis is context-insensitive, this resulted in including many power management related data structures. If we blacklist this particular check, the false positive rate is only 3.52%.

For false negatives, since our analysis is sound on inferring distinguishing regions, there should be no false negatives. However, because the effectiveness of our approach depends on the correctness of our assumptions (see §VII), we still wanted to check if all well-known sensitive data structures like the `cred` structure are included. The manual verification showed that all the well-known data structures like `av_decision`, `cred`, `dentry`, `file`, `iattr`, `inode`, `kernel_cap_struct`, `policydb`, `posix_acl`, `rlimit`, `socket`, `super_block`, `task_struct`, `thread_info`, `vfsmount`, `vm_area_struct` were included. Because of page limitations, we omit the detailed list of the discovered structures in this paper and provide them as part of an extended technique report. Here, we provide some high level statistic instead. Figure 3 shows the categories of discovered data field according to where they are used for access control[3]. The top 3 sources of distinguishing regions are network (mainly introduced by netfilter), file system, drivers and core kernel.

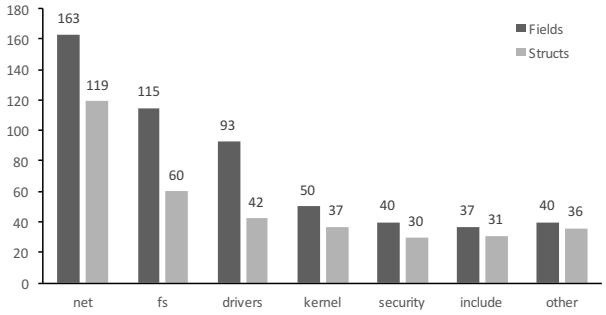

Fig. 3: Data fields categorized by their usage. Note, the total number of usage is larger than the total data fields, because some of the data fields are used in multiple components.

3) Sensitive pointers. Combining both control and non-control inference results, we have a total of 4906 data fields over 1316 data structures as the input for sensitive pointer inference. This step further introduced 4002 fields over 1103 structures as distinguishing regions. So, for the target kernel, KENALI should protect 2419 structures, which is about 27.30% of all kernel data structures (8861).

C. Security Evaluation

In this subsection, we first discuss the potential false negatives introduced by point-to analysis; then we use concrete attacks to show the effectiveness of our protection.

1) Theoretical limitation. Although our analysis is sound, because the point-to analysis is not complete (which is a typical problem for all defense mechanisms that rely on point-to analysis, including CFI, DFI and WIT), we may allow write operations that should never write to distinguishing regions to overwrite those critical data. To reduce the potential false negatives introduced by point-to analysis, we try to improve its precision by making the analysis field-sensitive and context-sensitive. Here, we provide an empirical estimation of this attack surface by measuring how many allocation sites can a pointer points to. The results showed that the majority of pointers (97%) can only point to one allocation site.

2) Real attacks. Since we could not find a real-world attack against our target device, we back-ported CVE-2013-6282 to our target kernel and attacked the kernel with techniques we discussed in §II-B. As shown in Table I, KENALI was able to stop all attacks.

TABLE I: Effectiveness of KENALI against different exploit techniques discussed in §II-B. `ret2usr` attack does not work for our target kernel because the AArch64 kernel has enabled `PXN`.

	`ret2usr`	`cred`	SELinux	RO Mount
Stock	✓	✗	✗	✗
KENALI	✓	✓	✓	✓

[3]We categorized them based on use because most of the structures are defined in header files that are not well categorized.

D. Performance Evaluation

In this subsection, we evaluate the performance overhead introduced by KENALI from the following perspectives: (1) instrumentation statistics; (2) overhead for our atomic operations and core kernel services; (3) overhead for user-mode programs; and (4) memory overhead incurred by shadow objects.

1) Instrumentation overhead. For instrumentation overhead, we report the following numbers.

a) Reallocated stack objects: Recall that to implement safe stack, we relocate unsafe stack objects to the heap. Since heap allocation is more expensive, we want to measure how many objects are relocated. Among the 155,663 functions we analyzed, there are 26,945 stack allocations. Our inter-procedural analysis marked 1813 allocations (6.73%) across 1676 functions as unsafe. Among these 1676 unsafe functions, there are 170 unsafe stores (e.g., storing a stack pointer to the heap) and 308 potential unsafe pointer arithmetics. The rest of them are due to indirect calls where the target functions cannot be statically determined (i.e., function pointers that are never defined), so we conservatively treat them as unsafe. As a comparison, the original safe stack analysis used in [36] marked 11,528 allocations (42.78%) across 7285 functions as unsafe.

b) Allocation sites: Data structures in the distinguishing regions can be allocated in four general ways: as global objects, from heap, on stack, or as an embedded object of a larger structure. Our analysis handles all four cases, with one limitation: our prototype implementation only handles heap objects allocated from SLAB. Overall, for the 2419 input structures, we were able to identify allocation sites for 2146 structures and cannot identify allocation sites for 385 structures. Note that the total number is larger than the input because the result also (recursively) included parent structures of embedded structures. We manually analyzed the result and found that some of those missing data structures like `v4l2_ctrl_config` actually are never allocated in our target kernel configuration, while others are allocated directly from page allocator, such as `f2fs_node`, or casted from a disk block, such as `ext4_inode`.

c) Instrumented instructions: For the target kernel, our analysis processed a total of 158,082 functions and 619,357 write operations. Among these write operations, 26,645 (4.30%) were identified as may access distinguishing regions whose allocation sites were successfully located and thus were replaced with atomic write primitives. Within instrumented write operations, only two operations were marked as unsafe and need to be instrumented to guarantee write integrity. Besides, we also instrumented 137 `memcpy`/`memset` calls.

d) Binary size In this experiment, we measure the binary size increment introduced by KENALI. The result is shown in Table II. As we can see, Clang-generated binaries are smaller than GCC (version 4.9.x-google 20140827), and the binary size increase is minor, so the instrumented binary is only slightly larger than the stock GCC-compiled kernel.

TABLE II: Compressed kernel binary size increment.

	Stock	Clang	KENALI	Increase
Size (in bytes)	7,252,356	6,796,657	7,165,173	5.42%

2) Micro benchmarks. For micro benchmarks, we measured two targets: (1) the overhead for a context switch and (2) the overhead for core kernel services.

a) Context switch: In this experiment, we used the ARM cycle count (`PMCCNTR_EL0`) to measured the latency for a round-trip context switch under our protection scheme. For each round of testing, we performed 1 million context switches, and the final result is based on five rounds of testing. The result, with little deviation, is around 1700 cycles. Unfortunately, lacking access to hypervisor mode and secure world on the target device, we cannot directly compare the expense for context switching to hypervisor and the TrustZone on that device. A recent study [50] showed that on an official Cortex-A53 processor, minimal round-trip cost for a guest-host switch is around 1400 cycles, and around 3700 cycles for a non-secure-secure switch (without cache and TLB flush).

b) LMBench: We used LMBench [44] to measure the latency of various system calls. To measure how different techniques affect these core system services, we consider four configurations: (1) baseline: unmodified kernel compiled with clang; (2) CI: kernel with only life-time kernel code integrity protection (§V-B); (3) Stack: code integrity plus stack protection (§V-D; and (4) KENALI: full DFI protection. The result, averaged over 10 times, is shown in Table III. For comparison, we also included numbers from Nested Kernel [24], which also provides lifetime kernel code integrity; KCoFI [22], which enforces CFI over the whole kernel; and SVA [23], which guarantees full spatial memory safety for the whole kernel. Please note that because these three systems are evaluated on x86 processors and with different kernels, the comparison can only be used as a rough estimation.

TABLE III: LMBench results. Overhead in times slower, the lower the better. ① Because the number for Nested Kernel (PerspicuOS) was reported in a graph, the numbers here are estimations. ② Because the original SVA paper did not use LMBench, we used the number reported in the KCoFI paper

	CI	Stack	KENALI	NK [24]	KCoFI [22]	SVA [23]
null syscall	0.99x	1.00x	1.00x	∼1.0x	2.42x	2.31x
open/close	0.97x	0.99x	2.76x	∼1.0x	2.47x	11.00x
select	1.00x	1.05x	1.42x	-	1.56x	8.81x
signal install	1.34x	1.32x	1.30x	∼1.0x	2.14x	5.74x
signal catch	0.99x	1.11x	2.23x	∼1.0x	0.92x	5.34x
pipe	0.95x	1.02x	3.13x	-	2.07x	13.10x
fork+exit	1.31x	1.40x	2.18x	∼2.9x	3.53x	-
fork+execv	1.50x	1.55x	2.26x	∼2.5x	3.15x	-
page fault	1.61x	1.69x	1.71x	∼1.0x	1.11x	-
mmap	1.60x	1.66x	1.63x	∼2.5x	3.29x	-

As we can see, for syscalls that involve distinguishing regions manipulation, KENALI tends to have higher performance overhead than KoCFI but lower than SVA. But for syscalls that do not involve distinguishing regions manipulation, e.g., null syscall, KENALI has no observable performance overhead. The overhead for enforcing lifetime code integrity is similar to PerspicuOS.

3) Android benchmarks To measure the performance impact on user-mode programs, we used four standard Android benchmarks: AnTuTu, Geekbench, PCMark, and Vellamo. All of these benchmarks simulate typical real-world scenarios, including web browsing, video playback, photo editing, gaming, etc. The configurations we used are similar to LMBench. The

result is shown in Figure 4. As we can see, with KENALI's protection, the slowdown for these user-mode benchmarks is between 7% - 15%, which we think is acceptable.

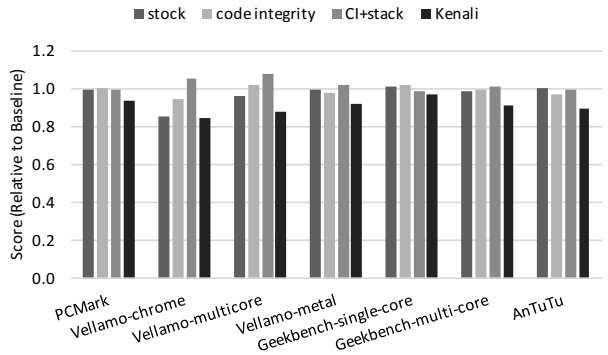

Fig. 4: Benchmark results from four standard Android benchmarks. Overhead in percentage of baseline performance, the higher the better. We used unmodified kernel compiled with clang as the baseline. Comparison configurations are (1) stock kernel, (2) kernel code integrity, (3) code integrity plus stack protection, and (4) full protection.

4) Memory overhead To measure the memory overhead introduced by KENALI (due to the use of shadow objects), we modified the `/proc/slabinfo` interface to report slabs with shadow objects. Based on this, we calculate how many additional pages are allocated and their percentage to the whole memory pages used by all slabs. We acquire this number at two time points: (1) after fresh reboot and (2) after finishing the AnTuTu benchmark. The result is shown in Table IV.

TABLE IV: Number of kmem_cache with shadow objects and the number of pages used by shadow objects.

	# kmem_cache	# pages	MB	% of total slab	% of total memory
Reboot	85	9945	38.85	65.11%	1.90%
Bench	85	9907	38.70	59.79%	1.89%

VII. DISCUSSION

In this section, we discuss limitations of our current design and implementation, insights we learned, and possible future directions.

Cross-platform. Although we choose AArch64 for the prototype, core techniques of KENALI are generic to most other commodity platforms. Specifically, data-flow isolation can also be implemented with the help of segmentation on the x86-32 architecture [67], WP-bit on the x86-64 architecture [24], and access domain on the AArch32 architecture [68]. For shadow objects, our current design is based on SLAB allocator, which is used by many *nix kernels like Solaris and FreeBSD. In theory, it can also be implemented on any memory pool/object cache based allocator. The rest two techniques, WIT and safe stack, are both platform-independent.

Better architecture support. A majority of KENALI overhead can be eliminated by having better hardware support. For example, with the application data integrity (ADI) feature from the SPARC M7 processor [49], (1) there would be no context switch for accessing distinguishing regions, and (2) all memory overhead introduced by shadow objects can be eliminated. With the kernel guard technology from Intel [30], enforcing lifetime kernel code integrity can be less expensive. We are exploring this direction as future work.

Reliability of assumptions. Our static analysis, INFERDISTS, relies on two assumptions: (1) there is no logic bug in the access control logic, and (2) there is no semantic error (i.e., failure of access control checks should always leads to returning corresponding error codes). For most cases, these assumptions usually hold, but may sometimes be violated [8, 18, 45]. However, we believe KENALI still makes valuable contributions, as it is an automated technique that can provide a strong security guarantee against memory-corruption-based exploits. In other words, by blocking exploits against low-level vulnerabilities, future research could focus on eliminating high-level bugs such as logical and semantic bugs.

Use-after-free. Our current design of KENALI focuses on spatial memory corruptions; thus it is still vulnerable to temporal memory corruptions, such as use-after-free (UAF). For example, if the `cred` of a thread is incorrectly freed and later allocated to a root thread, the previous thread would acquire the root privilege. However, KENALI still increases the difficulty of exploiting UAF vulnerabilities: if the wrongly freed object is not in distinguishing regions, exploiting such vulnerabilities cannot be used to compromise distinguishing regions. At the same time, many distinguishing regions data structures like `cred` already utilize reference counter, which can mitigate UAF. So we leave UAF mitigation as future work.

DMA protection. Since DMA can directly write to any physical address, we must also prevent attackers from using DMA to tamper distinguishing regions. Although we have not implemented this feature in KENALI yet, many previous works [55] have demonstrated how to leverage IOMMU to achieve this goal. Since IOMMU is also available on most commodity hardware, we expect no additional technical challenges but only engineering efforts.

VIII. RELATED WORK

In this section, we compare KENALI with a variety of defense mechanisms to defend memory-corruption-based attacks.

Kernel integrity. Early work on runtime kernel integrity protection focused on code integrity, including boot time [6] and after boot [9, 24, 55]. KENALI also needs to guarantee kernel code integrity, and our approach is similar to [24]. After rootkit became a major threat to the kernel, techniques have also been proposed to detect malicious modifications to the kernel [10, 13, 53, 54]. Compared to these works, KENALI has two differences: (1) threat model: rootkit are code already with kernel privilege, and their goal is to hide their existence; and (2) soundness: most of these tools are not sound on identifying all critical data (i.e., have false negatives), but our approach is sound.

Software fault isolation. Because many memory corruption vulnerabilities are found in third-party kernel drivers due to relatively low code quality, many previous works focused on confining the memory access capabilities of these untrusted code with SFI [16, 25, 43, 63]. A major limitation of SFI, as pointed out by [18], is that the core kernel may also contain many bugs, which cannot be handled by SFI.

Data-flow integrity. DFI is a more general technique than SFI, as it can mitigate memory corruptions from any module of the target program/kernel. KENALI differs from previous work on DFI enforcement [3, 15] in the following aspects. First, KENALI is designed for OS kernels, while previous work focused on user-mode programs; so KENALI requires additional care for the kernel environment. Second, previous work protects all program data, while KENALI aims at reducing the overhead by only enforcing DFI over a small portion of data that are critical to security invariants. Finally, as the effectiveness of DFI also depends on the quality of the point-to analysis, KENALI leveraged a more precise context-sensitive point-to analysis tailored for the kernel [13].

Dynamic taint analysis. DTA is similar to DFI and has also been used to prevent attacks [48]. The main difference is, in DTA, data provenance is defined by high-level abstractions like file, network, and syscall; but in DFI, data provenance is defined by the instruction that writes to the memory. For this reason, DTA is usually much more expensive than DFI, as it also needs to track the data provenance/tag for computation operations.

Memory safety. Besides DFI, another important defense technique is runtime memory safety enforcement. Theoretically, *complete* memory safety enforcement is more precise than DFI because it is based on concrete runtime information, but its performance overhead is also higher. For example, SVA [23] imposes a 2.31x - 13x overhead on LMbench. Generally, because the result generated by INFERDISTS is orthogonal to the runtime enforcement techniques, it can also be combined with memory safety. In this work, we chose DFI because we found that most accesses to distinguishing regions are safe; so DFI requires fewer checks and no metadata propagation for pointer assignments and arithmetic. However, if the overhead for memory safety enforcement becomes reasonable, e.g., with hardware assistant like Intel MPX [31], we can also switch the second layer protection from DFI to memory safety.

A recent memory safety work [36] demonstrated that, to defeat a certain type of attack (control-flow hijacking), it is sufficient to only protect a portion of data, thus reducing the performance overhead. While KENALI leveraged a similar idea, it addressed an important yet previously unsolved problem—what data are essential to prevent privilege escalation attacks.

Control flow integrity. Because a majority of real-world attacks are control-flow hijacking, CFI [1] has been a hot topic in recent year and has been demonstrated to be extensible to the kernel [22]. However, as discussed in §II-B, non-control-data attacks are both feasible and capable enough for a full privilege escalation attack. Furthermore, as CFI becomes more practical, attackers are also likely to move to non-control-data attacks. For these reasons, we believe KENALI makes valuable contributions, as (1) it can prevent both control-data and non-control-data attacks; and (2) compared to previous discussions on non-control-data attacks [20, 28], it provides an automated and systematic way to discover critical non-control-data.

IX. CONCLUSION

In this paper, we presented KENALI, a principled and practical approach to defeat all memory-corruption-based kernel privilege escalation attacks. By enforcing important kernel security invariants instead of individual exploit techniques, KENALI can fundamentally prevent all attacks. And by leveraging novel optimization techniques, KENALI only imposes moderate performance overhead: our prototype implementation for an Android device only causes 5-17% overhead for typical user-mode benchmarks.

ACKNOWLEDGMENT

We thank Ahmad-Reza Sadeghi and the anonymous reviewers for their helpful feedback, as well as our operations staff for their proofreading efforts. This research was supported by the NSF award CNS-1017265, CNS-0831300, CNS-1149051, and DGE-1500084, by the ONR under grant No. N000140911042 and N000141512162, by the DHS under contract No. N66001-12-C-0133, by the United States Air Force under contract No. FA8650-10-C-7025, by the DARPA Transparent Computing program under contract No. DARPA-15-15-TC-FP-006, by the ETRI MSIP/IITP[B0101-15-0644]. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, ONR, DHS, United States Air Force or DARPA.

REFERENCES

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *IEEE Symposium on Security and Privacy (Oakland)*, 2008.

[4] J. P. Anderson, "Computer security technology planning study," U.S. Air Force Electronic Systems Division, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), Tech. Rep. ESD-TR-73-51, 1972.

[5] Android Open Source Project, "Verified boot," https://source.android.com/devices/tech/security/verifiedboot/index.html.

[6] W. Arbaugh, D. J. Farber, J. M. Smith *et al.*, "A secure and reliable bootstrap architecture," in *IEEE Symposium on Security and Privacy (Oakland)*, 1997.

[7] ARM Limited, *ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile*. ARM Limited, 2015.

[8] K. Ashcraft and D. R. Engler, "Using programmer-written compiler extensions to catch security holes," in *IEEE Symposium on Security and Privacy (Oakland)*, 2002.

[9] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world," in *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[10] A. Baliga, V. Ganapathy, and L. Iftode, "Automatic inference and enforcement of kernel data structure invariants," in *Annual Computer Security Applications Conference (ACSAC)*, 2008.

[11] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," in *ACM EuroSys Conference*, 2006, pp. 73–85.

[12] J. Bonwick and B. Moore, "Zfs: The last word in file systems," 2007.

[13] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," in *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[14] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *Usenix Security Symposium*, 2015.

[15] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[16] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[17] H. Chen and D. Wagner, "MOPS: an infrastructure for examining security properties of software," in *ACM Conference on Computer and Communications Security (CCS)*, 2002.

[18] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek, "Linux kernel vulnerabilities: State-of-the-art defenses and open problems," in *Asia-Pacific Workshop on Systems (APSys)*, 2011.

[19] J. Chen, "Andersen's inclusion-based pointer analysis re-implementation in llvm," https://github.com/grievejia/andersen/tree/field-sens.

[20] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Usenix Security Symposium*, 2005.

[21] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, M. Negro, M. Qunaibit, and A. R. Sadeghi, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[22] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete control-flow integrity for commodity operating system kernels," in *IEEE Symposium on Security and Privacy (Oakland)*, 2014.

[23] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, "Secure virtual architecture: A safe execution environment for commodity operating systems," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[24] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested kernel: An operating system architecture for intra-kernel privilege separation," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[25] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "Xfi: Software guards for system address spaces," in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[26] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, "Missing the Point(er): On the Effectiveness of Code Pointer Integrity," in *IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[27] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[28] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits," in *Usenix Security Symposium*, 2015.

[29] R. Hund, C. Willems, and T. Holz, "Practical Timing Side Channel Attacks Against Kernel Space ASLR," in *IEEE Symposium on Security and Privacy (Oakland)*, 2013.

[30] Intel, "Intel kernel-guard technology," https://01.org/intel-kgt.

[31] ——, "Introduction to intel memory protection extensions," https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions.

[32] R. Johnson and D. Wagner, "Finding user/kernel pointer bugs with type inference." in *Usenix Security Symposium*, 2004.

[33] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kguard: Lightweight kernel protection against return-to-user attacks," in *Usenix Security Symposium*, 2012.

[34] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *Annual International Symposium on Computer Architecture (ISCA)*, 2014.

[35] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "sel4: Formal verification of an os kernel," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[36] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[37] C. Lameter, "Slub: The unqueued slab allocator," http://lwn.net/Articles/223411/, 2007.

[38] D. Larochelle and D. Evans, "Statically detecting likely buffer overflow vulnerabilities." in *Usenix Security Symposium*, 2001.

[39] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating return-oriented rootkits with return-less kernels," in *European Symposium on Research in Computer Security (ESORICS)*, 2010.

[40] J. Li, M. N. Krohn, D. Mazières, and D. Shasha, "Secure untrusted data repository (sundr)," in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[41] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang, "Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures." in *Network and Distributed System Security Symposium (NDSS)*, 2011.

[42] LLVM, "The LLVM compiler infrastructure project," llvm.org, 2015.

[43] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Software fault isolation with api integrity and multi-principal modules," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[44] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis." in *ATC Annual Technical Conference (ATC)*, 1996.

[45] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim, "Cross-checking semantic correctness: The case of finding file system bugs," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[46] MITRE, "Cve-2013-6282," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-6282, 2013.

[47] G. C. Necula and P. Lee, "Safe kernel extensions without run-time checking," in *Symposium on Operating Systems Design and Implementation (OSDI)*, 1996.

[48] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Network and Distributed System Security Symposium (NDSS)*, 2005.

[49] Oracle, "Introduction to sparc m7 and application data integrity (adi)," https://swisdev.oracle.com/_files/What-Is-ADI.html.

[50] M. Paolino, "ARM TrustZone and KVM Coexistence with RTOS For Automotive," in *ALS Japan*, 2015.

[51] PAX, "Homepage of the pax team," https://pax.grsecurity.net/, 2013.

[52] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *IEEE Symposium on Security and Privacy (Oakland)*, 2008.

[53] N. L. Petroni Jr, T. Fraser, A. Walters, and W. A. Arbaugh, "An architecture for specification-based detection of semantic integrity violations in kernel dynamic data." in *Usenix Security Symposium*, 2006.

[54] N. L. Petroni Jr and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[55] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[56] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-vm monitoring using hardware virtualization," in *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[57] S. Smalley and R. Craig, "Security enhanced (se) android: Bringing flexible mac to android." in *Network and Distributed System Security Symposium (NDSS)*, 2013.

[58] A. Srivastava and J. Giffin, "Efficient protection of kernel data structures via object partitioning," in *Annual Computer Security Applications Conference (ACSAC)*, 2012.

[59] "System error code," https://msdn.microsoft.com/en-us/library/windows/desktop/ms681382%28v=vs.85%29.aspx, 2001.

[60] The IEEE and The Open Group, `errno.h` *- system error numbers*. The Open Group, 2013, the Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2013 Edition, http://pubs.opengroup.org/onlinepubs/9699919799/functions/rename.html.

[61] The Linux Foundation, "Llvmlinux," http://llvm.linuxfoundation.org/index.php/Main_Page.

[62] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities." in *Network and Distributed System Security Symposium (NDSS)*, 2000.

[63] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *ACM Symposium on Operating Systems Principles (SOSP)*, 1994.

[64] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Improving integer security for systems with kint." in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[65] Z. Wang and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[66] J. Yang, T. Kremenek, Y. Xie, and D. Engler, "Meca: an extensible, expressive system and language for statically checking security properties," in *ACM Conference on Computer and Communications Security (CCS)*, 2003.

[67] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *IEEE Symposium on Security and Privacy (Oakland)*, 2009.

[68] Y. Zhou, X. Wang, Y. Chen, and Z. Wang, "Armlock: Hardware-based fault isolation for arm," in *ACM Conference on Computer and Communications Security (CCS)*, 2014.