COMPUTERS & SECURITY 74 (2018) 130–143

Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Domain Isolated Kernel: A lightweight sandbox for untrusted kernel extensions

Valentin J.M. Manès [a], *Daehee Jang* [b], *Chanho Ryu* [a], *Brent Byunghoon Kang* [b,*]

[a] *Cyber Security Research Center (CSRC), 291 Daehak-ro, Yuseong-gu, Daejeon, 34141, Republic of Korea*
[b] *Korea Advanced Institute of Science and Technology (KAIST), 291 Daehak-ro, Yuseong-gu, Daejeon, 34141, Republic of Korea*

ARTICLE INFO

Article history:
Received 20 July 2017
Received in revised form 4 December 2017
Accepted 2 January 2018
Available online 17 January 2018

Keywords:
Kernel
ARM
DACR
Rootkit
Software vulnerability
Extension

ABSTRACT

Monolithic kernel is one of the prevalent configurations out of various kernel design models. While monolithic kernel excels in performance and management, they are unequipped for runtime system update; and this brings the need for *kernel extension*. Although kernel extensions are a convenient measure for system management, it is well established that they make the system prone to rootkit attacks and kernel exploitation as they share the single memory space with the rest of the kernel. To address this problem, various forms of isolation (e.g., making into a process), are so far proposed, yet their performance overhead is often too high or incompatible for a general purpose kernel. In this paper, we propose Domain Isolated Kernel (DIKernel), a new kernel architecture which securely isolates the untrusted kernel extensions with minimal performance overhead. DIKernel leverages hardware-based memory domain feature in ARM architecture; and prevents system manipulation attacks originated from kernel extensions, such as rootkits and exploits caused by buggy kernel extensions. We implemented DIKernel on top of Linux 4.13 kernel with 1500 LOC. Performance evaluation indicates that DIKernel imposes negligible overhead which is observed by cycle level microbenchmark.

1. Introduction

Most of the current commodity operating system, like Linux, FreeBSD, and arguably Windows are monolithic architectures in which all the kernel is in a single address space. Even if some had predicted the ascendance of other kernel designs, monolithic kernels have passed the test of time. They are performant and provide ease of resource management resulting from a simpler design.

Nonetheless, as the kernel components share same execution privilege level and memory access permission, attackers can compromise the entire system by finding the least secure portion of kernel components. In general, the weakest chain of the kernel components is known to be *Kernel Extensions*[1]. For example, CVE-2017-2636, 2017 and CVE-2016-2384, 2016 demonstrate an attacker getting system root privilege by exploiting a small bug in the kernel extensions. Similarly, the monolithic kernel design gives advantage to malicious kernel extension – *rootkit* – for manipulating the entire operating

* *Corresponding author.*
E-mail address: brentkang@kaist.ac.kr (B.B. Kang)
https://doi.org/10.1016/j.cose.2018.01.009

[1] A kernel extension is code dynamically added to the kernel after its boot. It is also known as Loadable Kernel Module (LKM) in Linux, Device Driver (.sys) in Windows, or Dynamic Kernel Linker (KLD) in FreeBSD. In this paper, we use the term *kernel extension* as representative.

system. Once a rootkit is installed as a kernel extension, system data structures such as process control block and inode[2] are easily accessible by attacker.

Considering that kernel extension has high potential of being compromised by attackers, we find that it is important to isolate their execution privilege and memory access permission from the rest of the kernel components; thus we propose *Domain Isolated Kernel* which separates the kernel extensions from the rest of kernel by leveraging the *Domain Access Control Register* hardware feature in ARM architecture (ARM domain access control, 2001).

Previous research on extensions used isolation primarily to improve their reliability and fault isolation (LeVasseur et al., 2004; Nikolaev and Back, 2013; Seltzer et al., 1996; Swift et al., 2003). However, recently, efficient in-process memory isolation for user space and its application has been demonstrated by ARMlock (Zhou Y. et al., 2014) and Shred (Chen et al., 2016). Domain Isolated Kernel (DIKernel) is the first approach of utilizing this memory separation technique against kernel extensions. The high-level concept of DIKernel is not new, however, we found several non-trivial challenges while separating the kernel. The details regarding design issues and the challenges will be discussed in design in Section 3.

In this paper, we assume the kernel extension as the source of attacks; and specifically aim to fortify the base kernel by deprivileging the memory access permissions of such extension codes. Kernel extensions have often been found not to be carefully written and reviewed relatively compared to the kernel mainline code, thus likely to contain security vulnerabilities. In the case of Linux, kernel extensions are known to be around 70% of the code base and could contain around 2/3 of its vulnerabilities (Chen et al., 2011; Xu et al., 2004). Hence, isolating them from the main kernel component can make the system more secure from various attacks such as kernel exploitation or rootkits (Adore-NG 0.41, 2004; Knark 2.4.3, 2001).

Isolation of kernel extensions requires handling when there is a control flow transition between base kernel code and kernel extension codes. Depending on the usage model of kernel extension, we can enumerate several entry/exit points between base kernel and extension. DIKernel interposes such control flow transitions by redirecting kernel APIs (extension to base kernel) and call back functions (base kernel to extension) to a special trampoline code (DI-switcher) that changes the memory domain. In short, the design of DIKernel restricts direct memory access and control flow transition between base kernel and kernel extensions, and provides a secure channel between two entities.

We implemented our prototype of DIKernel on Linux 4.13.11. Although our prototype implementation is based on Linux, we expect the same concept and design principle could be extended to other commodity operating systems.

The contributions of this paper can be summarized as follow:

- First approach of using ARM's hardware-based domain isolation technique against untrustworthy kernel extensions.
- In-kernel code and data isolation without using memory virtualization.
- Demonstrate a new kernel organization strategy. DIKernel protects the base kernel (code and data) from malicious extension modifications despite both running at the highest privilege level, through the use of efficient memory access restriction.
- Provide an algorithm that considers code-reuse attacks to securely switch between two domains.
- Implementation and Evaluation of DIKernel to demonstrate its efficiency and security.

This paper is organized as follows: Section 2 discusses the problem space DIKernel aims to address. Then we present our system design and implementation in Section 3 and Section 4. We evaluate its performance and examine the security of our system in Section 5. We contrast the related work in Section 6 and discuss the limitations of DIKernel and how it could be improved in the future in Section 7. Section 8 concludes our paper.

2. Problem overview

This section presents the rootkit problem and the requirements we have for a practical solution. It then discusses existing solutions and how they don't fulfill these requirements. Then we define DIKernel assumptions and threat model.

2.1. *Rootkit installation*

The term "rootkit" denotes a set of tools which is used with malicious intent to gain access to the system without the knowledge of the administrator. The main purpose of a rootkit is making a stealthy backdoor or neutralizing anti-virus software by covertly changing the kernel code and data.

Inline hooking: redirects a legitimate kernel function to another one. This is done by modifying kernel code at the beginning and/or end of the targeted function.

Function Pointer Hooking: modifies control data (i.e. return addresses or function pointers) to redirect the normal control flow of the kernel. A very common example is the modification of the system call or interrupt tables so that whenever a specific system call or interrupt is triggered, the control flow is redirected to the attacker will.

Direct Kernel Object Modification: modifies kernel data to deceive the system and hide himself from detection. For example, modifying the user ID information inside `task_struct` kernel object effectively changes the security privilege of a process. Similarly, manipulating the `inode` kernel object allows to hide system resources such as files and process from the viewpoint of various application tools (e.g., `ps`, `ls`) which obtains system information via the filesystem interface.

These three techniques commonly used by rootkits all require modification of code or data that are needlessly accessible to extensions.

Various attack cases suggest that the simple design of monolithic kernels gives advantages to attackers mounting their exploits. The single shared address space system design is evidently not helpful in terms of mitigating attacks.

[2] Inode is a Linux specific data structure concerning the file system that contains critical information such as file owner, group and access permissions.

2.2. Secure extension isolation: Requirements

A solution to kernel extension exploits and rootkit attacks, is to isolate extensions from the base kernel to protect sensitive code and data. However, building boundaries in a system that was not designed with them incurs three drawbacks. A practical solution needs to minimize them:

R1 - Low Overhead: Creating a boundary means the system will need to cross it. Since the performance of kernel is the major factor that affects the overall system, any modification that involves high performance overhead is unacceptable. Our primary design goal aims to avoid such performance overhead and consider practicality and deployability.

R2 - Compatibility with existing code: Isolating components for new design often involves creating an interface for them to communicate; which might require modification of legacy extension codes. Extension code is often undocumented (in case of third party codes) and is hardware specific so that it is not easily modifiable. A practical solution must be easy to deploy in real-world. Hence, this interface must be invisible for extensions so that their code does not require to be modified. It also demands that no additional hardware needs to be installed.

R3 - Robustness against attacks: Rootkit and kernel exploit attacks corrupt/manipulate various data inside the kernel. In order to make a robust defense system, all the possible attack vectors should be carefully addressed. Our goal is to protect the critical kernel data even under the presence of rootkit and exploits in extensions by de-privileging and sanitizing their memory access capability against base kernel memory region.

2.3. Existing solution

Several methods or mechanisms are proposed to mitigate rootkit attacks. However, they often fail to meet some of the above requirements.

A large body of research (Azab et al., 2014; Jiang et al., 2007; Lee et al., 2017; Payne et al., 2008; Petroni and Hicks, 2007; Petroni et al., 2004; Rhee et al., 2009) has been dedicated to rootkit detection. These tools may be placed in kernel space, at the same privilege level as rootkits. Since detection techniques do not prevent infection from happening, but intend to detect and sanitize it afterwards, attackers have the opportunity to directly target and disable them, as shown recently (Phrack article, 2016).

Some protection tools can also run in another address space, based on hardware as TZ-RKP (Azab et al., 2014), or based on virtualization as Lares (Payne et al., 2008). While they are safely isolated from the kernel (which is discussed in Section 6), these methods add a context switch between the kernel space and their higher privilege system components, which requires a significant mode switching overhead.

In the case of signature or behavior based detection, attackers often bypass the defense with small efforts of transforming their signature, or using transient timing attacks.

Integrity monitoring tools like Copilot (Petroni et al., 2004) and KI-Mon (Lee et al., 2017) require extra hardware, or only cover the integrity of specific regions like HookSafe (Wang et al., 2009). Furthermore, even if detected, some attacks need system reboot or hardware replacement to be reverted (Zaddach et al., 2013). Running only pre-approved code effectively neutralizes kernel malwares (Riley et al., 2008; Seshadri et al., 2007), but suffers from compatibility issues.

Other extension isolation approaches have focused on the reliability of the system. They focus on avoiding fatal system failure caused by unintended errors in kernel extension codes rather than securing the system in terms of malicious attacks caused by software vulnerabilities (Bershad et al., 1995; Herder et al., 2009; Seltzer et al., 1996; Zhou et al., 2006). DD/OS and VirtuOS (LeVasseur et al., 2004; Nikolaev and Back, 2013) isolate kernel extensions by placing them inside a specially dedicated address space using memory virtualization techniques, however they suffer from relatively slow performance.

The major advantages of DIKernel are that it is a preemptive defense solution with low performance cost (domain switching is cheaper than context switch or memory virtualization), and it does not require external hardware add-on. We achieve such benefits by utilizing the domain management system provided in ARM architecture.

2.4. Assumptions and threat model

We assume that the CPU of the system provides support for ARM domain access feature (ARM domain access control, 2001) (currently supported by ARMv6 and ARMv7). I/O Memory Management Unit (IOMMU) is assumed present. The base kernel is the kernel after its initialization is complete, before any code is dynamically loaded as modules.

We consider kernel extensions have vulnerabilities thus considered as the source of attacks and assume the base kernel is intact. We assume the kernel is loaded intact with a secure boot mechanism such as UEFI (Unified, 2010). Every component in user space and any dynamically loaded kernel module, but the switcher module (special trampoline code of DIKernel design, described in Section 3.2) is regarded as malicious. The attacker in our threat model is capable of running existing kernel codes only by first calling the special trampoline area (referred as DI-switcher) but not the codes of other kernel code section (.text) as they are inaccessible by domain isolation. Also, the malicious kernel module is assumed *not to include any instructions that can change CPU domains* or the register pointing to the top of the page tables inside its code; which is a plausible assumption in ARM as RISC architecture does not allow unaligned instruction (Cortex-A7 MPCore technical reference manual, 2011), thus we can easily scan entire codes before loading it inside the kernel.

Any external hardware device apart from the CPU, the memory controller and system memory chips, are considered to be potentially under the attacker control. The trusted computing base (TCB) is reduced to the base kernel.

Devices are sometimes able to corrupt each other by the communication channel between themselves, or by the IOMMU buffers that are sometimes shared. In this way, extensions could attack each other bypassing any in-kernel protections. We exclude protection between different extensions in our defense design. Finally, our threat model does not include Denial of Service (DoS) attacks.

3. Design

3.1. Background: Domain access control

Domain Access Control Register (DACR) is a rarely considered ARM CPU feature. It is a per core 32-bit privileged register. Since it is a per core register, domain separation natively supports parallel computing. Being a privileged register, it can only be accessed in supervisor mode.

As shown in Fig. 1, DACR is divided into 16 2-bit fields, each field corresponding to a memory domain. The corresponding memory of a domain is set by the domain ID, a 4-bit flag in the Page Directory Entry (PDE). Thus, each domain is divided into 1 MB chunks. The 2-bit fields of the DACR register can have 4 values: No Access (`0b00`), Client (`0b01`), Reserved (`0b10`) and Manager (`0b11`). Any access to a "No Access" memory area will trigger a domain fault. Client means that the memory access will happen as usual, it lets the page table to determine the access right; this is the default setting. Manager mode ignores permission bits in the page table and allows unlimited access. Reserved mode is not used.

Upon a memory access, the Memory Management Unit (MMU) checks the domain which the requested memory belongs to, and its access permission in the DACR register. Then, it determines whether or not the access should be allowed based on the access level and the permission bits in the page tables. The manager mode is used by default in Linux when switching to privilege mode so as to bypass the permissions bits check for optimization.

The big benefit of using memory domains is that switching between them is a very quick process: update the DACR register suffice to close a domain (setting it to No Access) and to open another one (setting it to Client or Manager mode). It only involves one instruction and doesn't require any memory updates or Trusted Lookaside Buffer (TLB) flushing.

By default, domain 0 is reserved for kernel space, domain 1 for user space and domain 2 for I/O memory. We additionally use domain 3 for the DI-switcher (describe below, Section 3.2) and extensions.

3.2. DIKernel switcher

DIKernel is implemented by inserting a layer between the base kernel and its extensions. It intercedes all interactions between the extensions and the kernel to enforce isolation and compatibility with existing extensions. We refer to this special layer as *DI-switcher*.

DI-switcher is a module loaded at the end of the kernel initialization. All its code and data are placed in the same domain as other extensions. This domain is always set in client or manager mode. DI-switcher code is critical in terms of security since it is always accessible by attackers; it can be potentially exploited for code reuse attacks, thus allows attackers to execute forbidden instructions (regarding DACR manipulation). Therefore, DI-switcher code needs to be carefully designed considering various possibilities of being abused by code reuse attacks. We later demonstrate that our design and implementation of DI-switcher guarantee the safety against such attacks in Section 3.5.

DI-switcher has three roles. First, it enforces **memory access isolation** between the base kernel and the extensions. It opens and closes the different domain to protect the code and data of the base kernel. An extension should never be able to directly access anything from the base kernel without using the proper access channels.

Isolation of the code and static data memory (global variables) is achieved by manipulating the page tables and mapping the pages into domain 3 while loading the extension image into the kernel.

In order to isolate the kernel stack between extensions and base kernel, we spawn a kernel thread for potentially unsecure extensions and impose domain 3 for the new stack. In particular, DIKernel introduces additional stack layer (kernel extension stacks) on top of existing system stacks such as user stacks, kernel stacks, and interrupt handler stacks. The

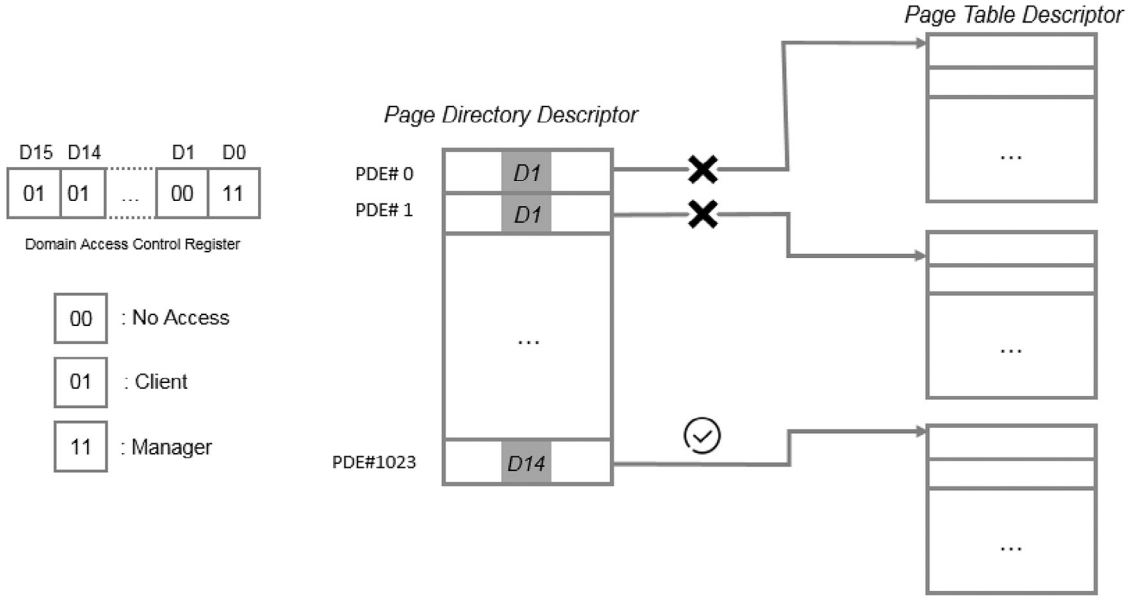

Fig. 1 – ARM Domain Access Control Register use example.

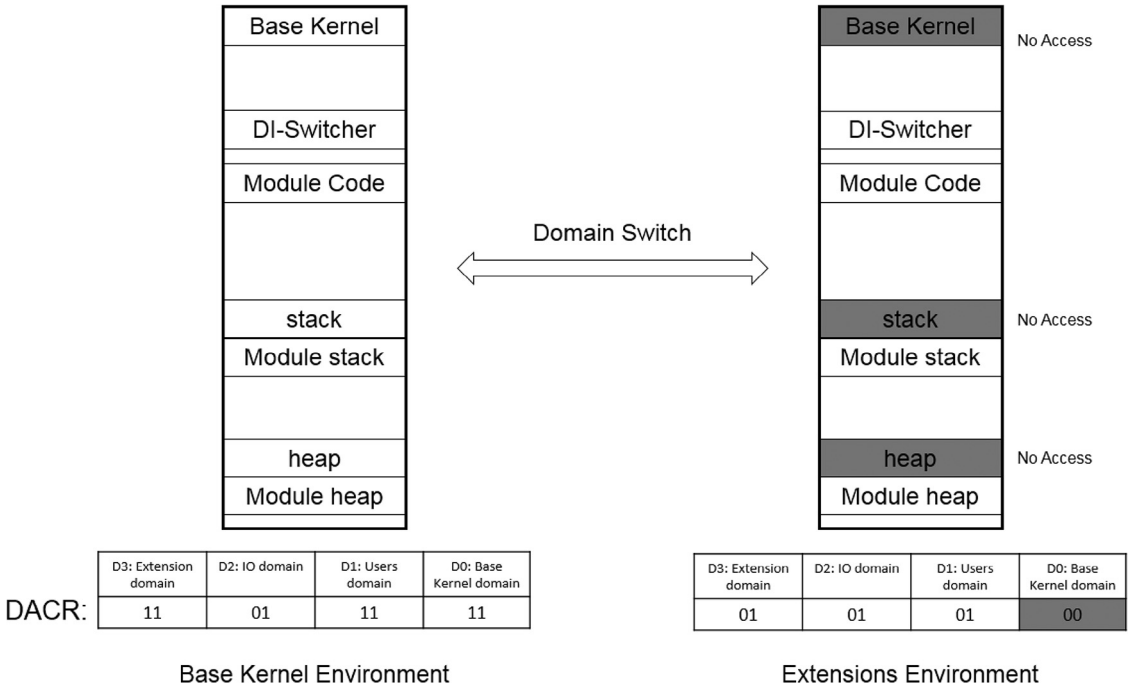

Fig. 2 – Domain Isolated Kernel approach overview.

DI-switcher is responsible for the kernel thread management and assignment to extensions. Isolating the heap into domain 3 can be accomplished by using standard methodologies (Akritidis, 2010) for changing the dynamic memory allocator. Once memory isolation is achieved, an interface is required to cross the boundary in both ways; from base kernel to extensions and vice versa.

The second role of the DI-switcher is **interposition**, it supervises all inter-domain control transfers and provides a secure interface between base kernel and extensions. Whenever the base kernel wants to call some extension code, it first calls the DI-switcher in order to close the base kernel domain, then jumps to the targeted extension code. The reversed way works similarly, the extension first calls the DI-switcher that will open the base kernel domain and then jump to it. These calls are made through wrappers described below (Section 3.3).

Kernel APIs such as `kmalloc` or `vmalloc` are specially handled in DI-switcher in order to use the isolated heap memory. Overall, entire memory access of the extension is isolated in a different domain from the base kernel: its code, stack and heap. As can be seen in Fig. 2, base kernel domain is closed while extensions are executed so that they are not able to directly access any of the base kernel code or data. Therefore, DIKernel prevents techniques seen in Section 2.1.

However, in some cases, violating the security protection of DIKernel could be considered as a normal behavior. For example, there are some edge cases in special kernel extensions which requires to directly access some kernel data structures. Since such circumstances are rare (Swift et al., 2003), we can register such extensions to let the domain fault handler exceptionally allow it.

Sensitive data like `task_struct` are allowed to be read by extensions, but not modified since there is no need for it. Dedicated memory is allocated for such data and the structure is copied on demand; thus allowing the read access, but preventing the write attempt against original data structure.

The DI-switcher also interposes interrupts to guarantee that there is no crash in case the base kernel is closed when the interrupt occurs. It opens the base kernel, executes the targeted code, and then writes the DACR register back as it was before the interrupt occurred.

The third role of the DI-switcher is to ensure **compatibility** with existing extension code. Redirecting the kernel APIs by hooking the kernel symbols, and changing the stack, heap into the isolated memory region happens transparently without altering any existing codes or logics of the extension.

Since the DI-switcher is one of the core components of DIKernel design, it is important to prevent this module from being deliberately unloaded by rootkits. We insert a check in the `delete_module` kernel API to prevent the DI-switcher removal in any case. One can imagine that unloading a kernel module can also be accomplished by sophisticatedly manipulating metadata of kernel modules, or manipulating the page table mappings. However, we note that basic security enforcement of DIKernel preempts to stop such manipulation attempts as it isolates all such memory regions into isolated domain.

3.3. *Wrapper*

Wrappers are used to interpose kernel functions that can be called by extensions. Before and after calling functions in the base kernel or in an extension they will update the DACR register according to our access policy. The base kernel domain is always closed when untrusted extension code is executing. On the other hand, all the domains are open while the base kernel code is executing.

Wrappers also encapsulate kernel calls into extension. In this case, the kernel code has to be directly modified to call a

wrapper in the DI-switcher. For example, this is the case for module init and exit function, or for the functions that are registered for drivers.

Most of the wrapper code is shared, but as mentioned above, some wrappers occasionally need to execute extra operations if they involve passing kernel data or dynamic memory allocation.

The entry and exit gate is the key of domain transitions since they update the DACR value. There is only one such gate for each way and they are shared among all wrappers. We emphasize that the entry gate is the only entry point from the extensions to the base kernel. This is an important design point for preventing code reuse attack; we explain more details below.

3.4. *Secure switching*

The entry and the exit gates are critical to the enforcement of the isolation between the base kernel and extensions. The gate code is considered always exposed to attackers since it is in the same domain as extension. No data in the extension domain can be trusted. The DI-switcher has to be developed with this consideration and the gates in particular.

For the switching between extensions and the base kernel to be secure, the switching mechanism must prevent any extension code from regaining control while the base kernel domain is open. Therefore, the exit gate is not the main concern since it doesn't give access to the base kernel. On the other hand, the entry gate must be deterministic and exclusive in order to be secure.

3.4.1. *Deterministic execution*

The entry gate semantic is controlled and cannot be changed by any input. Since it is always exposed to extensions, its execution must not trust any input and lead to a unique outcome. The possible values for updating the DACR register is hardcoded in the code, which prevents attackers from corrupting it with arbitrary value.

3.4.2. *Exclusive access to the base kernel*

The entry gate must be the only way to enter the base kernel. Opening the base kernel domain implies writing the DACR register which is done with a particular instruction. We thus make sure this instruction is not present in any extension. This is done through de-privileging extensions, which is detailed in Section 3.6.

3.4.3. *Atomic domain transition*

In case an attacker can control the interrupt handler, the domain transition process must be atomic. Otherwise, an attacker can jump into an instruction which changes the domain; then interrupts the execution flow immediately before the confirmation routine is executed. However, the attack model of this paper does not allow the attacker to manipulate the interrupt handling.

The essence of DIKernel is isolating the potentially untrusted kernel extensions from the base kernel. The basic motivation of this paper stems from the fact that dynamically loaded extensions are relatively insecure than the well reviewed base kernel code. Assuming such attack model, we can trust the interrupt handler codes, therefore the privilege transition of DIKernel does not have to be atomic as our attack model does not involve malicious interrupts. Unlike other approaches that suppose high privilege infection without assuming address space isolation like nested kernel (Dautenhahn et al., 2015) and SKEE (Azab et al., 2016), we don't have to worry about the atomicity of the gates because the interrupt handling is trusted and relevant data structures are isolated from in place.

3.5. *Secure gates*

As said in Section 3.4, gates have to be designed carefully because they are always exposed to extensions. The implementation of DIKernel must guarantee to disallow the extension codes from accessing the DACR and TTBR registers. Such registers can only be changed by particular instructions. As detailed in Section 3.6, DIKernel does not let this instruction to be loaded in any extension code or to be generated at runtime. Therefore, the two gates are the only locations outside of the base kernel where this instruction is present.

DIKernel have to prevent an attacker from jumping to this code with his own crafted DACR value to be loaded. To do so, the DACR value should be fixed, written in the code. Since the `MCR` instruction requires a general-purpose register (`R0-R12`) to contain the value loaded in the DACR, it is not possible to write a single instruction that updates DACR with a fixed value. As DACR update requires multiple instructions (1. setup DACR value in a register, 2. execute `MCR` instruction), an attacker can set up an arbitrary DACR value in a register and directly jump into the `MCR` instruction, then regain the control flow; which will effectively bypass the DIKernel isolation. To prevent such code-reusing attack, the register containing the DACR value is compared with a hardcoded number immediately after `MCR` instruction is executed. If the register value does not match to the hardcoded number, execution flow jumps back to the beginning of the DACR update code until the updated DACR value is confirmed to be correct. Therefore, an attacker cannot regain the execution flow if the changed DACR value is different from the fixed value.

Details can be seen in the code snippet below. Lines 3–4 is the instruction loading the DACR value. After the DACR is written (line 5), we compare it to a fix value (line 8) and go back to writing the DACR of the value loaded is not the expected one (line 10).

Another important design for DIKernel security is that an attacker should never be allowed to hijack the execution flow between entry and exit gate. In order to guarantee this, DI-switcher implementation does not use any data that affects the control flow dynamically (e.g., return address from the stack, or function pointer), the execution path from entry/exit gate and base kernel/extension is hardcoded thus deterministic. Hence, extensions should not be able to regain the control flow between the entry gate and the following exit gate call by manipulating data structure that affects the control flow.

As can be seen in Fig. 3, there are two cases where the entry gate can be called. In the first one, an extension calls a base kernel function, the entry gate is called, the function is

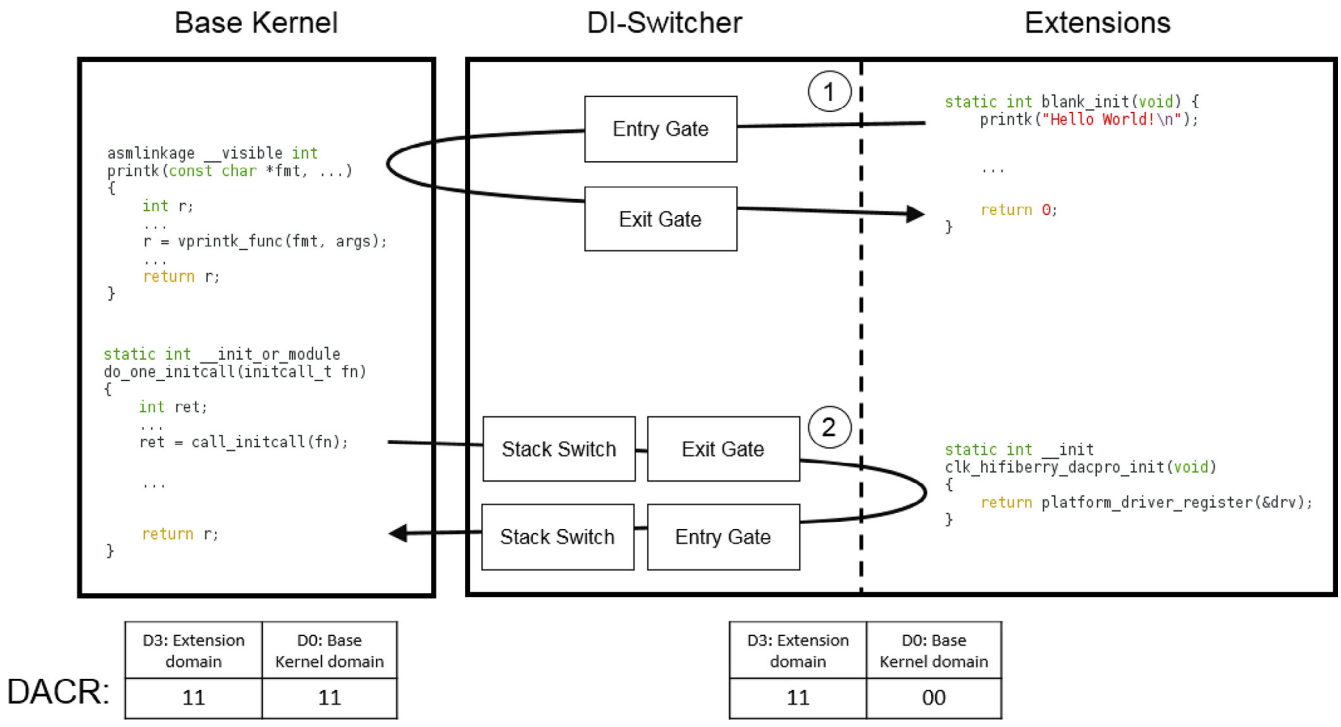

Fig. 3 – Entry and exit gates two usage cases. In the first one, a kernel extension calls a base kernel function. The base kernel then returns to the extension. In the second case, the base kernel calls an extension, then returns to the base kernel. Both calls are interposed by the DI-switcher to modify the DACR value.

executed and then returns to the extension via the exit gate. In this case, extensions do not have the opportunity to corrupt any of the base kernel data that could be corrupted to redirect the base kernel control flow.

In the second case, the base kernel is executing, then calls an extension that executes and then returns to the base kernel. This case has to be carefully treated. If the base kernel and the called extension would share the same stack, the extension would have the opportunity to corrupt control data the base kernel would use when called back. Hence, the extension would be able to redirect the control to itself while the base kernel domain is open. Therefore, before the exit gate is called we switch to another stack for the extension to be run on, and execute the called extension function. When the base kernel is called back, the stack is switched back to its original. We note that DACR is a per core register, therefore an extension running on one core cannot corrupt the base kernel data running in another core.

To avoid relying on the extension stack, in particular for control data as return addresses, the entry gate is defined as a macro rather than a function. In case a memory pointer provided from extension must be used, we sanitize such pointer by checking its first level descriptor in the page tables, to check the value of its domain.

Control flow redirection based on interrupt mechanism is considered to be safe in this paper as we prevent extensions from changing the interrupt handler. Therefore, we consider interrupts received while a gate is executing are not a threat to the enforcement of DIKernel isolation.

Listing 1: Entry Gate

```
1  <entry_gate>:
2  push    {r8, r9}
3  movw    r8, #21855      ; 0x55df
4  movt    r8, #21845      ; 0x5555
5  mcr     15, 0, r8, cr3, cr0, {0}
6  movw    r9, #21855      ; 0x55df
7  movt    r9, #21845      ; 0x5555
8  cmp     r8, r9
9  pop     {r8, r9}
10 bne     <entry_gate>
11 isb     sy
```

3.6. *De-privileging extensions*

DIKernel relies on the ARM domain feature to isolate extension from the base kernel. As detailed in Section 3.1, this feature uses bits in the page tables and CPU per core registers, DACRs. Both abilities need to be removed from extensions so that their isolation is enforced.

Extensions are not able to modify page tables by design since they are in the base kernel area. In order to prevent extension codes from changing the domain, we restrict the access to DACR and TTBR register by scanning the extension code before loading it into the kernel. If it does, the extension loading is rejected. Also, we forbid the extension from generating executable code at runtime by restricting the use of page permission of related kernel APIs. We refer to these protection mechanisms as *extension de-privileging*.

De-privileging extensions does not require any alteration of their code. The extensions do not have to be recompiled in order to enforce DIKernel security policies.

Scanning specific instructions from extension is fairly straightforward task as the ARM architecture uses fixed length instruction encoding. This makes sure that a non-malicious extension cannot inadvertently contain an unwanted instruction as it could happen in x86 architecture that allows unaligned access.

Although an extension passes the initial scanning of restricted instructions (DACR manipulation, etc.), an attacker could try to generate his own code containing such instruction at runtime. DIKernel handles such attempts by ensuring the kernel extension does not obtain any additional executable pages once it has been loaded into the kernel. As the page tables are part of the base kernel domain (Domain 0) any extension code running in the isolated domain cannot directly modify the page table without alarming the base kernel's domain fault handler. In case the extension uses kernel APIs such as `do_mmap`, the DI-switcher monitors the requested page permission and forbids the extension from having a newly executable page.

3.7. Attack vectors against DIKernel

DIKernel goal is to ensure extensions cannot modify any of the base kernel data (except special access created for wrappers). We want to make sure an attacker cannot bypass the DIKernel isolation scheme in any way.

We identified six possible ways to bypass DIKernel:

- Modifying the system call or interrupt table to gain control over the system while the base kernel domain is opened.
- Since ARM domain access control technology that we are using for isolation entirely relies on page tables, an attacker could create fake page tables and point to them as the used page tables.
- Updating the DACR to open the base kernel domain.
- Regaining the control flow post entry gate by modifying control data (i.e. return address, function pointer) while the extension is executed.
- Injecting newly crafted code that could modify DACR or redirect the top of the page tables so as to open the base kernel domain.
- Finally, an attacker could just unload the DI-switcher module so that isolation is not enforced.

3.7.1. System call or interrupt table modification

It is a common technique to install itself in a system and a convenient way to set up specific call back. However, these tables are marked as read-only in the page tables. Hence, attackers need to modify the page tables in order to be able to modify the system call or interrupt tables. DIKernel protects page tables from direct modification and since all kernel APIs are interposed, it is also able to check the arguments of functions, as `do_mmap`, that are able to modify the page tables.

3.7.2. Fake base tables

The page tables are pointed by the TTBR register which is read and written by a privileged instruction. An attacker could craft his own page tables and change the TTBR to point to it (Jang et al., 2014). However, we remove the extension ability to access the TTBR register as mentioned in Section 3.6.

3.7.3. DACR update

As above, DACR is read and written using a privilege register which we remove access from extensions.

3.7.4. Regaining control post entry gate

It would defeat the whole purpose of DIKernel if an extension was able to regain the control flow after the entry gate is executed and the base kernel domain is opened. To prevent this, as detailed in Section 3.5 we avoided using any control data after the entry gate as much as possible. When control data is used, it is placed in the base kernel domain and only a pointer to it is let under the extension access. Once the base kernel is opened, we verify it is directed toward data in the base kernel domain by reading the page table.

3.7.5. Code injection

Since we prevent extensions to run certain instructions, an extension could try to create data and mark it as executable. As for the system call or interrupt tables modification attack, it requires to modify the page tables. This capability is removed from extensions. We assume ARM PxN (Kernel.org, 2014) is enabled, thus it is not possible for extension to execute user code in privilege mode.

3.7.6. Unloading DI-switcher module

This has been disallowed by inserting a check in the base kernel code. Otherwise, an extension could force a situation where the base kernel domain would not be closed before extensions are called.

4. Implementation

Our implementation is based on Linux version 4.13.11, and tested on a Raspberry Pi 2 model B, which has a Broadcom BCM2836 chip with a 900 MHz quad-core ARM Cortex A7 (ARMv7) processor.

4.1. Linux patch

While our design does not require modification of kernel extension codes, additional implementation on top of the main kernel is required with minimal alteration of existing code. Indeed, DIKernel can be considered as any other Linux kernel update. Its modification does not rely on a specific kernel feature and would be easy to adapt to another Linux version or even another operating system like BSD. DIKernel code is mostly gathered in new `dikernel/` and `drivers/diswitcher/` subdirectories, and modification to the existing Linux code is made to call our newly created code (Table 1).

The main kernel modification has two goals. First, we modify the two system calls that can be used to load a module in kernel space, `init_module` and `finit_module`, so that when the module code is allocated, its domain ID is modified.

Table 1 – Number of added or modified line in Linux kernel code.

arch/arm/	2
include/linux/	2
init/main.c	6
kernel/module.c	5
drivers/base/	1
Total mainline modifications	17
include/linux/dik/	75
dikernel/	571
drivers/diswitcher/	929
Total appended lines	1575

Moreover, when the kernel calls into extension, we need to modify these calls to go through the DI-switcher first so that it can close the kernel domain before executing the extension codes. In particular, the `init` and `exit` functions of every module need to be called this way.

The initial prototype of DIKernel was adopted against Linux 4.1.13 which was the latest version at the time. Final implementation of DIKernel uses Linux 4.13.11 which is the latest stable version at the time of publishing this paper. Current version of ARM Linux kernel, which the final version of DIKernel is adopted, fully supports DACR security features including user space memory access restriction which lacked before 4.3.

4.2. *Wrappers*

In the general Linux environment, symbols are used as an interface for accessing extension code or data from base kernel. Symbols are variables and functions that are exported. The kernel dynamically retrieves them from the extensions when they are loaded.

Since, the base kernel domain is closed while they are executing, extensions cannot directly call base kernel functions without causing a system failure. Thus, for extensions to be able to access kernel API, symbols are modified as to point to point to a wrapper in the DI-switcher. Wrappers are in charge of opening the base kernel, calling the originally intended base kernel argument, closing the base kernel domain back, and passing return argument to extensions. Other than the additional support for user space memory access restriction issue (which does not involve the goal of this paper), the rest of the domain related features related to this paper are not changed by kernel version.

When the return argument is a pointer to base kernel data, we have to allocate memory, change its domain and copy this structure into the allocated memory and return a pointer to this copy to the extension. Hence, extensions can have a read access to base kernel data, but no write access.

We modify symbols at the end of the kernel initialization, after the DIKernel switcher extension is loaded. This way, we don't disrupt the base kernel and the DI-switcher internal symbol usage, but we ensure that all extensions that will be loaded in the future will use the modified symbol value. Symbols are redirected to their corresponding wrapper in the DI-switcher code.

Listing 2: Kernel API wrapper example

```
int wrapper__pdr(
        struct platform_driver *pd,
        struct module *mod) {
    int ret;
    entry_gate();
    ret =
    __platform_driver_register(pd,
        mod);
    exit_gate();
    return ret;
}
EXPORT_SYMBOL(wrapper__pdr);
```

Listing 3: Data wrapper structure

```
struct data_wrapper_info {
    char *data_sym_name;
    unsigned long new_value;
    void (*init_wrapper) (void);
    void (*update_wrapper) (void);
    void (*delete_wrapper) (void);
};
```

Similarly, to ensure the kernel will not crash when the system receives an interrupt (including the common system call case), the interrupt handler can be modified so that it first calls a function in the DI-switcher. It would then open the base kernel domain and call the original handler specific to this interrupt. Unlike Intel, ARM involves no register to point to the interrupt table. Hence, DIKernel can ensure an attacker would not be able to abuse the interrupt process.

Symbols concerning data are also modified after the kernel initialization, but are more complicated to wrap since their value needs to be updated. If this data contains a pointer, the pointed data has to be updated as well. The `update_wrapper` function is called when the module starts and whenever a kernel API function is called that could have modified this data symbol.

5. Evaluation

5.1. *Performance*

We evaluated the performance impact of DIKernel by measuring the speed of function call between base kernel and extension. To measure the time precisely, we use the ARM performance counter and get the clock cycles while issuing a function call that involves domain transitions.

Tables 2 and 3 summarize our evaluation on Rasberry-pi ARMv7 board. The evaluation measures the overhead on the three operations DIKernel needs to interpose: (i) the loading of extensions, (ii) the call of extension functions from the base kernel, (iii) the call of kernel APIs from an extension. Each operation is performed more than 1000 times for accurate result.

The overhead on module insertion have low impact on the overall system performance since it usually is only done once at the system boot, we notice it has an overhead of about 20K clock cycles. It is the result of walking the page table in order

Table 2 – Module loading and init function call measured in clock cycles.

Module	Loading time		Init function call time	
	DIKernel turned off	DIKernel turned on	DIKernel turned off	DIKernel turned on
Blank module	304K	318K	13.7K	15.0K
clk_hifiberry_dacpro	380K	407K	17.4K	18.7K
bcm2835_rng	454K	478K	16.6K	18.3K

Table 3 – Kernel APIs call measured in clock cycles. `kmalloc` called to allocate only 4 bytes.

Kernel API	DIKernel turned off	DIKernel turned on
`kmalloc`	239	1586
`kfree`	319	1625
`platform_diver_unregister`	37,094	37,305
`-aeabi_unwaind_cpp_pr0`	18	96

Table 4 – Known rootkits targeting the base kernel.

Rootkit name	Affected base kernel data structure
Adore-NG 0.41	`inode, task_struct, module`
Knark 2.4.3	`proc_dir_entry, task_struct, module`
Kis 0.9	`proc_dir_entry, tcp4_seq_fops, module`
EnyeLKM 1.3	`module`
Adore-NG 0.56	`proc_root_inode_operations, ext3_dir_operations, ext3_file_operations, unix_dgram_ops`
override	`system_call_table`
Sebek 3.2.0	`system_call_table`

to change the extension domain. The scanning of DACR and TTBR modifying instruction is not included in this evaluation.

On the other hand, the overhead to call an extension, that is measured in the case of the `module_init` call can be frequently repeated. For example, in Linux, extensions are often used via the `ioctl` interface which involves a base kernel to extension call; triggered by userspace application. Since there are no publicly available application benchmarks for measuring the performance of kernel to extension function calls, we made a dedicated cycle-level microbenchmark. According to the microbenchmark result, function calls to extension from base kernel required 1.6K additional clock cycles for the domain transition.

Wrappers interpose extension kernel API calls. Generally, wrappers operations are fairly straightforward: (i) they need to open the base kernel domain, (ii) call the base kernel function, and (iii) close back the base kernel and pass the return value to the extension. As can be seen for `-aeabi_unwaind_cpp_pr0` and `platform_diver_unregister` this involves little overhead.

However, wrappers for other kernel APIs as `kmalloc` and `kfree` involve more operations. Once the original function is called, they need to synchronize the copy of the `kmalloc_caches` data structures with their current value. Moreover, `kmalloc` needs to walk the page table to change the domain ID of the newly allocated memory. Nevertheless, wrappers are shown to involve 1.3K clock cycles overhead or less.

Comparing DIKernel to the other most practical solution, placing extensions in a virtual machine (LeVasseur et al., 2004; Nikolaev and Back, 2013), the induced overhead is much more lightweight. A domain transition has been shown to cost at most 1.6K clock cycles for both directions, from base kernel to extensions (calls to extension init function) and from extensions to base kernel (calls to kernel APIs). Moreover, this also includes some synchronization operation in `kmalloc` and `kfree` cases.

On the other hand, process based isolation or virtualization based isolation performance cost depends on the speed of context switches. A process context switch is generally an expensive operation that requires usually around 30K clock cycles and up to millions of cycles[3] (Li et al., 2007).

The strength of DIKernel performance wise, is that once the domains are set up by modifying the page tables, switching from the execution environment from the base kernel to the extensions only requires updating the DACR which is a single register access operation. On the other hand, process based isolation involves switching the page tables and flushing the TLB which is a memory access operation.

5.2. *Security*

The security effectiveness of DIKernel is provided by prohibiting direct memory access from extension to kernel memory. The majority operations of rootkits are blocked under the DIKernel environment. Table 4 is the summary of popular rootkits and their manipulation targets. We can summarize the main targets of such rootkits: (i) system call table, (ii) module metadata, and (iii) file system related data structure such as inode.

5.2.1. *System call table manipulation*

Manipulation of the system call table is a classic rootkit attack example. Recent kernel does not allow direct manipulation of system call table by marking the page read-only. However, fully-privileged rootkit can easily bypass such protection by changing the page permission. DIKernel does not allow page table access to untrusted extensions, therefore direct manipulation attempt against the system call table is blocked.

5.2.2. *Module metadata manipulation*

Rootkits often hide themselves from the extension linked list which is maintained by the kernel. The essence of this manipulation is to remove the extension (rootkit module) metadata

[3] Context switch cost is a high variance operation that depends on the cache architecture and scheduling mechanism.

from the global linked list, yet preserves all the actual contents in memory and page table. In order to perform this manipulation, rootkit should access the kernel memory that saves the linked list head data structure. The design of DIKernel does not allow such access without alarming the base kernel (interrupt handler).

5.2.3. *Inode manipulation*

Inode is also a common target of rootkits. The kernel maintains convoluted data structures regarding the file system such as inode. By changing some member variables inside such data structure, rootkits can hide specific files and various system information (by manipulating the `proc` file system). Due to the presence of DIKernel, stealthy manipulation attempts against critical kernel data become infeasible.

6. Related work

6.1. *Rootkit mitigation*

The first answer to malware exploiting Kernel vulnerabilities is to patch them directly or to mitigate their impact with features such as non-executable pages, supervisor mode access prevention, or supervisor mode execution protection. While these techniques greatly improve the overall kernel security, attackers still find ways to bypass them.

Considering, a monolithic kernel code base is so large that it cannot be patched not to contain any exploitable vulnerability, the ensuing approach is to let attacks happen, but to detect them in hope to be able to reverse the system state to a healthy one. However, since detection tools have to be placed in the kernel address space as well, they can also be targeted and defeated by rootkits. As shown by recent attacks like E-DKOM (Graziano et al., 2016) and "hypervisor for rootkits" (Phrack article), attackers can either outsmart these tools to still hide themselves, or they can target and disable them directly. Rootkits and monitoring tools are engaged in a race of counters where, ultimately, the attacker always has the advantage of the initiative.

Copilot (Petroni et al., 2004) and later KI-Mon (Lee et al., 2017) are hardware-assisted solutions using a PCI device to monitor the RAM by either taking snapshots or spoofing its modification. Information is then sent to another monitor machine over an independent communication link to check the kernel binary text and certain code pointers are not modified to an unknown value. They require heavy architecture installation in order to be used.

Hardware-based protection approaches are monitoring tools that rely on the isolation provided by hardware so that malwares are not able to target them. For instance, TrustZone is used by TZ-RKP (Azab et al., 2014) and SPROBES (Ge et al., 2014) to run in a secure world where only approved applications can be executed. Aside from the incompleteness entailed by any detection approach, the context switch between the secure and the normal world is more expensive than domain transition (evaluated by TZ-RKP (Azab et al., 2014) to be over 2K clock cycles for a round trip). The purpose and design of TrustZone differs from domain isolation which provides multiple memory regions with different access rights. TrustZone uses dedicated global system registers and its page tables with additional operating system running inside the secure world.

Security tools can also benefit from virtualization and the isolation it provides. For practical reasons, they are usually run alongside the hypervisor. Yet, common hypervisors suffer from the same issue than monolithic kernel: since they must provide guest OSes many functionalities (i.e. resource allocation and hardware peripheral virtualization), they have a very large code base, thus both VMware and Xen have a growing number of vulnerabilities (CVEdetails.com, 2017a, 2017b).

Lares (Payne et al., 2008) places hook in the kernel to be able to actively monitor the kernel, analyze its behavior and detect rootkits. Besides being as vulnerable as the platform it runs on, Lares dynamic analysis is incomplete and cannot ensure rootkit detection. HookSafe (Wang et al., 2009) uses a method very similar to Lares, but to prevent kernel control data (i.e. function pointers) from being modified.

So as not to rely on higher privileged system components, Nested Kernel (Dautenhahn et al., 2015) and SKEE (Azab et al., 2016) create a lightweight, isolated execution environment within a monolithic kernel by depriving it from the capability to modify the system's memory layout, respectively on the x86 and the ARM architecture. Their protected address space can then be used to run monitoring tools. Unfortunately, both techniques were found not to be practical: Nested Kernel can only operate on a single core system, and SKEE has high performance overhead.

NICLKE (Riley et al., 2008) and SecVisor (Seshadri et al., 2007) prevent rootkit from loading their own code and executing it. They perform real-time kernel code authentication so that only authenticated kernel code is allowed to be run. Recently developed return oriented rootkits (Hund et al., 2009) can defeat this defense scheme.

6.2. *Decomposition and isolation*

Rather than addressing kernel infection by countering attack strategies, another approach is to decompose the kernel to reduce its attack surface.

Micro-Kernels as L4 (Liedtke, 1995) or Minix 3 (Herder et al., 2006) restructure commodity OSes by only providing essential resource management functions like task scheduling and IPCs[4] in the kernel while moving the rest, kernel extensions included, in the user space. Although this provides high assurance, it requires extensive OS re-design and can cause high performance loss, which is what we want to avoid. Hybrid kernels such as Windows NT have a similar structure, except most of their components are in the kernel address space. Microsoft dropped the project because of its performance issues.

Likewise, Wimpy Kernel (Zhou Z. et al., 2014) tries to provide on-demand isolated I/O to application by taking away some of the driver functionalities from commodity OS and re-implemented them as a user process relying on a microhypervisor. It requires drivers to be re-designed with different OSes primitives, which we are trying to stay away from.

Several approaches (Bershad et al., 1995; Boyd-Wickizer and Zeldovich, 2010; Herder et al., 2009; LeVasseur et al., 2004;

[4] Inter-Process Communication.

Nikolaev and Back, 2013; Swift et al., 2003; Zhou et al., 2006) exist to isolate kernel extension from the base kernel or move them to user space. However, their primary purpose is improving the system reliability and fault isolation. In particular, DD/OS (LeVasseur et al., 2004) and VirtuOS (Nikolaev and Back, 2013) run unmodified extensions in separate virtual machines. SUD (Boyd-Wickizer and Zeldovich, 2010) places drivers in a user space process. While they have achieved both compatibility and strong isolation, their high performance cost makes them unpractical.

Nooks (Swift et al., 2003) is a closely related work to ours that mainly differs in two ways. First, our goal is to securely isolate the base kernel from the kernel extensions, while Nooks aims at enhancing the OS reliability and being able to recover from extension failures. Second, Nooks relies on copying page tables to create its isolated domains, synchronizing them and flushes the TLB when switching domain. As a result, communication between the base kernel and extensions is very expensive.

ARMlock (Zhou Y. et al., 2014) and shreds (Chen et al., 2016) are DACR based isolation solutions targeting user space malwares using in-process memory isolation. ARMlock is a sandboxing like solution to safely use libraries without increasing the potential vulnerabilities of an application. Similarly, shreds are secure execution units by having their code and memory isolated from the rest of the process.

7. Limitations and discussion

DIKernel isolate the extensions from the base kernel. Thus, containing attacks coming from the perspective of the base kernel. However, an attack that achieves to exploit an expansion could spread in the user space. If a user that is limited in its access rights succeeds in exploiting an extension and is able to execute arbitrary code (using in kernel ROP (Hund et al., 2009) for example), it can read security critical files like `/etc/shadow` and gain access to every user. DIKernel achievement is to protect code in the kernel space; the user space is not DIKernel focus.

DIKernel in its current state does not prevent attacks coming from the base kernel. DIKernel is not able to protect monitoring tool from the base kernel as nested kernel (Dautenhahn et al., 2015) and SKEE (Azab et al., 2016) can, or even hardware TEE solutions like TIMA (Azab et al., 2014) do unless it is reworked in a major way. DIKernel goal is to narrow the attack surface.

In the rare case where an extension has to directly access kernel data to modify it (Swift et al., 2003), we have to write a specific wrapper to copy and synchronize data, or register such extension and let them access the base kernel domain. This breaks our R1 isolation requirement since some extensions are able to modify some kernel data. As discussed in Section 3.2, this shows the limit of our model, where containment and isolation conflict with compatibility. Existing code wasn't designed considering our isolation scheme.

Direct Memory Access (DMA) lets an external device read or write in the memory without using virtual memory, thus bypassing our MMU based isolation. Therefore, an extension in control of such a device could defeat DIKernel isolation.

In this case, to the best of the authors' knowledge, no other solution than IOMMU (Input-Output Memory Management Unit) has been found. It creates a third memory space, besides virtual and physical memory, often called bus memory. Hence, it is possible to restrict external devices memory access to specific areas. In particular for DIKernel, it is possible to prevent extensions from accessing the base kernel code or data and restrict them to the extension domain.

Extensions are only one attack vector rootkits can use. Exploitable vulnerabilities can still be found in the base kernel. While this is outside of our scope, DIKernel shows it is possible to securely isolate a part of the kernel from the rest without relying on memory virtualization (LeVasseur et al., 2004; Nikolaev and Back, 2013); being as secure but with a lower performance cost.

A way of expanding DIKernel would be to divide the kernel further and to use more domains. System calls could be a good target since they are directly exposed to users, crafted arguments can be passed to them, hence their vulnerabilities are more likely to be exploited (Jones, 2011). Thus, it would be interesting to remove system calls as an attack surface to the rest of the kernel as we did with extensions. The challenge is to make their code independent enough. This would require careful optimization since system call performances are critical to the overall system.

8. Conclusion

We have presented DIKernel, a system to securely isolate kernel extension execution and limit their memory access permission from the rest of the kernel components while being performant, and compatible with existing kernel extensions. As a result, it prevents malicious extensions from manipulating the entire operating system.

Unlike previous works, DIKernel achieves monolithic kernel division in two address spaces without relying on memory virtualization or on external hardware installation. It leverages hardware component so called *domain access control* introduced in ARM architecture.

Our experiments show that DIKernel involves negligible overhead to the overall system performance yet efficiently prevents known rootkit attacks.

Acknowledgment

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Ministry of Science and ICT (MSIT) (Grant No. B0717-16-0109).

This research was also supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2017R1A2B3006360).

REFERENCES

Adore-NG 0.41. Rootkit; 2004. Available from: https://packetstormsecurity.com/files/32843/adore-ng-0.41.tgz.html. [Accessed 22 January 2018].

Akritidis P. Cling: a memory allocator to mitigate dangling pointers. In: USENIX security symposium. 2010. p. 177–92.
ARM domain access control; 2001. Available from: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0198e/Babegcjf.html. [Accessed 22 January 2018].
Azab AM, Ning P, Shah J, Chen Q, Bhutkar R, Ganesh G. Hypervision across worlds: real-time kernel protection from the ARM TrustZone secure world. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM; 2014. p. 90–102.
Azab AM, Swidowski K, Bhutkar JM, Shen W, Wang R, Ning P. SKEE: a lightweight secure kernel-level execution environment for ARM. In: NDSS. 2016.
Bershad BN, Savage S, Pardyak P, Sirer EG, Fiuczynski ME, Becker D, et al. Extensibility safety and performance in the SPIN operating system. In: ACM SIGOPS Operating Systems Review, vol. 29, No. 5. ACM; 1995. p. 267–83.
Boyd-Wickizer S, Zeldovich N. Tolerating malicious device drivers in Linux. In: USENIX annual technical conference. 2010.
Chen H, Mao Y, Wang X, Zhou D, Zeldovich N, Kaashoek MF. Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In: Proceedings of the second Asia-Pacific workshop on systems. ACM; 2011. p. 5.
Chen Y, Reymondjohnson S, Sun Z, Lu L. Shreds: fine-grained execution units with private memory. In: Security and Privacy (SP), 2016 IEEE symposium on. IEEE; 2016. p. 56–71.
Cortex-A7 MPCore technical reference manual; 2011. Available from: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0464d/DDI0464D_cortex_a7_mpcore_r0p3_trm.pdf. [Accessed 22 January 2018].
CVE-2016-2384. Vulnerability in the usb-midi Linux kernel driver; 2016. Available from: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-238. [Accessed 22 January 2018].
CVE-2017-2636. Race condition in the n_hdlc Linux kernel driver; 2017. Available from: https://cve.mitre.org/cgi-bin/cvename.cgi?name=2017-2636. [Accessed 22 January 2018].
CVEdetails.com. XEN: vulnerability statistics; 2017a. Available from: https://www.cvedetails.com/product/23463/XEN-XEN.html. [Accessed 23 January 2018].
CVEdetails.com. Vmware : vulnerability statistics; 2017b. Available from: https://www.cvedetails.com/vendor/252/Vmware.html. [Accessed 23 January 2018].
Dautenhahn N, Kasampalis T, Dietz W, Criswell J, Adve V. Nested kernel: an operating system architecture for intra-kernel privilege separation. ACM SIGPLAN Notices 2015;50(4):191–206.
Ge X, Vijayakumar H, Jaeger T. SPROBES: enforcing kernel code integrity on the TrustZone architecture. arXiv preprint arXiv:1410.7747. 2014
Graziano M, Flore L, Lanzi A, Balzarotti D. Subverting operating system properties through evolutionary DKOM attacks. In: Detection of intrusions and malware, and vulnerability assessment. Springer International Publishing; 2016. p. 3–24.
Herder JN, Bos H, Gras B, Homburg P, Tanenbaum AS. MINIX 3: a highly reliable, self-repairing operating system. ACM SIGOPS Operat Syst Rev 2006;40(3):80–9.
Herder JN, Bos H, Gras B, Homburg P, Tanenbaum AS. Fault isolation for device drivers. In: Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP international conference on. IEEE; 2009. p. 33–42.
Hund R, Holz T, Freiling FC. Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In: USENIX security symposium. 2009. p. 383–98.
Jang D, Lee H, Kim M, Kim D, Kim D, Kang BB, et al. ATRA: address translation redirection attack against hardware-based external monitors. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications *Security*. ACM; 2014. p. 167–78.
Jiang X, Wang X, Xu D. Stealthy malware detection through VMM-based out-of-the-box semantic view reconstruction. In: Proceedings of the 14th ACM conference on computer and communications security. ACM; 2007. p. 128–38.
Jones D. Trinity: a system call fuzzer. In: Proceedings of the 13th Ottawa Linux symposium. 2011.
Kernel.org. Support for the PXN CPU feature on ARMv7; 2014. Available from: https://patchwork.kernel.org/patch/5151581. [Accessed 23 January 2018].
Knark 2.4.3. Rootkit; 2001. Available from: https://packetstormsecurity.com/files/24853/knark-2.4.3.tgz.html. [Accessed 23 January 2018].
Lee H, Moon H, Heo I, Jang D, Jang J, Kim K, et al. KI-Mon ARM: a hardware-assisted event-triggered monitoring platform for mutable kernel object. IEEE Trans Depend Secure Comput 2017.
LeVasseur J, Uhlig V, Stoess J, Götz S. Unmodified device driver reuse and improved system dependability via virtual machines. In: Proceedings of the 6th conference on symposium on operating systems design & implementation. 2004.
Li C, Ding C, Shen K. Quantifying the cost of context switch. In: Proceedings of the 2007 workshop on experimental computer science. ACM; 2007. p. 2.
Liedtke J. On micro-kernel construction, vol. 29. No. 5. ACM; 1995. p. 237–50.
Nikolaev R, Back G. VirtuOS: an operating system with kernel virtualization. In: Proceedings of the twenty-fourth ACM symposium on operating systems principles. ACM; 2013. p. 116–32.
Payne BD, Carbone M, Sharif M, Lee W. Lares: an architecture for secure active monitoring using virtualization. In: Security and Privacy, 2008. SP 2008. IEEE symposium on. IEEE; 2008. p. 233–47.
Petroni NL Jr, Hicks M. Automated detection of persistent kernel control-flow attacks. In: Proceedings of the 14th ACM conference on computer and communications security. ACM; 2007. p. 103–15.
Petroni NL Jr, Fraser T, Molina J, Arbaugh WA. Copilot – a coprocessor-based kernel runtime integrity monitor. In: USENIX security symposium. 2004. p. 179–94.
Phrack article. How to hide a hook: a hypervisor for rootkits; 2016. Available from: http://www.phrack.org/issues/69/15.html. [Accessed 23 January 2018].
Rhee J, Riley R, Xu D, Jiang X. Defeating dynamic data kernel rootkit attacks via VMM-based guest-transparent monitoring. In: Availability, Reliability and Security, 2009. ARES'09. international conference on. IEEE; 2009. p. 74–81.
Riley R, Jiang X, Xu D. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In: International workshop on recent advances in intrusion detection. Berlin Heidelberg: Springer; 2008. p. 1–20.
Seltzer MI, Endo Y, Small C, Smith KA. Dealing with disaster: surviving misbehaved kernel extensions. In: OSDI, vol. 96, No. 56. 1996. p. 213–27.
Seshadri A, Luk M, Qu N, Perrig A. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In: ACM SIGOPS Operating Systems review, vol. 41, No. 6. ACM; 2007. p. 335–50.
Swift MM, Bershad BN, Levy HM. Improving the reliability of commodity operating systems. In: ACM SIGOPS operating systems review, vol. 37, No. 5. ACM; 2003. p. 207–22.
Unified EFI. Unified extensible firmware interface specification: Version 2.2d, November 2010.

Wang Z, Jiang X, Cui W, Ning P. Countering kernel rootkits with lightweight hook protection. In: Proceedings of the 16th ACM conference on computer and communications security. ACM; 2009. p. 545–54.
Xu H, Du W, Chapin SJ. Detecting exploit code execution in loadable kernel modules. In: Computer security applications conference, 2004. 20th annual. IEEE; 2004. p. 101–10.
Zaddach J, Kurmus A, Balzarotti D, Blass EO, Francillon A, Koltsidas I. Implementation and implications of a stealth hard-drive backdoor. In: Proceedings of the 29th annual computer security applications conference. ACM; 2013. p. 279–88.
Zhou F, Condit J, Anderson Z, Bagrak I, Ennals R, Harren M, et al. SafeDrive: safe and recoverable extensions using language-based techniques. In: Proceedings of the 7th symposium on operating systems design and implementation. USENIX Association; 2006. p. 45–60.
Zhou Y, Wang X, Chen Y, Wang Z. ARMlock: hardware-based fault isolation for arm. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM; 2014. p. 558–69.
Zhou Z, Yu M, Gligor VD. Dancing with giants: wimpy kernels for on-demand isolated I/O. In: Security and Privacy (SP), 2014 IEEE symposium on. IEEE; 2014. p. 308–23.

Valentin J.M. Manès is currently a researcher at the Cyber Security Research Center (CSRC). He received his Master's degree from Télécom ParisTech with a specialization in network security and information systems security. During a one year exchange at KAIST, he developed an interest for HW-based trusted execution environment.

Daehee Jang received the B.S. degree in Computer Engineering from Hanyang University, South Korea, in 2012. He also received the M.S. degree in Information Security from Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2014. He is currently working toward the Ph.D. degree at the Division of Computer Science, Korea Advanced Institute of Science and Technology (KAIST). His research interests include software vulnerability, operating system, and Intel SGX.

Chanho Ryu is currently a senior researcher and research manager at the Cyber Security Research Center (CSRC). He has also worked in the president's office and has been a senior researcher at the Korea Atomic Energy Research Institute and at the Korea Computerization Agency. Dr. Ryu received his Ph.D. in Computer Science from the Chungnam National University.

Brent Byunghoon Kang is currently an associate professor at the Graduate School of Information Security at KAIST. He has also been with George Mason University as an associate professor in the Volgenau School of Engineering. Dr. Kang received his Ph.D. in Computer Science from the University of California at Berkeley, and M.S. from the University of Maryland at College Park, and B.S. from Seoul National University. Dr. Kang has been working on systems security including OS kernel integrity monitors, HW-based trusted execution environment, VM Introspection, Memory address translation integrity, Code-Reuse Attack defenses, invisible server, anti-spam, and botnet malware analysis.