BarriCCAde: Isolating Closed-Source Drivers with ARM CCA

Matti Schulze
FAU Erlangen-Nürnberg

Christian Lindenmeier
FAU Erlangen-Nürnberg

Jonas Röckl
FAU Erlangen-Nürnberg

***Abstract*—Due to the nature of monolithic kernels, a driver's vulnerability always results in a vulnerable kernel, as disastrous examples show. Some research aims to solve this issue by isolating drivers from the main kernel. The most promising approaches rely on virtualization techniques. However, as they usually require driver and kernel modification, they can only be applied to open-source drivers. As closed-source drivers are also standard these days, e.g., for GPUs or anti-cheat tools, we propose BarriCCAde, a new design for isolating even closed-source drivers from the kernel. Most notably, BarriCCAde allows driver isolation without adding a hypervisor to the TCB. Relying on upcoming confidential computing techniques, we only add small-scale memory protection components to our TCB. Contemporary approaches focus on deciding which of the kernel's resources the driver may access and how they can be synchronized between the kernel and the driver. However, one aspect mainly ignored is cases in which the driver misuses kernel resources it can access to attack the system, e.g., by crafting malicious inputs for kernel functions. To cover such cases, we integrate an eBPF-based filter into our system architecture, which allows a fine-granular specification of which kernel-level resource can be accessed in which way. We believe that BarriCCAde is an important step towards isolating future closed-source drivers and, thus, strengthening the confidentiality, integrity, and availability of future kernels.**

1. Introduction

With Linux, MacOS, and Windows, most modern devices run a monolithic kernel, usually written in unsafe languages like C. This risk, combined with the enormous codebases of modern operating systems (OS), results in a de facto guarantee of OS vulnerabilities. While modern OSs deploy many security mechanisms to mitigate this risk, new vulnerabilities are found frequently, with, for example, 532 CVEs being reported for Linux alone in 2023 [22]. Crucially, many of these vulnerabilities do not stem from the kernel itself but from drivers, which make up approximately 70% of the Linux source code [11]. As these OSs are monolithic by design, a vulnerable driver means that exploiting a vulnerable driver can lead to a full system compromise, which implies that all data managed by the OS is at risk. Many researchers have already observed this, leading to a long history of techniques that aim to isolate a driver from the rest of the kernel.

1.1. Related Work

While some of the earliest approaches discussed the usage of micro- and nanokernels to isolate drivers [8], [15], they never really caught on in commodity OSs.

Another string of research focusses on isolating drivers from the kernel in monolithic OSs by using virtualization techniques [14], [17], [24]. Typically, such systems employ a hypervisor on which two VMs are executed, one for the driver and the kernel each. From this base design, we observe two different categories of approaches.

The first hosts a fully-featured second kernel in the "driver-VM" which has access to hardware needed by the driver and shares the driver's data with a second VM hosting the main system [7], [24]. Here, the driver has direct access to all the kernel resources needed for its execution (e.g., kernel functions and data structures) but has the downside of requiring modifications to the driver and kernel source-code to enable the synchronization of the computed data. Moreover, having a higher resource consumption, especially in regards to memory.

The second category tries to run the driver as "bare metal" as possible in the second VM to reduce its memory footprint [2], [9], [16], [21]. However, this leads to numerous challenges, especially the synchronization of needed kernel resources between the kernel and the now "kernel-less" VM. Contemporary research, primarily the string of research around Narayanan et al., has suggested some solutions to this problem. Starting in 2019, the authors introduced so-called Lightweight Execution Domains (LXDs) [2], a concept to enable kernel subsystems to execute in a, from the kernel isolated, environment. A microkernel added to the commodity OS manages IPC calls between the LXDs and the isolated kernel, which can be used to request capabilities (e.g., access to memory regions or function calls). Which capabilities can be requested and how the resources of capabilities are synced between the two VMs is manually specified via an interface definition language (IDL), which compiles to the glue-code needed to actually synchronize the resources. The glue-code is then added to the driver's source-code and compiled into it. The concept was expanded multiple times. First, they propose replacing the microkernel with a full hypervisor and to add new lightweight isolation mechanisms [21]. Subsequently, to use the driver modules LLVM intermediate representation to generate automatic IDL descriptions and lastly in a work-in-progress paper to also implement additional security features like heap isolation or single ownership of the shared resources [9]. While only briefly mentioned by the authors, proxy interfaces generated from the IDL could be used to prevent some accesses to shared resources which would result in cyclic dependencies. The technique could also be used to only allow accesses of kernel functions with certain parameters. However, the authors do not go into how proxies can be generated for this use case, leaving the implementation of the actual parameter-based filtering vague.

1.2. Contributions

While the research on LXDs offers the most promising techniques for isolating malicious drivers from the main kernel, we identify three main issues. First, all tools assume access to the source code of the drivers, which prevents a holistic solution **(I1)**. The authors require this to implement the context switches between the VMs and resource sharing. However, we believe this can also be achieved in a partial black-box scenario where the driver is closed-source. To do this, hardware faults can be used to initiate the context switches and the kernel's side of the driver-kernel interface can be used to generate the rules for resource synchronization. While we believe that this is a step in the right direction, it still leaves the door open for attacks against the hypervisor used for the isolation. As hypervisors also tend to have vulnerabilities due to the size and complexity of their source code, adding this component also adds another risk to the system **(I2)**. Depending on the size and complexity of the driver, this risk may outweigh the risk reduction by isolating the driver. While this could not be avoided for a long time, novel TEE architectures like ARM's Confidential Compute Architecture (CCA) allow for the isolation of a system's OS from an untrusted hypervisor. Therefore, applying ARM CCA to the ideas of LXDs can reduce one of their significant drawbacks. Lastly, while the recent work introduces mature concepts for synchronizing resources between the kernel and isolated domains, no design offers solutions for enabling the filtering of malicious usage of validly shared resources **(I3)**.

To that end, we present BarriCCAde which, in summary, offers the following solutions for the issues listed above.

- **(I1)** BarriCCAde is, to the best of our knowledge, the first tool, which will be able to run *any* kernel module, including closed-source drivers, completely bare-metal in virtual machines, without any modification to the driver. For this, we present a novel technique for deciding on the driver's resource synchronization requirements relying on the kernel's debug information.
- **(I2)** BarriCCAde leverages ARM CCA and its concept of realms to minimize the increase of the TCB.
- **(I3)** BarriCCAde presents a solution for preventing the malicious usage of shared resources. We achieve this by leveraging eBPF to filter illegal resource access.

2. Background

2.1. ARM CCA

For a long time, research on trusted execution environments focused on protecting information on a running system from a compromised OS (e.g., Intel SGX, ARM TrustZone). With the rise of cloud computing, novel technologies focus on preventing an untrusted hypervisor from accessing or modifying information inside a VM. Solutions for this have been presented in recent years by most CPU manufacturers, with AMD SEV [23], Intel TDX [10], and ARM's CCA [18]. While ARM CCA contains an architecture for the entire chain of trust of cloud systems, its backbone is the extension of its CPU architecture with the so-called *Realm Extension*. The extension expands on the long-standing separation of ARM TrustZone systems in the secure and normal world by introducing two new system states, the *realm world* and *root world*, as shown in Figure 1. On most ARM systems, EL3 is the highest privilege level and reserved for the *Monitor* (often implemented by the "Trusted Firmware(TF)-A" [20]). While EL3 used to be assigned to the secure world, with ARM CCA it is now moved to its own system state, the so-called root state. Additionally, the realm world contains with the *Realm Management Monitor* (RMM) another firmware component running at R-EL2,

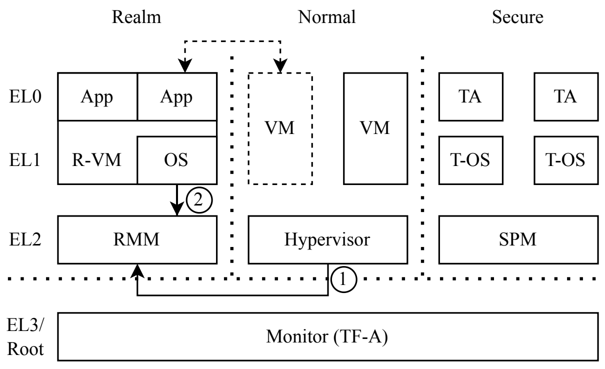

Figure 1. Overview of the system architecture of ARM CCA and the different system states, also called *worlds*. A hypervisor can issue RMIs (1) to request actions from the RMM, while a R-VM can issues RSIs (2) to call the RMM.

ARM CCA employs the concept of separating the management of software components from their protection. This is implemented by having the normal world hypervisor create, schedule, and destroy VMs but forwarding all requests through the RMM, which handles the protection of the VMs' resources. This protection is primarily achieved with a new hardware component, the Granule Protection Table (GPT). The GPT, only writeable from the root world, holds an entry for every physical memory page of the system, where each entry is associated with the physical address space (PAS) it belongs to (i.e., normal, secure, realm or root world). On every memory access, the system inputs the physical memory address calculated by the MMU in this GPT and checks which PAS this page belongs to. This information is checked against the current system state to decide whether access will be allowed. Table 1 shows which PAS can be accessed in which system state. A "Y" implies that a page assigned to a certain security state (x-axis) can be accessed when the system is in the security state defined on the y-axis. In general, pages belonging to the realm world can only be accessed if the system itself is in the realm world or the root world.

When a hypervisor creates a new realm VM, it allocates all the needed resources (e.g., memory) and management structures like it would with any "normal" VM and then requests the protection of the resources from the RMM. This is achieved with the so-called Realm Management Interface (RMI), a collection of Secure Monitor Calls (SMC) routed to the RMM by the firmware. The RMM then creates a new *realm* for this VM and, with

TABLE 1. OVERVIEW OF ARM CCA PAS.

Security State	Normal	Secure	Realm	Root
Normal	Y	N	N	N
Secure	Y	Y	N	N
Realm	Y	N	Y	N
Root	Y	Y	Y	Y

the help of the Monitor, assigns all of the VM's memory pages to the realm world via the GPT. With this, any attempt from the hypervisor to access the memory from a realm is blocked by hardware. Furthermore, a realm can request services from the RMM or the hypervisor by issuing so-called Realm Service Interface (RSI) calls, also implemented via SMCs (shown at ② in Figure 1).

ARM provides formal verification for the correctness of the reference implementations of the RMM and the aspects of TF-A concerning realms, meaning that even though they have full access to all of a realm's information, they are highly unlikely to contain vulnerabilities [13], [18]. Therefore, when considering the topic of driver isolation via virtualization, ARM CCA could be used to assign the VMs to realms so that a hypervisor can not break the confidentiality and integrity of the system. Considering **I2**, this means that in such a case the hypervisor does not effectively increase the systems TCB (in regards to confidentiality and integrity), which makes CCA a promising approach.

2.2. Extended Berkeley Packet Filter

The extended Berkeley Packet Filter (eBPF) [12] is the successor of the Berkeley Packet Filter (BPF), designed initially to implement user-defined filtering rules for network traffic. eBPF expands on this to implement a full virtual machine with a RISC-like architecture. It hooks into various kernel subsystems and allows the execution of user-defined code in the privileged context of the kernel at runtime. To achieve this securely, eBPF sandboxes the code and uses a verifier that analyses it before approving it for just-in-time compilation. This verifier uses elaborate checks to make sure that the program does terminate, has a finite complexity and does not break out of the sandbox. To achieve this, eBPF disallows certain features that prevent the verifier from executing its checks, like unbound loops, unsafe memory accesses via uninitialized variables. To improve the usability of eBPF, the LLVM frontend can generate eBPF bytecode as a target, meaning that any LLVM-based programming language can be used to create eBPF programs. While initially designed for filtering network traffic, it can be theoretically used to filter *any* stream of information/requests, like in the case of driver isolation, requests from the driver-VM to kernel resources.

3. Threat Model

We assume two attack scenarios. Firstly, we assume an attacker who can execute arbitrary code in a driver's closed-source kernel module. We identify three attack vectors which can be used for this. Firstly, it is possible that the user unknowingly loads a driver which is in itself malicious, e.g., because it hides as anti-cheat software. In the other two scenarios a vulnerable driver is exploited either by userspace malware to escalate its privileges or by malicious hardware (e.g., specifically designed DMA devices [6]). We assume the driver is the kernel level's only source of malicious behavior, meaning the other kernel components are considered trusted.

To align with ARM CCA's design, we also include the scenario, where an attacker has achieved full control the hypervisor used to isolate the driver. Furthermore, we assume that the firmware components of the ARM CCA software stack (TF-A and RMM) can be trusted. Lastly, we consider side-channel attacks and attacks requiring physical access out of scope (e.g., fault injection attacks).

4. Design and Implementation

The goal of BarriCCAde is to offer an isolated environment for closed-source drivers, preventing any malicious behavior of these from affecting the main kernel and userspace. Inspired by Narayanan et al. [21], we aim to enforce five aspects of security:

- **(R1)** *Data Structure Confidentiality*, meaning that the driver can not read confidential data structures from the kernel that are not required for the driver to run
- **(R2)** *Data Structure Integrity*, meaning that the driver cannot modify confidential data of the kernel
- **(R3)** *Function Call Integrity*, meaning that the driver can only execute a subset of the kernel functions required to function correctly
- **(R4)** *Synchronization Safety*, meaning that the function calls and data structures synchronized between the driver and the kernel can be prevented from being used with values and parameters that may lead to malicious behavior. However, it must be noted that we only refer to securing inputs used for synchronized resources and do not include securing the interface from vulnerabilities based on race conditions.
- **(R5)** *Minimal TCB increase*, meaning that an implementation for isolating drivers does not significantly increase the system's TCB.

While **R1**-**R3** are defined in a similar way by Narayanan et al., **R4** expands on the requirements by not only defining *which* resources may be accessed by a driver, but also *how* the driver may access the resources. This is an important aspect of security since the misuse of resources actually needed by the driver can still be used to exploit the kernel, e.g., by triggering a kernel panic by intentionally calling a function with specific parameters, resulting in a crash. While this is also briefly mentioned in the most recent work-in-progress successor of KSplit [9], we showcase a concrete implementation. Furthermore, we expand this concept applying it to the global data structures of the kernel. **R5** focusses on scenarios, where the driver isolation is implemented by adding other higher privileged software components (e.g., a full hypervisor) with potentially higher complexity than the driver. While isolating drivers aims to reduce the kernel's TCB effectively, the addition such software may increase the TCB to an extent even higher than in a scenario where the system does not utilize such isolation techniques.

4.1. BarriCCAde's Approach

Figure 2 presents an overview of the architecture of BarriCCAde, designed for modern ARM platforms that implement the realm extension. In detail, we use the extension's hardware primitives while also implementing the entire runtime logic of BarriCCAde and its modules in the new firmware component, the RMM. The main runtime logic is implemented in the BarriCCAde-Modules in the RMM, and more minor additions for the setup phases are done to the system's kernel and the hypervisor (in our case, KVM) used. However, the components in the hypervisor are not assumed to be be trusted, aligning with the threat model. The setup is used to run the main kernel in a virtualized environment (K-VM), which the RMM then protects. Any potential driver is also loaded in a new VM created by the hypervisor and protected by the RMM, thus guaranteeing integrity and confidentiality even in the face of an untrusted hypervisor. To achieve this, the Linux procedure to load kernel modules is modified; Figure 3 presents an overview of the new process implemented. The following three subsections describe the steps of the process.

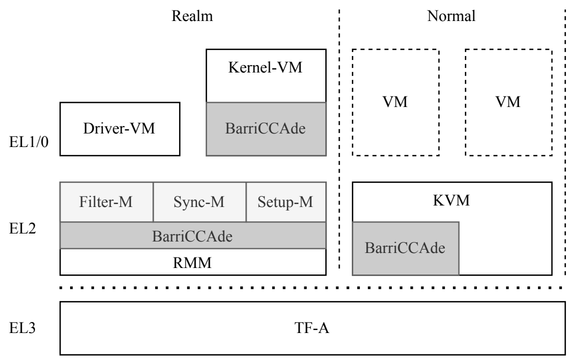

Figure 2. Overview of how BarriCCAde is embedded into the ARM CCA components. Additions are made to the RMM, the Kernel and KVM. It must be noted that the additions from KVM are not considered trusted.

4.2. Resource Synchronization

When a driver is loaded, BarriCCAde generates a so-called *sync config* for this driver to decide how to synchronize the resources (i.e., the parameters and return values used for kernel functions and data structures) needed by the driver. To decide which resources need to be synchronized in which way, we design the workflow shown in step (1) of Figure 3.

In general, most approaches for resource synchronization require the source code of both the kernel and the driver to be isolated, as they assume that both aspects are needed to specify rules for the synchronization. The sum of resources accessible can be interpreted as an interface that the kernel presents to potential modules, which are also linked against this interface after compilation. Even if a driver offers resources *to* the kernel, it has to adhere to the way the kernel accesses such resources, which also can be interpreted as an interface provided by the kernel.

For effective synchronization of resources, this interface is usually analyzed to generate the ruleset. However,

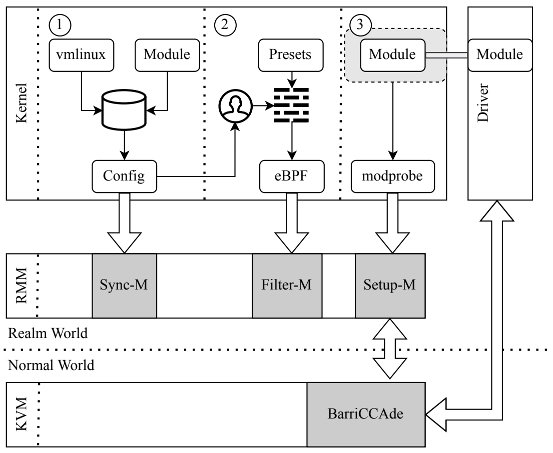

Figure 3. Overview of the three steps of BarriCCAde's kernel-module loading mechanism.

this interface is solely defined by the kernel, including information on how resources can be accessed and their assumed properties (e.g., data structure layouts, size, constness, lifetimes). We believe that instead of having to rely on the user of the interface (i.e., the driver) and its provider (i.e., the kernel), the interface definition specified by the provider is enough to derive rules for how resources must be synchronized between the driver and the kernel. Furthermore, we believe that even without having access to the driver's source code, the information provided in its `ELF` is sufficient to achieve this in an efficient way.

Therefore, to analyze the kernel-driver interface, we only assume the kernel to be open source. While analyzing the entire source code may be beneficial for future work, we only extract the kernel's interface from it. For this, we use the `DWARF` debug information from the kernel, which can be extracted from the Linux kernel if it is compiled accordingly. While we can derive all rules for synchronization from the kernel's interface, doing this for every resource is unnecessary, as generating rules for those required by the driver is sufficient. For this, the driver's interface usage must be analyzed, i.e., which resources are required.

One way to achieve this is by extracting the ELF information from the driver, including the list of its symbols. This list is then searched for undefined symbols, which means that the module calls a function or accesses a data structure that is provided by the kernel. With this, we generate the list of kernel dependencies of the kernel module. For each dependency, we check the `DWARF` entries and generate a *sync_config* by checking the return values and parameters. Suppose a parameter is of a primitive data type. In that case, we can synchronize it by copying the kernel's register to the register in the driver-VM's saved context before returning to it. If a parameter is a pointer, we must decide on its semantics (i.e., "normal" pointer, a pointer to an array, ...). While the problem is not decidable in general, with more profound analysis, most of these semantics can be decided. If a parameter is a pointer, it can be synced by coping bytes equal to the size of its targeted data type to a memory region accessible by the driver. We call this the *sync_region*, which will be further discussed in Section 4.4. In this case, we also have to modify the register storing the parameter to point to the location of the synchronized data in the sync_region.

Suppose a pointer points to a composite data type (i.e., a *struct*). In that case, we can unroll the `DWARF` information of the type (recursively) to get the exact layout of it and copy the needed information accordingly. For example, if a parameter points to a structure that stores two pointers, we can follow the pointers, copy the needed information, and adjust the struct's pointers accordingly before copying the whole structure to the sync_space and adjusting the parameter to point to this copy of the structure. After a function call, resources may also be synchronized back to the source if it does not contain constant pointers. The creation of such rules, where possible, is handled automatically by a simple user-space script. The same approach is also used for global variables, with the only difference being that the adjusted pointer is not stored in a function parameter but instead at the memory location or register storing the original pointer.

If the script is unable to generate a configuration for a function call (e.g., a parameter is a `void *`), it lists these functions in the output. While, for now, the operator may manually create a sync_config for such calls, we believe that by analyzing the kernel's source code in a way comparable to KSplit's approach [16], even such ambiguous cases can be resolved in most cases, leaving only a small amount to be specified manually. However, requiring manual configuration is in line with the latest research for approaches focussing on open-source drivers, as these also require manual intervention in some cases [16].

Finally, the sync_configs generated from this approach are then uploaded to BarriCCAde in the RMM as raw C-structures via shared memory. BarriCCAde then uses these configs to synchronize calls to functions and kernel data structures on demand.

4.3. Function Call Filtering

While our synchronization mechanism already prevents the driver-VM from directly accessing or modifying resources that it must not access, **R4** also requires considering the situation where a resource that is synchronized with the driver is miss-used maliciously, e.g., by executing a valid function with specific parameters leading to a kernel panic or information leakage.

To tackle this issue, we integrate an eBPF interpreter into BarriCCAde in the RMM (see Section 4.5) and implement the workflow shown in step ② of Figure 3. The eBPF interpreter executes a user-defined program before every synchronization step. For the input of this program, BarriCCAde uses the address of the resource and, in the case of a function, its parameters, and in the case of a data structure, the value it shall be set to. Furthermore, the program is always given the synchronization rules for this address. To support the creation of the filter rules, the output of the sync_config generator may be used, as this output already lists all the functions and data objects used by the driver and which may be considered for filtering.

As a modern driver accesses a huge number of kernel resources, we believe that there are multiple approaches to aid in specifying the rules, especially in regards to *how* the rules can be specified by *whom*.

1) **Presets**: There are cases in which drivers most likely use certain events in the event space of a function for malicious purposes. For example, while a driver may require functions to map certain physical memory regions for MMIO and DMA, it is highly unlikely that a driver has benign reasons to use such functions to map the physical memory of the kernel's text segment. Therefore, general presets can be generated to protect the kernel from common attack patterns.
2) **Automatic Derivition**: To protect the kernel from malicious drivers that may try to crash the system, resulting in a DoS, we believe that source-code analysis of the kernel can be used in some cases to generate rules preventing this. For example, if the analysis shows that a function contains a call to the kernel's panic function, we can backtrack the control flow to potentially decide on a set of parameters leading to this panic. By generating rules that filter out function calls with these arguments, we can automatically protect the kernel from some DoS attacks.
3) **Vendor-Defined Rules**: While most vendors publishing closed-source tools aim to protect their intellectual property by not publicly sharing the source code, the driver's usage of the kernel's interface can always be analyzed by, e.g., tracing components of the kernel. Furthermore, publishing the interface's usage generally maintains the confidentiality of the driver's inner workings. Consequentially, it could be considered for driver vendors to specify the rules themselves, meaning that even if their driver contains a vulnerability, the running system will not be compromised. In cases of a potentially malicious vendor, such rules should, be reviewed by a third party to prevent rules that allow malicious behavior.

4.4. Loading the Driver

After loading all the configurations to the RMM, the kernel can then load the driver as a BarriCCAde-driver, as shown in step ③ of Figure 3, by placing the driver's relocated elf in a shared memory region of the hypervisor and the kernel (marked light gray). Next, the kernel issues an RSI to the RMM with the location of the driver in the shared memory and a hash of it, which forwards the RSI as a `HOSTCALL` to the hypervisor, which then creates a new VM, placing the driver's elf in it. The driver-VM is then also protected by the RMM, which also uses the hash provided by the kernel to prove its integrity in the context of a potentially malicious hypervisor. Furthermore, the hypervisor allocates additional memory regions for the driver-VM, containing its stack and regions for synchronizing resources (i.e., the sync_region). While a copy of the driver is loaded in its realm, the original driver is still loaded normally in the Linux kernel. However, its physical pages are marked as inaccessible in the GPT (`GPT_GPI_NO_ACCESS`), meaning that any call to the driver from the kernel (starting with its `init`) call is trapped to BarriCCAde in the RMM. Furthermore, if the driver is executing and a kernel function is called or the address of a kernel data structure is accessed, these addresses are not mapped into the VM. Therefore, this results in a page fault, trapping it to BarriCCAde in the RMM.

4.5. Runtime

Figure 4 shows the flow of its execution upon loading a driver. First, BarriCCAde synchronizes the the parameter for the *init* function of the driver and then returns to the driver-VM at the according address. The driver executes until a kernel resource is accessed, e.g., a function or a global variable. As the function's address is not mapped into the driver-VM, this results in a trap to BarriCCAde in the RMM. Here, the function's address, parameters, and sync_config are fed into the *filter module* of BarriCCAde, which contains the eBPF mentioned above. The filter then executes the kernel-provided program and checks the function call for malicious parameters. Similarly, if the requested resource is a kernel object, the address of the object and the value it shall be set to is used as input for the filter. If the resource request passes the filter, it is forwarded to the *sync module*. Here, the sync_configs provided to the RMM are checked for the address of the requested resource. The request is rejected if no configuration is provided, and execution continues at the driver's next instruction. Otherwise, the resource is synchronized according to the provided configuration, and in case of a function call, execution is resumed at the kernel's desired function. Suppose the kernel returns from the function call itself or calls driver functions at some point. In that case, the execution is again trapped to the RMM (as this results in accessing memory pages marked as inaccessible in the GPT), where the filtering and synchronization are executed again, this time in reversed order. The combination of these two techniques results in the driver-VM being unable to read or write data structures of the kernel, addressing **R1** and **R2**, and to execute kernel functions it does not require, addressing **R3**. Furthermore, as the filter also allows the detection of malicious usage of valid resources via runtime-provided eBPF programs, we also adhere to **R4**. Using eBPF programs for filtering also means that the same program, or parts of it, can be reused for different drivers, increasing this design's portability.

Finally, we use a version of the eBPF filter that is formally verified for correctness [25], which means that, other than the synchronization module and some RSI/RMIs forwarding, we add no considerable amount of unverified code to the TCB, adhering to **R5**.

5. Status

The tool presented is a work in progress. It will be further developed to offer a mature and complete solution comparable to the designs around LXDs. Currently, BarriCCAde can synchronize simple function calls between a simple dummy-driver and host and the filtering functionality via eBPF is already in place and working. We have achieved this by adding 2195 single lines of code (SLOC) to the TCB, out of which 1722 contain the formally verified eBPF interpreter (calculated with `cloc 2.0` [5]). Compared to the 228120 SLOC of the RMM (out of which 209473 SLOC stem from external dependencies of the RMM like `mbedtls`), we only add 0.96% to the TCB.

The current focus of the project is to improve the synchronization capabilities of BarriCCAde regarding complex data structures and opaque parameters. For this we also consider the approaches of LXDs [2] and LVDs [21].

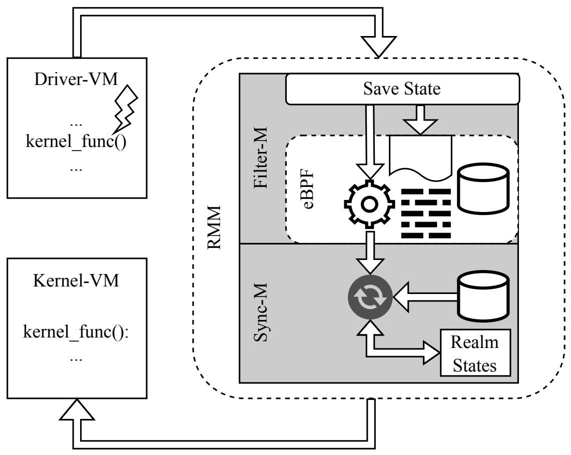

Figure 4. BarriCCAde's process for responsing to a driver's resource request. The cylindric symbols denote the sync_configs and the dotted lines describe the filter rules, which are used by the eBPF interpreter in combination with the driver-VM's state to compute its verdict.

As there is no publicly available hardware implementing the ARMs Realm extension, we implemented the prototype on QEMU with custom ports for the ARM Trusted Firmware and RMM. While this allows for fast prototyping, QEMU cannot be used for performance measurements. Another option is to use ARM's Fixed Virtual Platforms (FVPs) [19], which are custom emulators for ARM's novel architectures. Sadly, these perform very poorly regarding their emulation speed, making them an unfit tool for prototyping BarriCCAde. Additionally, while cycle-accurate, timing measurements on the FVPs are inaccurate, reducing their usability for a performance evaluation [1], [3], [4].

Therefore, to evaluate performance aspects of BarriCCAde, we plan to rely instead on microbenchmarks running on already available ARMv8 hardware without the realm extension. We expect BarriCCAde to introduce some performance overhead introduced by effectively adding two context switches to each function call that is synchronized between the kernel and the driver. The impact of this highly depends on the exact driver being isolated and the amount of kernel function it calls and how performance critical its operation is. This is also described for the comparable designs of LXDs and LVDs [2], [21]. Here, with the performance improvements of LVDs the authors achieve 65% of native speeds to nearly native speeds (99%), depending on the kind of driver and its exact configuration (e.g., multithreading). Moreover, we additionally introduce overhead by adding the eBPF-based filtering mechanism. As these will most likely only consist of simple range-checks, we expect the overhead introduced to be negligable.

Furthermore, we will also evaluate the LoC of eBPF code needed to effectively protect the kernel from different attack scenarios against a real-world driver, for example, using valid functions with malicious parameters which would allow an attacker to comprimise kernel code.

6. Conclusion

This work explores a possible design and implementation for isolating closed-source drivers. To the best of our knowledge, it is the first design that isolates closed-source drivers without relying on multiple kernels. To achieve this, we suggest a novel resource synchronization technique relying on kernel debug information. Furthermore, we also improve on current approach by describing a concrete solution for filtering out potentially malicious accesses of the isolated module on synchronized resources. We achieve this by integrating a formally-verified eBPF interpreter into our system. Furthermore, by aligning BarriCCAde to the upcoming ARM CCA architecture, we improve on other virtualization-based isolation methods by keeping the additions to the TCB small in regards to its confidentiality and integrity due to adding a hypervisor. In conclusion, our concept, when matured, will offer a solution for securely isolating drivers without additional memory overhead or increasing the system's TCB. Thus, we believe that our design is an important step forward for driver isolation in future confidential computing architectures.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback. Additionally, we thank Rüdiger Kapitza for reviewing a previous version of this paper. This research was partly supported by the German Federal Ministry of Education and Research (BMBF) as part of the CELTIC-NEXT project AI-NET-ANTILLAS ("Automated Network Telecom Infrastructure with inteLLigent Autonomous Systems", Förderkennzeichen "16KIS1314").

References

[1] ACAI: Protecting Accelerator Execution with Arm Confidential Computing Architecture.

[2] LXDs: Towards isolation of kernel subsystems.

[3] PAC it up: Towards pointer integrity using ARM pointer authentication.

[4] SHELTER: Extending Arm CCA with Isolation in User Space.

[5] AlDanial. cloc. https://github.com/AlDanial/cloc/tree/v2.00, 2024. Accessed: 2024-03-14.

[6] Ramtin Amin and Ulf Frisk. Pcileech. https://github.com/ufrisk/pcileech-fpga/tree/master/pciescreamer, 2019. Accessed: 2024-03-14.

[7] Sebastian Angel, Riad S. Wahby, Max Howald, Joshua B. Leners, Michael Spilo, Zhen Sun, Andrew J. Blumberg, and Michael Walfish. Defending against malicious peripherals with cinch. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, 2016.

[8] Allen C. Bomberger, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. The keykos nanokernel architecture. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures, Seattle, WA, USA, 27-28 April 1992s*, 1992.

[9] Anton Burtsev, Vikram Narayanan, Yongzhe Huang, Kaiming Huang, Gang Tan, and Trent Jaeger. Evolving Operating System Kernels Towards Secure Kernel-Driver Interfaces. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, 2023.

[10] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel tdx demystified: A top-down approach. *arXiv preprint arXiv:2303.15540*, 2023.

[11] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001.

[12] eBPF.io authors. An introduction and deep dive into the ebpf technology. https://ebpf.io/what-is-ebpf/., 2023. Accessed 2024-03-05.

[13] Anthony CJ Fox, Gareth Stockwell, Shale Xiong, Hanno Becker, Dominic P Mulligan, Gustavo Petri, and Nathan Chong. A Verification Methodology for the Arm® Confidential Computing Architecture: From a Secure Specification to Safe Implementations. *Proceedings of the ACM on Programming Languages*, 2023.

[14] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.

[15] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S Shapiro. Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, 2004.

[16] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. KSplit: Automating Device Driver Isolation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

[17] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, 2004.

[18] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and verification of the arm confidential compute architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, 2022.

[19] Arm Limited. Fixed Virtual Platforms. https://developer.arm.com/Tools%20and%20Software/Fixed%20Virtual%20Platforms, 2024. Accessed: 2024-03-14.

[20] Linaro Limited. TrustedFirmware-A (TF-A). https://www.trustedfirmware.org/projects/tf-a, 2024. Accessed: 2024-03-14.

[21] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight kernel isolation with virtualization and VM functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS international conference on virtual execution environments*, 2020.

[22] National Institute of Standards and Technology. National Vulnerability Database. https://nvd.nist.gov/vuln/search/statistics?form_type=Basic&results_type=statistics&query=Linux&search_type=all&isCpeNameSearch=false, 2024. Accessed: 2024-03-14.

[23] AMD Sev-Snp. Strengthening VM isolation with integrity protection and more. *White Paper, January*, 2020.

[24] Lin Tan, Ellick Chan, Reza Farivar, Nevedita Mallick, Jeffrey C. Carlyle, Francis M. David, and Roy H. Campbell. ikernel: Isolating buggy and malicious device drivers using hardware virtualization support. In *Third IEEE International Symposium on Dependable, Autonomic and Secure Computing, DASC 2007, Columbia, MD, USA, September 25-26, 2007*, 2007.

[25] Shenghao Yuan, Frédéric Besson, Jean-Pierre Talpin, Samuel Hym, Koen Zandberg, and Emmanuel Baccelli. End-to-end Mechanized Proof of an eBPF Virtual Machine for Micro-controllers. In *International Conference on Computer Aided Verification*, 2022.