Low Overhead Security Isolation using Lightweight Kernels and TEEs

John R. Lange
University of Pittsburgh
jacklange@cs.pitt.edu

Nicholas Gordon
University of Pittsburgh
nick.gordon@cs.pitt.edu

Brian Gaines
Sandia National Labs
bgaines@sandia.gov

***Abstract*—The next generation of supercomputing resources is expected to greatly expand the scope of HPC environments, both in terms of more diverse workloads and user bases, as well as the integration of edge computing infrastructures. This will likely require new mechanisms and approaches at the Operating System level to support these broader classes of workloads along with their different security requirements. We claim that a key mechanism needed for these workloads is the ability to securely compartmentalize the system software executing on a given node. In this paper, we present initial efforts in exploring the integration of secure and trusted computing capabilities into an HPC system software stack. As part of this work we have ported the Kitten Lightweight Kernel (LWK) to the ARM64 architecture and integrated it with the Hafnium hypervisor, a reference implementation of a secure partition manager (SPM) that provides security isolation for virtual machines. By integrating Kitten with Hafnium, we are able to replace the commodity oriented Linux based resource management infrastructure and reduce the overheads introduced by using a full weight kernel (FWK) as the node-level resource scheduler. While our results are very preliminary, we are able to demonstrate measurable performance improvements on small scale ARM based SOC platforms.**

***Index Terms*—Operating Systems, Virtualization, Security**

I. Introduction

Trusted execution and secure resource isolation have become increasingly common in modern hardware architectures [1]–[3], however the use cases for these feature sets has focused on end user devices and commodity systems platforms. While there is significant interest in adapting these technologies to server class systems, so far the focus has primarily been on commodity cloud based environments, and has not directly addressed the needs of HPC class systems. At the same time, these features present a unique opportunity to more readily integrate HPC oriented system software solutions, due to the fact that security isolation inherently requires system level partitioning leading to compartmentalized system software environments. While previous generations of HPC system software required the development of mechanisms to allow HPC OS/Rs to co-exist with monolithic OS/Rs [4]–[6], the next generation will likely provide a much more distributed OS/R model making it relatively easy to swap in HPC oriented sub-components as needed.

In this paper we introduce preliminary work to explore this problem space through the use of the Kitten LWK in combination with secure virtualization capabilities available on modern ARM based systems through the Hafnium hypervisor [7]. Hafnium is a reference implementation of a Secure Partition Manager developed as part of the Trusted Firmware [8] effort led by Linaro. Hafnium provides isolated resource partitions through the use of secure virtualization mechanisms that provides full memory isolation between each VM in the system, including the primary host OS. This capability allows the execution of private workloads that cannot be tampered with or spied on by any other locally executing software, and thus enables the deployment of sensitive data and applications to otherwise untrustworthy system platforms. However, while Hafnium currently provides these capabilities on ARM based systems, it is implemented around a commodity use case that requires a full weight Linux kernel instance to manage VM scheduling on each CPU core. This results in workload interference and other overheads due to the complexity of Linux and its primary focus on commodity environments. To address these issues we have adapted Kitten to allow us to replace the Linux instance as the node-level VM scheduler.

A high level overview of our approach is shown in Figure 1. In our system model, a lightweight kernel acts as a scheduler for each VM hosted on the system. Using Kitten in this role enables low noise, easily configurable, and more deterministic scheduling behaviors to an HPC compute node, without having to rely on the default Linux kernel scheduler. In previous work we have already demonstrated the efficacy of deploying Kitten as a scheduling layer for virtual machines in HPC contexts [4], however in this work we have augmented the performance benefits with added security isolation provided by Hafnium. In this model, neither Kitten nor any other OS instance can access the memory contents of another OS/R environment executing on the system. This is a marked contrast to our previous work with Palacios [9] that allowed (and in some cases required) full access to internal VM memory contents by the host OS. While Palacios took a similar approach to the separation of VM state management from resource management and scheduling, the separation was entirely software based and relied on respecting API boundaries. In contrast, with Hafnium VM state management is separated from resource management

This material is based upon work supported by the National Science Foundation under Grant No. 1704139. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

2021 SC Workshops Supplementary Proceedings (SCWS) | 978-1-6654-8388-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/SCWS55283.2021.00016

978-1-6654-8388-9/21/$31.00 ©2021 IEEE
DOI 10.1109/SCWS55283.2021.00016

and scheduling by hardware enforced protection boundaries implemented by the ARMv8 exception levels. All VM state management functions execute at the higher privilege level of Exception Level 2 (EL2), while scheduling and VM execution all occur at Exception Level 1 (EL1). This allows the system software to separate scheduling policies into a lower privilege level from the dispatch mechanisms that handle VM context switching.

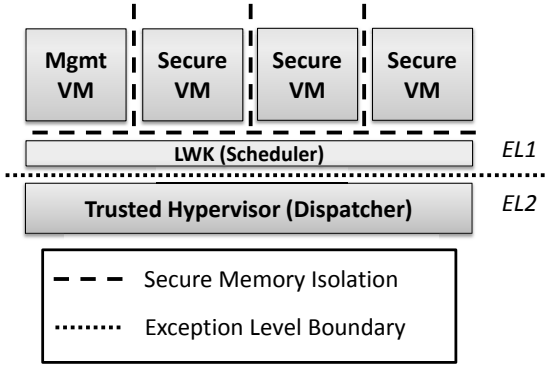

Fig. 1. Using LWKs to manage isolated VMs

In the rest of this paper we will describe our work in porting the Kitten LWK to ARM as well as integrating it with Hafnium. In addition we will present a set of preliminary evaluations on a small ARM SOC platform to demonstrate the implementation as a proof of concept. We will also provide background information on the ARMv8 security mechanisms, as well as discuss future directions in adapting these mechanisms for HPC environments.

II. Background

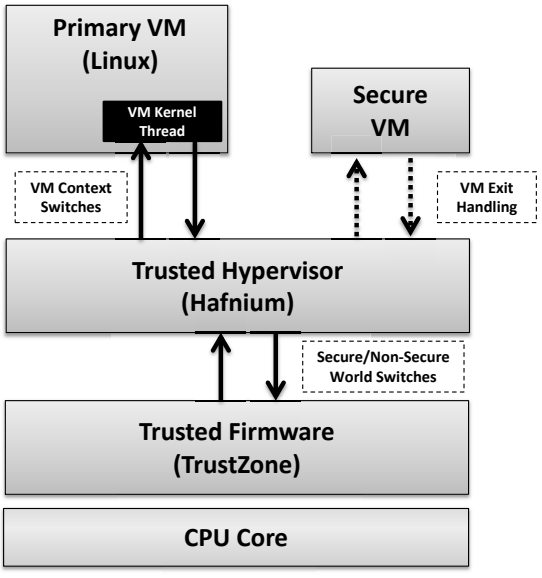

Fig. 2. Default Hafnium VM configuration

a) Hafnium: Hafnium is a member project in the Trusted Firmware Initiative, an open governance community project hosted by Linaro, that aims to provide reference implementations of secure system software for ARM based platforms. Hafnium builds on other TrustedFirmware projects to provide a security isolation layer for virtual machines. By focusing on virtualization, Hafnium provides general purpose execution environments that can be used to deploy full OS/R stacks that run in secure virtual system partitions. This is in contrast to other trusted execution approaches on ARM such as OP-TEE [10] and Trusty [11] that use higher level RPC based services to provide specific security services such as secure secret storage and authentication. With Hafnium, an entire VM context can be isolated from other software components on the system, thus allowing deployment of full HPC applications into secure computing environments.

The design of Hafnium is influenced heavily by the architecture of the ARMv8 virtualization extensions, which are based on a more purely hierarchical approach than the SVM/VMX extensions on x86 platforms. In ARMv8, a hypervisor is directly invoked as part of the boot sequence and is thus able to virtualize the platform before an OS instance is ever run. This allows, the deployment of trusted hypervisor implementations in a straightforward way, as it is simply a link in the chain of the trusted boot sequence. While this allows Hafnium to provide resource protection outside the control of the OS itself, it complicates the design of type-2 hypervisors (such as KVM and Palacios) due to the need to initialize the hypervisor before the host OS is loaded. This poses a challenge to Hafnium because, in order to keep the implementation small to minimize its attack surface, it eschews many of the features typically seen in a full type-1 hypervisor implementation (such as a CPU scheduler and I/O virtualization) and instead adopts a type-2 architecture. With this approach, VM management operations are pushed up to a host OS instance executing at EL1 (kernel level), while the hypervisor executing at EL2 handles only the context switches between VM instances. It should be noted that KVM takes the same general approach by loading a thin hypervisor layer into EL2 as part of the Linux boot process, but currently lacks the inter-VM security isolation of Hafnium. One notable limitation of this approach, is that dynamic resource partitioning becomes difficult due to the need to create secure partitions as part of the early boot process before resource management services have been initialized.

One of the key design decisions of Hafnium was to focus primarily on memory isolation between VM instances. As a result Hafnium provides no guarantees about a VM's availability or performance. This decision is a side effect of the type-2 hypervisor architecture that relies on a host OS instance (executing inside a VM context), to provide scheduling capabilities for the system. The system architecture resulting from this can be seen in Figure 2. As the diagram shows, Hafnium relies on a *primary* VM to make scheduling decisions and explicitly invoke context switches to *secondary* VMs through a privileged hypercall interface. The Hafnium

reference implementation provides a Linux device driver that provides VM lifecycle management and a small set of management operations. The Linux device driver provides scheduling by creating a Linux kernel thread for each VCPU belonging to a particular VM. Each kernel thread holds a handle to a single VCPU context managed by Hafnium's hypervisor, and so can direct Hafnium to context switch to that VCPU instance via a dedicated hypercall. Once a VM has been scheduled in, Hafnium performs a context switch into the given VCPU and resumes execution inside the VM's context. As the VM executes, VM exits are taken to the Hafnium hypervisor with the majority being handled internally by the hypervisor, and only a subset of exits (e.g. IRQs from timers/devices or VM aborts) resulting in the invocation of the Primary VM's host OS.

An important aspect to this is that Hafnium's hypercall interface is core local, meaning that Hafnium itself does not support inter-core communication and instead relies on the primary VM/host OS to handle all inter-core communication and coordination. As a result, it is not possible for Linux to invoke a VM context switch on another core than the one it is executing the hypercall from. This in turn means that in order for Hafnium to execute VMs on every core in the system, the primary VM's scheduler must be actively running on every core as well. Therefore, under the default configuration with a Linux primary VM, Linux must be running on every core in the system (along with its associated kernel threads and background tasks). This is especially egregious since few if any of Linux's features are actually necessary because each VM is operating in an isolated and self contained partition.

b) ARM TrustZone: Memory security and isolation enabled by Hafnium is capable of being supported using two different mechanisms. In the simplest case, Hafnium is included as part of the trusted boot sequence and instantiates nested page tables over all of memory before any OS is initialized. This allows Hafnium to have direct control over all memory mapping operations, and so is able to enforce memory isolation via hardware virtual memory mechanisms. In this case the security guarantees provided by Hafnium are dependent on the attested boot chain as well as the correctness of Hafnium itself. As a result, the implementation of Hafnium is kept as small as possible to minimize the bug surface and reduce its TCB. For added memory protection Hafnium also supports TrustZone [1] enabled memory protection. TrustZone is ARM's implementation of a Trusted Execution Environment (TEE), and provides hardware enforced memory isolation at the system firmware level (EL3). With TrustZone an ARM system is divided into secure and non-secure worlds and memory is configured at boot time into either the secure or non-secure partitions. With TrustZone enabled, the ARM boot sequence forks at EL3 and executes parallel boot sequences in both secure and non-secure instances of EL2, EL1, and EL0. This allows a verifiable boot chain to load trusted OS/R environments into isolated hardware partitions enforced entirely at the firmware layer. Non-secure partitions are prevented from accessing secure memory contents, while secure partitions are allowed access to both secure and non-secure memory regions.

Hafnium is designed to support both secure and non-secure VM instances, and provide memory isolation for each. This is complicated by the fact that TrustZone requires the existence of both secure and non-secure hypervisor instances to handle VMs of each type. Nevertheless, the complexity is entirely contained in the state handling and context switching code and does not impact the scheduling approach taken by Hafnium (that is, a single primary VM can provide scheduling for both secure and non-secure VM instances). The one limitation of note in current TrustZone architectures is the requirement that the secure and non-secure memory partitions must be statically sized and configured during the early boot process.

III. Design

In this paper we look to explore how to leverage the security isolation capabilities of Hafnium (and the ARM architecture in general) for the purpose of securing sensitive HPC workloads and data. While a fair amount of work has gone into developing secure and trusted computation environments for ARM based systems, very little if any has focused on HPC environments. As past experience has shown, it is unlikely that commodity oriented approaches will be able to fully meet the scalability and performance requirements of HPC systems.

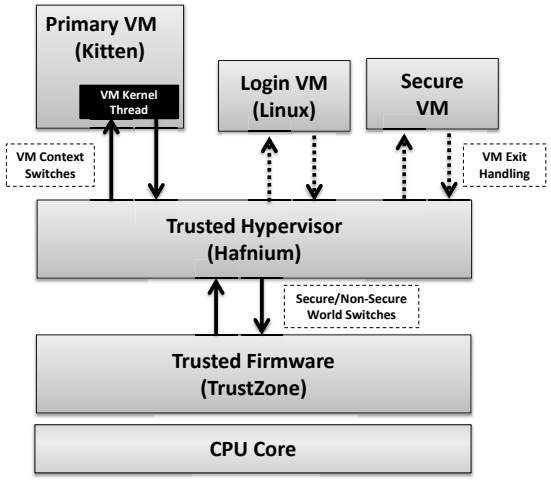

Fig. 3. Proposed architecture to use the Kitten LWK as the primary Hafnium VM

The primary design goal of this work is to replace Linux with Kitten as the VM scheduler for Hafnium protected environments, with a secondary goal of keeping a Linux OS/R instance available to provide system management operations required by modern HPC environments. In essence, we wish to take the exascale co-kernel approach (isolating a Linux instance to a subset of local resources) and adapt it to the security isolation environments likely to emerge in the next generation of systems. A high level overview of our proposed approach is shown in Figure 3.

a) Kitten as the Primary VM: The core component of our approach is the deployment of the Kitten LWK as the primary scheduling VM in a Hafnium based secure virtualization environment. Our goal is to eliminate, or at least greatly reduce, the OS overheads imposed by the primary scheduling VM on HPC workloads executing in secure VM instances. Kitten is well suited to this task, and has already been shown to provide superior performance in this role on x86 based systems [4], [12]. While there are a number of reasons for Kitten's superior performance, at a high level it comes down to the fact that the Linux scheduler is optimized around a time-shared process based model where applications' behaviors are able to directly influence scheduling decisions. However, when virtual machines are present on a system the behaviors the Linux scheduler have been optimized around are masked by the additional software layers implementing the VM abstraction. As a result Linux often lacks the necessary information needed to optimize VM scheduling decisions, especially in the HPC use case. Kitten (and other LWKs) on the other hand take a very different approach to scheduling based on the their general philosophy of exposing hardware capabilities directly to the higher software layers. This maps very well to virtualized environments where each VM OS/R assumes it is running on physical hardware. In other words, with Kitten, the assumptions that a VM's kernel scheduler makes are much closer to reality than the assumptions made when running on top of Linux, which in turn translates to better performance for the VM's applications. This benefit is particular acute when the Linux scheduler is placed under load with multiple competing workloads.

In addition to reducing the semantic gap between a VM's OS/R and the underlying hardware, Kitten is also able to provide a number of other benefits that are applicable to our use case. By default Kitten is designed for non-interactive jobs, allowing significantly larger time slices for the scheduler quantum and thus lower timer tick rates. This reduces the overheads imposed by scheduler policy evaluation, interrupt handling, and switching between the host and VM contexts. In addition Kitten has little to no background tasks that need to periodically run, nor does it have deferred work that is randomly assigned to a CPU core. Finally, Kitten provides a number of indirect benefits as well, primarily due to its much simpler design that allows easier customization and modification to directly optimize a much narrower set of target workloads.

Deploying Kitten as a primary VM with Hafnium does however introduce a number of challenges, in particular regarding I/O and device drivers. Due to the fact that Hafnium assumes that it will have a full featured OS kernel running as the primary VM, it is designed to allow the primary VM to have full control over I/O devices. This is achieved by providing the primary VM with direct access to the underlying MMIO ranges (unless specifically partitioned to a dedicated secondary VM) and delivers all hardware interrupts to the primary VM instance. Device driver support has been a consistent problem for LWKs, and was a key consideration in the adoption of co-kernel based approaches that offloaded I/O driver responsibilities to a concurrent full featured OS instance. This will likely become an even bigger problem in the future given the trends towards more heterogeneous accelerator devices with correspondingly complex device driver dependencies.

b) Linux as an Service VM: As was recognized by multiple projects during the exascale OS/R research effort, supporting Linux environments on each compute node is a requirement in order to make the system usable and accessible. This is likely to hold true on future systems, and so our approach includes the ability to host a Linux based environment as a management VM instance. In our model a VM running Linux would be the login environment that would be responsible for running the node's "native" user space environment. With this approach, job control operations such as launching and stopping VM instances would be invoked from the Linux environment. Supporting such a configuration is not possible with Hafnium as currently designed, so we have extended the Hafnium architecture with the concept of a *super-secondary* VM instance that exists in a semi-privileged state between the primary VM and other secondary VMs. As originally designed, the Hafnium architecture only supports a single primary VM that is allowed to run with greater privileges than the secure secondary VMs. These privileges include access to the local I/O devices as well as privileged architectural features that are masked from secondary VMs due to security concerns. In our model the super-secondary instance gains the ability to directly interact with I/O devices but does not have full access to the hypercall API or the ability to assume control over CPU cores. In Figure 3 the super-secondary is denoted as the "Login VM".

Providing a super-secondary instance with direct access to the I/O devices allows the use of the Linux device driver ecosystem and removes that responsibility from the Kitten primary VM. Delegating I/O responsibilities to the super-secondary requires splitting the role of the primary VM inside Hafnium itself. For the most part this is fairly straightforward once Hafnium is extended with the concept of a super-secondary. During system initialization Hafnium already maps all the MMIO regions to the primary VM, so this simply needs to be changed to map those regions into the super-secondary instead with appropriate updates made to the device tree configuration to reflect which I/O devices are actually available in the super-secondary partition. The more challenging issue is IRQ routing between the primary and super-secondary. Because Hafnium assumes it will only have a single primary VM, by default it routes all interrupts to the Primary VM instance. While it would be easy to simply redirect all hardware interrupts to the super-secondary this is not possible due to the fact that the Kitten Primary VM requires that all hardware timer interrupts be routed directly to it. As a result, it is necessary to provide some form of selective IRQ routing where timer interrupts are delivered to the primary VM, while device IRQs are instead routed to the super-secondary. This is an area of future work for us, and our current approach is to continue to route all interrupts to the primary VM which

is then responsible for forwarding any device IRQ on to the super-secondary.

Finally, because the super-secondary includes a Linux user space environment, system management frameworks can be easily deployed and accessed. With this approach, job control responsibilities would live inside the super-secondary VM, and include the ability to configure system resources as well as manage the lifecycles of the secondary VMs. VM management is handled by a secure communication channel between the super-secondary and primary VMs allowing the super-secondary to issue commands to a control task executing in the Kitten VM instance. The control task would then be responsible for translating these job control commands into the appropriate scheduling policy changes and hypercall invocations necessary to execute them.

IV. Implementation

To demonstrate the feasibility of our system architecture, we have developed a prototype implementation as a proof of concept. Our implementation includes a Kitten LWK primary VM that is able to host multiple instances of Kitten based secondary VMs. In addition we have very early work to support super-secondary VM instances. As part of our development effort we undertook the porting of the Kitten LWK from x86 to ARM64. This work was initially conducted at Sandia National Labs and completed at the University of Pittsburgh. All of this work is currently available in the Kitten Github repository [1]. Currently Kitten supports a limited number of ARM SOC platforms based on either the GIC2, GIC3, or Broadcom 2836 IRQ controllers. Verified hardware platforms include the Pine A64 [13] SBC, Raspberry Pi [14], and Qemu ARM64 VM profile. While we have demonstrated Kitten running on actual ARM hardware platforms, we have not yet gotten access to HPC oriented platforms such as ThunderX2 based systems. We intend to add support for these systems in the near future as access becomes available.

a) Kitten as the Primary VM: Extending Kitten to run as the Primary VM on Hafnium required relatively little modification from the native version. The implementation effort primarily required porting the hypercall interface from the Linux driver implementation, and exporting VM management operations via a device file to user space. When running as the primary VM, Kitten executes a control task in user space that is responsible for handling VM management operations (such as starting and stopping VMs). By default when the Kitten primary boots up, it queries the Hafnium hypervisor for information regarding the resource partitions and available VM images. It then immediately launches the super-secondary VM instance in order to provide an accessible user space environment for the system and initialize the local I/O devices. The control process is then responsible for launching and terminating secondary VMs on demand.

When launching a VM hafnium uses the same approach as the Linux implementation and creates a dedicated kernel thread for each of the VM's VCPUs. By default these VCPUs are spread across available CPU cores incrementally, but CPU assignments can be configured and even modified during the secondary VM's execution. These kernel threads are then placed into the Kitten scheduler run-queues and upon execution immediately invoke Hafnium to switch to the associated VCPU context.

b) Kitten as a secondary VM: In contrast to the primary VM environment, porting Kitten to execute as a secondary VM under Hafnium required a greater deal of effort. In order to fully isolate each secondary VM instance, Hafnium disallows access to a significant number of hardware features. As such, it is not straightforward to directly execute a native capable kernel inside a secondary VM context. In order to port Kitten to this secondary environment required disabling a number of low level architectural features and providing work-arounds where appropriate. These included features such as the performance counter and debug registers and various low level architectural instructions such as the *dcisw* cache flushing operations. In addition, the secondary VMs must use a para-virtual interrupt controller interface provided by Hafnium as well as the dedicated virtual architectural timer channel. However, with these modifications in place Kitten is able to execute normally.

c) Super-Secondary VM: Implementing support for a Linux based super-secondary requires modifying Hafnium to support the concept of a super-secondary VM as well as modifying Linux to run in a semi-privileged VM context. As stated earlier, the modifications to Hafnium were relatively straightforward, since privilege checks are done by comparing the internal VM identifier against known constants. The necessary modifications were simply adding an additional hardcoded VM ID for the super-secondary and changing various conditionals to match against the super-secondary's ID instead of the primary's. While this was a minor set of modifications, Linux poses a more significant challenge and is currently a work in progress. The immediate requirements are the addition of the same para-virtual interrupt controller interface as is required in secondary VMs as well as the virtual timer. However, Linux also requires a more extensive set of architectural features than Kitten and a significant number of those are blocked by Hafnium. Given the semi-privileged nature of the super-secondary, we expect that most of these features can simply be enabled (as they are enabled by default for the primary), but each one nevertheless requires verification that it does not negatively impact the security guarantees provided by Hafnium. This work is still ongoing.

V. Evaluation

In order to evaluate our approach we have performed a number of performance benchmarks using a Pine A64-LTS Single Board Computer (SBC) platform. While the Pine A64 is a performance and power constrained embedded platform, it is nonetheless useful to demonstrate the feasibility, if not the full efficacy, of our approach. The Pine A64 system we used in our evaluation is based on a 4 core ARM Cortex A53 running

[1] https://github.com/HobbesOSR/kitten

at 1.1Ghz with 2GB of RAM. All of our experiments are single node and focus entirely on CPU and memory performance, as we do not yet have the ability to support virtual I/O interfaces. Each benchmark was executed on a Kitten OS/R instance running inside a Hafnium secondary VM, with the exception of the baseline performance measurements which were collected using a native Kitten environment. The benchmarks were the only workloads executing on the system, so there should have been little to no resource contention from competing workloads. The benchmarks we used in our evaluation were the HPCG mini-app, stream and random access memory micro benchmarks, selfish-detour, and a subset of the NAS parallel benchmark suite.

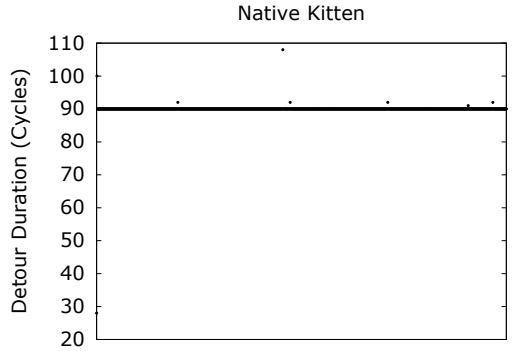

Fig. 4. Selfish Detour Benchmark running on native Kitten.

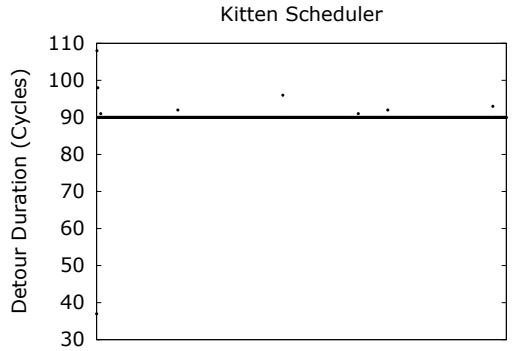

Fig. 5. Selfish Detour Benchmark running on secondary Kitten VM with a Kitten scheduler VM.

a) Selfish: We first evaluated the noise profile of each of the execution configurations using the selfish-detour benchmark. Due to the fact that the only load in the system was from the benchmark itself, the noise profiles should reflect a close to best case scenario with no competing workloads. As can be seen in Figure 4, the native Kitten configuration presents a constrained noise profile with only a small number of pauses due to timer ticks. This behavior is consistent with previous evaluations of LWK performance, and serves as a baseline for the other two configurations. Figure 5 shows the results from

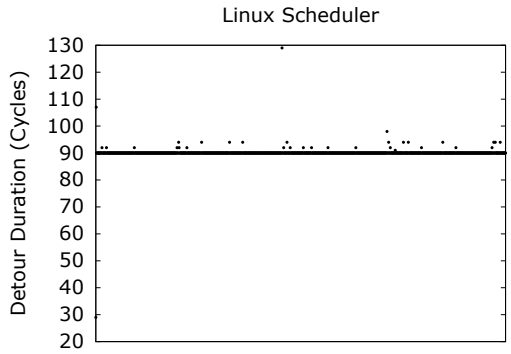

Fig. 6. Selfish Detour Benchmark running on secondary Kitten VM with a Linux scheduler VM.

running selfish inside a Kitten secondary VM instance with Kitten acting as the primary VM scheduler on the system. As the figure shows adding a virtualization layer causes little to no change to noise profile of the environment. The only difference is a slight increase in detour latencies when they do occur. Figure 6 shows the results of using Linux as the scheduling VM with a Kitten secondary VM. In this case, the noise events are more frequent and more randomly distributed due to a combination of timer tick latencies and competing threads in the Linux environment. Based on this evaluation, we can show that replacing Linux with Kitten for VM scheduling can have a measurable effect on reducing the noise of the secondary VM environments.

b) Stream and RandomAccess: We next performed a series of memory tests using the Stream and RandomAccess micro-benchmarks. Because Hafnium relies on hardware virtualization of the memory map, memory operations from a secure VM will be required to traverse two sets of page tables as part of the translation. This additional overhead will be particularly noticeable in the RandomAccess benchmark due to its low TLB hit rates. The results of these experiments are shown in Figure 7 as normalized values, with the raw measurements included in Figure 8. As expected, RandomAccess is the most impacted by the presence of Hafnium, and experiences performance degradation for both of the virtualized cases. However, the Linux scheduler performs the worst, most likely due to the increased number of timer interrupts resulting in increased TLB pressure from the more frequent VM context switches. Oddly, the Stream results show that performance increases with the presence of Hafnium, however it should be noted that the mean performance of each configuration falls within the standard deviation, so the performance differences are not statistically significant.

c) HPCG: Beyond micro-benchmarks, we also evaluated system performance using the HPCG mini-app, whose results are included in Figure 7 and Figure 8. These results also show very similar performance across each of the three configurations, however on average the Hafnium with Kitten scheduler configuration does show slightly better overall performance.

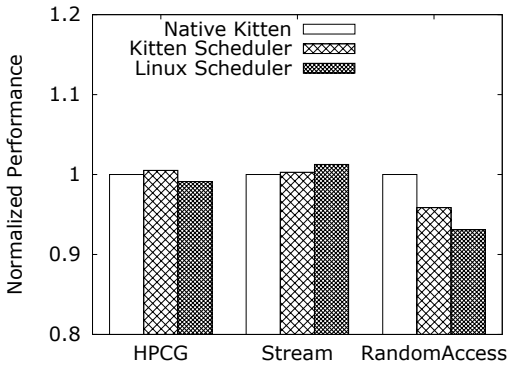

Fig. 7. Normalized Performance of the HPCG, Stream, and RandomAccess Benchmarks

	HPCG		Stream		RandomAccess	
	Mean	*Stdev*	*Mean*	*Stdev*	*Mean*	*Stdev*
Native	0.0018	$3e^{-5}$	59.6	0.14	$6.5e^{-5}$	$5.7e^{-10}$
Kitten	0.0019	$3e^{-5}$	59.8	0.14	$6.2e^{-5}$	$3.4e^{-8}$
Linux	0.0018	$3e^{-5}$	60.2	0.42	$6.04e^{-5}$	3.6^{-9}

Fig. 8. HPCG, Stream, and RandomAccess Benchmark performance (GFlops, MB/s, GUP/s)

The main take away from this experiment is that our proposed approach is capable of executing a full mini-app benchmark with minimal to know overheads inside a securely virtualized partition.

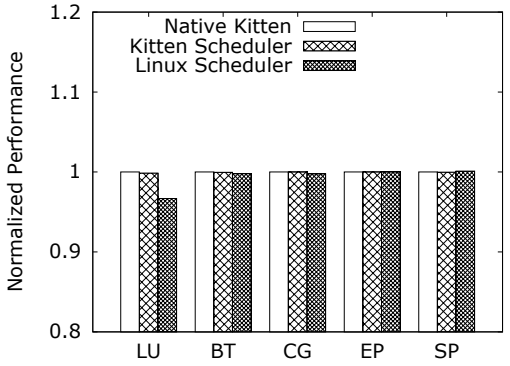

Fig. 9. Normalized Performance of the LU, BT, CG, EP, and SP NAS Parallel Benchmarks

	LU	BT	CG	EP	SP
Native	33.16	34.214	4.38	0.77	15.084
Kitten	33.116	34.2	4.38	0.77	15.08
Linux	32.06	34.142	4.37	0.77	15.1

Fig. 10. NAS Parallel Benchmark performance (Mop/s))

d) NAS Parallel Benchmarks: Finally we evaluated the performance of a subset of the NAS Parallel Benchmark suite (specifically the LU, BT, CG, EP, and SP benchmarks). As with the earlier evaluation, normalized performance is shown in Figure 9 and the raw measurements in Figure 10. In these experiments application performance showed little to no degradation across each of the three configurations. The one exception was a very slight performance drop with the Linux based scheduler running the LU benchmark.

e) Discussion: While these benchmarks are hardly definitive in regards to the effectiveness of our approach, they are consistent with previous studies of LWK performance. Therefore it is reasonable to expect that we will see similar trends documented in previous studies comparing LWK and FWK overheads. All told the benchmarks do not present anything that is new in regards to LWK performance, which is the point to take away for our evaluation. The impact of virtualization and secure isolation appear to be minimal in our benchmark results, indicating the security based approaches do not intrinsically impose significant performance overheads. We believe these results demonstrate that further exploration of using LWKs to manage securely partitioned HPC systems is warranted. Overall, our evaluation demonstrates that we can deploy security isolation mechanisms with minimal overheads for HPC applications and workloads.

VI. Related Work

To our knowledge there is relatively little work investigating the integration of secure and trusted computation environments into HPC OS/R stacks. Design papers and whitepapers exist outlining what an ARM-capable system designed for HPC would need [15], [16], but at the time of publishing we know of no work that integrates ARM TrustZone features into HPC OS/Rs.

VII. Future Directions

While we have presented preliminary results of our approach and an initial proof of concept implementation, there are a number of open questions and directions to be explored in the space of secure HPC OS/Rs. In the case of our initial system architecture a number of immediate potential research avenues are evident. Most obvious of these is the evaluation of our approach on more realistic systems and workloads. We need to perform more experiments to measure the impact of our approach on more realistic workloads at scale. In the near term we are working on deploying our approach on larger scale and higher performance systems, such as the Astra ThunderX2 based system at Sandia. As part of this effort we intend to not only study the scalability but also the performance isolation capabilities of our approach when multiple workloads are hosted on the same compute node. To achieve this there are additional extensions that need to be made to both Hafnium and Kitten in order to more fully support large scale applications and systems. The most significant of these is the design I/O mechanisms that are able to maintain secure system isolation without imposing significant performance overheads. This alone will be a major challenge moving forward, and will likely require some level of device support to achieve native levels of performance.

In addition to support for I/O access from VM instances, we also need to investigate dynamic partitioning approaches

that allow the creation and reconfiguration of secure partitions at run-time. Currently, Hafnium requires that secure partitions and VM images be defined at boot time. This allows hafnium to create secure memory partitions before any OS is initialized, and removes the complexity of having to reclaim memory in order to launch a new VM. To make our approach suitable for a more dynamic set of workloads, we need to design appropriate management interfaces to allow dynamic memory allocation and reclaiming as part of normal operation. In addition, we will likely need to develop support for launching VM images supplied after the system has booted. This is a complex issue, due to the fact that ARM does not currently have the necessary attestation support that ensures each VM instance is tamper proof. Without hardware support, hafnium will require some mechanisms of verifying VM signatures to ensure their authenticity and provenance. One potential solution would be to leverage certificate verification, where Hafnium is able to verify VM signatures using a known public key that is included as part of the trusted boot sequence.

We are also still unsure whether the Hafnium hypervisor is a suitable long term solution for secure virtualization in HPC. We have already had to modify it in order to support basic HPC workloads, and it is not entirely clear how many further modifications will be necessary to make it a truly viable architecture. We intend to continue to evaluate Hafnium, with the consideration as to whether a wholly new hypervisor architecture needs to be developed that has HPC environments as a foundational design goal. It is also important to note that the next versions of the ARM platform (ARMv9) are introducing significant security, isolation, and trusted computing features that will change both the hardware capabilities as well as the system software approaches needed to leverage them. We expect that these and other hardware security features will play a significant role in the HPC space for the next supercomputing generation.

VIII. Conclusion

In this paper we have introduced a new use case for Lightweight Kernels as resource management services in securely isolated HPC systems. As part of this work we have ported the Kitten LWK to the ARM64 architecture and integrated it with the Hafnium hypervisor to support secure virtual machine instances on a compute node. We have provided an initial proof of concept implementation and provided a preliminary evaluation that shows our approach does not impose significant performance overheads to a range of HPC benchmarks. We have also identified a number of future challenges, and made the case that security isolation and trusted computing will be important features of next generation HPC platforms. Fully supporting security isolation in a scalable and performant way will likely be a challenge for HPC OS/R architectures, and one that merits future research moving forward.

References

[1] "AMD TrustZone," https://developer.arm.com/technologies/trustzone.

[2] "Secure Encrypted Virtualization Key Management," http://support.amd.com/TechDocs/55766_SEV-KMAPI_Spec.pdf.

[3] Intel, "Get Started with the SDK," https://software.intel.com/en-us/sgx/sdk, 08 2019.

[4] J. Ouyang, B. Kocoloski, J. Lange, and K. Pedretti, "Achieving Performance Isolation with Lightweight Co-Kernels," in *Proc. 24th International ACM Symposium on High Performance Distributed Computing (HPDC)*, 2015, To Appear.

[5] B. Gerofi, M. Takagi, and Y. Ishikawa, "IHK/McKernel," in *Operating Systems for Supercomputers and High Performance Computing*, 2019.

[6] R. W. Wisniewski, T. Inglett, P. Keppel, R. Murty, and R. Riesen, "mOS: An Architecture for Extreme-scale Operating Systems," in *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers*, 2014.

[7] "Hafnium Hypervisor," https://www.trustedfirmware.org/projects/hafnium/.

[8] "Trusted Firmware Project," https://www.trustedfirmware.org/.

[9] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, M. Levenhagen, R. Brightwell, A. Gocke, and S. Jaconette, "Palacios: A New Open Source Virtual Machine Monitor for Scalable High Performance Computing," in *Proc. 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[10] "Open Portable Trusted Execution Environment," https://www.op-tee.org/.

[11] "Trusty TEE," https://source.android.com/security/trusty.

[12] B. Kocoloski and J. Lange, "Better Than Native: Using Virtualization to Improve Compute Node Performance," in *Proc. 2nd International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, 2012.

[13] "Pine64," https://www.pine64.org/.

[14] "Raspberry pi," https://www.raspberrypi.org/.

[15] B. Gaines and K. Pedretti, "Weaving Security into Large-Scale HPC Systems," Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2015.

[16] Ayaz Akram. (2021) Trusted Execution for High-Performance Computing. [Online]. Available: http://www.ayazakram.com/papers/eurodw.pdf