BlackBox: A Container Security Monitor for Protecting Containers on Untrusted Operating Systems

Alexander Van’t Hof and Jason Nieh, Columbia University

https://www.usenix.org/conference/osdi22/presentation/vant-hof
BlackBox: A Container Security Monitor for Protecting Containers on Untrusted Operating Systems

Alexander Van’t Hof  
Columbia University

Jason Nieh  
Columbia University

Abstract

Containers are widely deployed to package, isolate, and multiplex applications on shared computing infrastructure, but rely on the operating system to enforce their security guarantees. This poses a significant security risk as large operating system codebases contain many vulnerabilities. We have created BlackBox, a new container architecture that provides fine-grain protection of application data confidentiality and integrity without trusting the operating system. BlackBox introduces a container security monitor, a small trusted computing base that creates protected physical address spaces (PPASes) for each container such that there is no direct information flow from container to operating system or other container PPASes. Indirect information flow can only happen through the monitor, which only copies data between container PPASes and the operating system as system call arguments, encrypting data as needed to protect interprocess communication through the operating system. Containerized applications do not need to be modified, can still make use of operating system services via system calls, yet their CPU and memory state are isolated and protected from other containers and the operating system. We have implemented BlackBox by leveraging Arm hardware virtualization support, using nested paging to enforce PPASes. The trusted computing base is a few thousand lines of code, many orders of magnitude less than Linux, yet supports widely-used Linux containers with only modest modifications to the Linux kernel. We show that BlackBox provides superior security guarantees over traditional hypervisor and container architectures with only modest performance overhead on real application workloads.

1 Introduction

Containers are widely deployed to package, isolate, and multiplex applications on shared computing infrastructure. They are increasingly used in lieu of hypervisor-based virtual machines (VMs) because of their faster startup time, lower resource footprint, and better I/O performance [6, 15, 26, 47]. Popular container mechanisms such as Linux containers rely on a commodity operating system (OS) to enforce their security guarantees. However, commodity OSes such as Linux are huge, complex, and imperfect pieces of software. Attackers that successfully exploit OS vulnerabilities may gain unfettered access to container data, compromising the confidentiality and integrity of containers—an undesirable outcome for both computing service providers and their users.

Modern systems incorporate hardware security mechanisms to protect applications from an untrusted OS, such as Intel Software Guard Extensions (SGX) [30] and Arm TrustZone [2], but they require rewriting applications and may impose high overhead to use OS services. Some approaches have built on these mechanisms to protect unmodified applications [7] or containers [3]. Unfortunately, they suffer from high overhead, incomplete and limited functionality, and massively increase the trusted computing base (TCB) through a library OS or runtime system, potentially trading one large vulnerable TCB for another.

As an alternative, hypervisors have been augmented with additional mechanisms to protect applications from an untrusted OS [11, 12, 27, 35, 67]. This incurs the performance overhead of hypervisor-based virtualization, which containers were designed to avoid. The TCB of these systems is significant, in some cases including an additional commodity host OS, providing additional vulnerabilities to exploit to compromise applications. Theoretically, these approaches could be applied to microhypervisors [10, 61] with smaller TCBS. Unfortunately, microhypervisors still inherit the complexity of hypervisor-based virtualization, including virtualizing and managing hardware resources. The reduction in TCB is achieved through a much reduced feature set and limited hardware support, making their deployment difficult in practice.

To address this problem, we have created BlackBox, a new container architecture that provides fine-grain protection of application data confidentiality and integrity without the need to trust the OS. BlackBox introduces a container security monitor (CSM), a new mechanism that leverages existing hardware features to enforce container security guarantees in a small
trusted computing base (TCB) in lieu of the OS. The monitor creates protected physical address spaces (PPASes) for each container to enforce physical memory access controls, but provides no virtualization of hardware resources. Physical memory mapped to a container’s PPAS is not accessible outside the PPAS, providing physical memory isolation among containers and the OS. Since container private data in physical memory only resides on pages in its own PPAS, its confidentiality and integrity is protected from the OS and other containers.

The CSM repurposes existing hardware virtualization support to run at a higher privilege level and create PPASes, but is itself not a hypervisor and does not virtualize hardware. Instead, the OS continues to access devices directly and remains responsible for allocating resources. This enables the CSM to be minimalistic and simple while remaining performant. By supporting containers directly without virtualization, no additional guest OS or complex runtime needs to run within the secured execution environment, minimizing the TCB within the container itself.

Applications running in BlackBox containers do not need to be modified and can make use of OS services via system calls, with the added benefit of their data being protected from the OS. The monitor interposes on all transitions between containers and the OS, clearing container private data in CPU registers and switching PPASes as needed. The only time in which any container data in memory is made available to the OS is as system call arguments, which only the monitor itself can provide by copying the arguments between container PPASes and the OS. The monitor is aware of system call semantics and encrypts system call arguments as needed before passing them to the OS, such as for interprocess communication between processes, protecting container private data in system call arguments from the OS. Given the growing use of end-to-end encryption for I/O security [55], in part due to the Snowden leaks [36], the monitor relies on applications to encrypt their own I/O data to simplify its design. Once a system call completes and before allowing a process to return to its container, the monitor checks the CPU state to authenticate the process before switching the CPU back to using the container’s PPAS.

In addition to ensuring a container’s CPU and memory state is not accessible outside the container, BlackBox protects against malicious code running inside containers. Only trusted binaries, which are signed and encrypted, can run in BlackBox containers. The monitor is required to decrypt the binaries, so they can only run within BlackBox containers with monitor supervision. The monitor authenticates the binaries before they can run, so untrusted binaries cannot run in BlackBox containers. It also guards against memory-related Iago attacks, attacks that maliciously manipulate virtual and physical memory mappings, that could induce arbitrary code execution in a process in a container by preventing virtual or physical memory allocations that could overwrite a process’s stack.

We have implemented BlackBox on Arm hardware, given Arm’s growing use in personal computers and cloud computing infrastructure along with its dominance on mobile and embedded systems. We leverage Arm hardware virtualization support by repurposing Arm’s EL2 privilege level and nested paging, originally designed for running hypervisors, to enforce separation of PPASes. Unlike x86 root operation for running hypervisors, Arm EL2 has its own hardware system state. This minimizes the cost of trapping to the monitor running in EL2 when calling and returning from system calls because system state does not have to be saved and restored on each trap. We show that BlackBox can support widely-used Linux containers with only modest modifications to the Linux kernel, and inherits support for a broad range of Arm hardware from the OS. The implementation has a TCB of less than 5K lines of code plus a verified crypto library, orders of magnitude less than commodity OSes and hypervisors. With such a reduced size, the CSM is significantly easier for developers to maintain and ensure the correctness of than even just the core virtualization functionality of a hypervisor. We show that BlackBox can provide finer granularity and stronger security guarantees than traditional hypervisor and container architectures with only modest performance overhead for real application workloads.

2 Threat Model and Assumptions

Our threat model is primarily concerned with OS vulnerabilities that may be exploited to compromise the confidentiality or integrity of a container’s private data. Attacks in scope include compromising the OS or any other software to read or modify private container memory or register state, including by controlling DMA-capable devices, or via memory remapping and aliasing attacks. We assume a container does not voluntarily reveal its own private data whether on purpose or by accident, but attacks from other compromised containers, including confidentiality and integrity attacks, are in scope. Availability attacks by a compromised OS are out of scope. Physical or side-channel attacks [5, 32, 43, 52, 68, 69] are beyond the scope of the paper. Opportunities for side-channel attacks are greater in BlackBox than in systems that isolate at a lower level, e.g. VMs. The trust boundary of BlackBox is that of the OS’s system call API, enabling adversaries to see some details of OS interactions such as sizes and offsets.

We assume secure key storage is available, such as provided by a Trusted Platform Module (TPM) [31]. We assume the hardware is bug-free and the system is initially benign, allowing signatures and keys to be securely stored before the system is compromised. We assume containers use end-to-end encrypted channels to protect their I/O data [21, 37, 55]. We assume the CSM does not have any vulnerabilities and can thus be trusted; formally verifying its codebase is future work. We assume it is computationally infeasible to perform brute-force attacks on any encrypted container data, and any encrypted communication protocols are assumed to be designed to defend against replay attacks.
BlackBox enclaves traditional Linux containers to protect the confidentiality and integrity of container data. We refer to a container as being enclaved if BlackBox protects it from the OS. From an application’s perspective, using enclaved containers is little different from using traditional containers. Applications do not need to be modified to use enclaved containers and can make use of OS services via system calls. Container management solutions [48, 49] such as Docker [20] can be used to manage enclaved containers. BlackBox is designed to support commodity OSes, though minor OS modifications are needed to use its enclave mechanism, in much the same way that OS modifications are typically required to take advantage of new hardware features. However, BlackBox does not trust the OS and a compromised OS running enclaved containers cannot violate their data confidentiality and integrity.

BlackBox introduces a container security monitor (CSM), as depicted in Figure 1, which serves as its TCB. The CSM’s only purpose is to protect the confidentiality and integrity of container data in use. It achieves this by performing two main functions, access control and validating OS operations. Its narrow purpose and functionality makes it possible to keep the CSM small and simple, avoiding the complexity of many other trusted system software components. For example, unlike a hypervisor, the CSM does not virtualize or manage hardware resources. It does not maintain virtual hardware such as virtual CPUs or devices, avoiding the need to emulate CPU instructions, interrupts, or devices. Instead, interrupts are delivered directly to the OS and devices are directly managed by the OS’s existing drivers. It also does not do CPU scheduling or memory allocation, making no availability guarantees. The CSM can be kept small because it presumes the OS is CSM-aware and relies on the OS for complex functionality such as bootstrapping, CPU scheduling, memory management, file systems, and interrupt and device management.

To enclaved containers, the CSM introduces the notion of a protected physical address space (PPAS), an isolated set of physical memory pages accessible only to the assigned owner of the PPAS and the CSM. Each page of physical memory is mapped to at most one PPAS. The CSM uses this mechanism to provide memory access control by assigning a separate PPAS to each enclaved container, thereby isolating the physical memory of each container from the OS and any other container. The OS determines what memory is allocated to each PPAS, but cannot access the memory contents of a PPAS. Similarly, a container cannot access a PPAS that it does not own. Memory not assigned to a PPAS, or the CSM, is assigned to and accessible to the OS. The CSM itself can access any memory, including memory assigned to a PPAS. Within a PPAS, addresses for accessing memory are the same as the physical addresses on the machine; physical memory cannot be remapped to a different address in a PPAS. For example, if page number 5 of physical memory is assigned to a PPAS, it will be accessed as page number 5 from within the PPAS. Container private data in memory only resides on pages mapped to its own PPAS, therefore its confidentiality and integrity is protected from the OS and other containers. Section 4 describes how BlackBox uses nested page tables to enforce PPASes.

To support OS functionality that traditionally requires access to a task’s CPU state and memory, the CSM provides an application binary interface (ABI) for the OS to request services from the CSM. The CSM ABI is shown in Table 1. For example, create_enclave and destroy_enclave are called by the OS in response to requests from a container runtime, such as runC [29], to enclave and unenclave containers, respectively. For CSM calls that require dynamically allocated memory, the OS must allocate and pass in the physical address of a large enough region of contiguous memory to perform the respective operation. Otherwise, the call will fail and return the amount of memory required so that the OS can
Table 1: BlackBox Container Security Monitor ABI

<table>
<thead>
<tr>
<th>CSM Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>create_enclave</td>
<td>Create new enclave for a container.</td>
</tr>
<tr>
<td>destroy_enclave</td>
<td>Destroy enclave of a container.</td>
</tr>
<tr>
<td>protect_vectors</td>
<td>Validate OS exception vectors.</td>
</tr>
<tr>
<td>alloc_iopgtable</td>
<td>Allocate I/O device page table.</td>
</tr>
<tr>
<td>free_iopgtable</td>
<td>Free I/O device page table.</td>
</tr>
<tr>
<td>set_iopt</td>
<td>Update entry in I/O device page table.</td>
</tr>
<tr>
<td>get_ioaddr</td>
<td>Get physical address for I/O virtual address.</td>
</tr>
<tr>
<td>enter_os</td>
<td>Context switch CPU to OS.</td>
</tr>
<tr>
<td>exit_os</td>
<td>Context switch CPU from OS.</td>
</tr>
<tr>
<td>set_vma</td>
<td>Update virtual memory areas of a process/thread.</td>
</tr>
<tr>
<td>set_pt</td>
<td>Update page table entry of a process/thread.</td>
</tr>
<tr>
<td>copy_page</td>
<td>Copy contents of a page to a container.</td>
</tr>
<tr>
<td>task_clone</td>
<td>Run new process/thread in a container.</td>
</tr>
<tr>
<td>task_exec</td>
<td>Run in new address space in a container.</td>
</tr>
<tr>
<td>task_exit</td>
<td>Exit a process or thread in a container.</td>
</tr>
<tr>
<td>futex_read</td>
<td>Read the value of a futex in a container.</td>
</tr>
</tbody>
</table>

3.2 Enclaved Container Initialization

To securely initialize an enclaved container, an image that is to be used for such a container must first be processed into a BlackBox container image, using a process similar to how Amazon enclaves are created using Docker images [1]. BlackBox provides a command line tool build_bb_image, which can be used by a cloud customer, that takes a Docker image, finds all executable binary files contained within the image, and encrypts the sections containing the code and data used by the code using the public key paired with a trusted private key stored in the secure storage of the host and accessible only by the CSM. These encrypted sections are then hashed and their hash values recorded along with the binaries they belong to. These values are then signed with the private key of the container image’s creator whose paired public key is accessible in the secure storage of the host to ensure authenticity and bundled with the container image for later reference during process creation, as described in Section 3.3. This ensures the binaries cannot be modified without being detected, or run unless decrypted by the CSM. Other than additional hashes and using encrypted binaries, the BlackBox container image contains nothing different from a traditional Docker image.

To start a container using a BlackBox container image, the container runtime is modified to execute a simple shim process in place of the container’s specified init process. The container runtime passes the shim the path of the init process used by the container along with any arguments and its environment. The shim is also given the signed binary hash information bundled with the container image. The shim process runs a tiny statically linked program that initiates a request to the OS to call the create_enclave CSM call before executing the original init process, passing the signed hash information to the CSM as part of the call. Other than the shim process, which exits upon executing the init process, there is no additional code that runs in a BlackBox container.
Beyond vanilla Linux containers. There are no additional libraries and no need for a library OS, avoiding the risks of bloating the TCB of the container itself.

`create_enclave` creates a new enclave using the BlackBox container image and returns with the calling process running in the enclaved container, the return value of the call being the new enclave’s identifier. `create_enclave` performs the following steps. First, it creates a new PPAS for the container. Second, it freezes the userspace memory of the calling process so it, and its associated page tables, cannot be directly changed by the OS, then moves all of its pages of physical memory into the container’s PPAS so that they are no longer accessible by the OS. Finally, it checks the contents of the loaded shim binary in memory against a known hash to validate the calling process is the expected shim process.

After returning from `create_enclave`, the shim executes the container’s init process from within the container. Since the container’s init process obtains its executable from the BlackBox container image whose code and data are encrypted, the OS may load it, but cannot actually execute it without the CSM using its private key to decrypt it. Further details on `exec` with encrypted binaries are described in Section 3.6. In this way, the OS is incapable of running a BlackBox container image without the CSM. Therefore, if it is running, the CSM must be involved and protecting it. Because the CSM itself is securely booted and enclave code is encrypted and only runnable by the CSM, an unbroken chain of trust is established enabling remote attestation similar to that of other security systems, such as Samsung Knox [56].

The container runtime calls `destroy_enclave` to remove the enclave of a container, which terminates all running processes and threads within the container to ensure that any container CPU state and memory is cleared and no longer accessible to the OS or any other container before removing the enclave. The container is effectively returned to the same state it was in before `create_enclave` was called.

### 3.3 Enclaved Task Execution

BlackBox supports the full lifecycle of tasks executing in enclaved containers, including their dynamic creation and termination via standard system calls such as `fork`, `clone`, `exec`, and `exit`. This includes tracking which tasks are allowed to execute in which containers. This is achieved by requiring the OS to call a set of CSM calls, `task_clone` on task creation via `fork` and `clone`, `task_exec` when loading a new address space via `exec`, and `task_exit` when a task exits via `exit`. These calls request the CSM to perform various functions related to task execution that the OS is not able to do because it does not have access to task CPU state and memory. If the OS does not make the respective CSM call, the created task and executed binary will simply not run in its enclave and therefore will not have access to its data. These calls update the enclaved task array, the index of which is used as the enclaved task identifier. Each entry in the array includes the enclave identifier of the container in which the task executes, as well as the address of the page table used by the task as discussed earlier.

When a task running in an enclaved container creates a child task via a system call, the OS calls `task_clone` with the enclaved task identifier of the calling task and a flag indicating whether the new task will share the same address space as the caller, as when creating a thread, or have its own copy of the address space of the caller, as when creating a process. In the latter case, new page tables will be allocated for the child task and the CSM will ensure that they match those of the caller’s and cannot be directly modified by the OS. The CSM will also confirm that the calling task issued the task creation system call. If all checks pass, the CSM will create a new entry in the enclaved task array with the same enclave identifier as the calling process, and return the array index of the new entry as the identifier for the task. The entry will also contain the address of the task’s page table, which will be the same as the caller’s entry if it shares the same address space as the caller.

When the OS runs the child and the task returns from the OS, the OS provides the CSM with the enclaved task’s identifier. The CSM then looks up the task in its enclaved task array using this identifier and confirms that the address of the page table stored in the entry matches the address stored in the page table base register of the CPU. If the checks pass, it will then restore CPU state and switch the CPU to the container’s PPAS, thereby allowing the task to resume execution in the container. If the OS does not call `task_clone`, then upon exiting the OS, the task’s PPAS would not be installed and it would fail to run.

On `exec`, the calling task will replace its existing address space with a new one. The OS calls `task_exec`, which, like `task_clone` for `fork`, creates a new enclaved task entry with a new address space. The difference is that the new address space is validated by ensuring that the new process’ stack is set up as expected and the executable binary is signed and in the BlackBox container image, as described in Section 3.6. After creating the new enclaved task entry, the original address space is disassociated from the container, scrubbing any memory pages to be returned to the OS and removing them from the container’s PPAS.

On `exit`, the OS will call `task_exit` so the CSM can remove the enclaved task entry from the enclaved task array. If an address space has no more tasks in the container, the CSM disassociates it in a similar manner to the `exec` case.

### 3.4 Memory

BlackBox prevents the OS from directly accessing a container’s memory, but relies on the OS for memory management, including allocating memory to tasks in the container. This avoids introducing complex memory management code into BlackBox, keeping it small and simple, but means that BlackBox also needs to protect against memory-based Iago
attacks [9] by the untrusted OS through manipulation of system call return values. For example, if a process calls mmap, it expects to receive an address mapping that does not overlap with any of its existing mappings. If the OS were to return a value overlapping the process’s stack, it could manipulate the process into overwriting a return address on its stack through a subsequent read with an attacker controlled address, opening the door for return-oriented-programming [53] and return-into-libc [58] attacks. Furthermore, the OS may return an innocuous looking non-overlapping virtual address from mmap, but still maliciously map the returned address to the physical page the stack is on.

To rely on the OS for memory management while preventing memory-based Iago attacks, BlackBox protects the container’s memory at the application level by preventing the OS from directly updating per process page tables. It instead requires the OS to make requests to the CSM to update process page tables, allowing the CSM to reject updates if the OS behaves incorrectly. Figure 2 depicts how a container’s page table is updated during a page fault. When a process in a container faults on a page, an exception causes control to transfer to the OS by way of the CSM (steps 1-3). The OS then allocates a page for the process, but instead of updating the process page table directly, it performs a set_pt CSM call (step 4). Upon receiving the set_pt call, the CSM verifies if the allocation is acceptable (step 5). To do so, the CSM maintains a list of valid mappings for each process. This list is maintained by interposing on system calls that adjust memory mappings. In Linux these calls include mmap and brk. Prior to writing the page table entry, the CSM first verifies that the virtual address specified belongs to a valid mapping. If it does not, the update is rejected. Second, the CSM checks if the physical page assigned is already in the container’s PPAS and therefore already in use. This can commonly occur innocuously when, e.g., two processes in a container have the same file mapped in their address spaces. However, to prevent the risk of a malicious OS coercing an enclave to overwrite existing memory via a malicious memory allocation, the CSM marks any physical page mapped more than once read only in the container’s PPAS, unless it was inherited from a parent as part of process creation in which case it can be trusted. While this is effective at preventing these attacks, the downside is that writes to such memory will trap and need to be handled by BlackBox; for simplicity, BlackBox disallows writable memory-mapped file I/O as it is uncommonly used. Finally, if the virtual address is valid and not mapped to an existing physical page in a container’s PPAS, the CSM unmaps the assigned physical page from the OS and maps it into the container’s PPAS. The CSM then updates the page table entry on the OS’s behalf (step 6). Control is then returned back to the OS (step 7). When returning control back to the process that faulted, the process’s container PPAS will be switched to (steps 8-10). Section 4 describes further details about this process. The CSM also invalidates TLB entries as needed when it performs page table updates, ensuring that a malicious OS cannot violate a container’s PPAS through stale TLB entries.

BlackBox provides support for copy-on-write (CoW) memory, a key optimization commonly used in OSes. The OS traditionally expects to be able to share a page in memory among multiple processes and when a write is attempted, break the CoW by copying the contents of the page to a new page assigned to the process. With BlackBox, the OS does not have the ability to copy container memory though, so the OS instead makes a copy_page CSM call to have the CSM perform the CoW break on its behalf. The CSM will check that the source page belongs to the container’s PPAS and the destination page is in the OS’s memory. If so, it will move the destination page into the container’s PPAS and perform the copy.

BlackBox supports the dynamic release of memory back to the OS as tasks adjust their heap, unmap memory regions, and exit, while preserving the privacy and integrity of a container’s memory. As with memory allocation, system calls that can allow for returning of an application’s memory, like munmap and _exit are tracked to maintain an accurate view of a container’s memory mappings. During these calls, the OS may attempt to free pages allocated to the process. In doing so, as with memory allocation, it must make use of the set_pt CSM call since it cannot update page tables directly. The CSM will then check if the application has made a call to release the specified memory and reject the update if it has not. If the update is valid, the CSM will perform the page table update, and if no longer needed, scrub the page and remove it from the container’s PPAS.

While BlackBox ensures that container memory is not accessible to the OS, many OS interactions via system calls expect to use memory buffers that are part of an application’s memory to send data to, or receive data from, the OS. BlackBox treats the use of such memory buffers in system calls as implicit directives to declassify the buffers so they can be shared with the OS. To support this declassification while ensuring that a container’s PPAS is not accessible to the OS, BlackBox provides a syscall buffer for each task running in an enclaved container that is outside of the container’s PPAS and accessible to the OS. When interposing...
on a system call exception, the CSM replaces references to memory buffers passed in as system call arguments with those to the task’s syscall buffer. For buffers that are used to send data to the OS, the data in those buffers is copied to the syscall buffer as well. When returning to the container, the references to the syscall buffer are replaced with those to the original memory buffers. For buffers that are used to receive data from the OS, the data in the syscall buffer is copied to the original memory buffers as well.

Most system calls are interposed on by a single generic wrapper function in the CSM that uses a table of system call metadata to determine which arguments must be altered. System calls with more complex arguments, like those involving iovec structures are interposed on with more specific wrapper functions. On Linux, this interposing and altering of arguments works for most system calls with a few notable exceptions as discussed in Section 3.5.

As part of the copying of data from the OS to an enclaved container, BlackBox also does simple checks on system call return values to ensure they fall within predefined correct ranges. This has been shown to protect against many Iago attacks [14]. However, to keep its TCB simple and small, BlackBox only guarantees the correctness of system call semantics for memory management and inter-process communication (IPC), the latter discussed in Section 3.5. As a result, BlackBox protects against Iago attacks related to memory management and IPC, but is susceptible to some other Iago attacks. Augmenting BlackBox with a user-level runtime library in an enclaved container that guarantees the correctness of system call semantics could improve Iago attack protection, but at the cost of a larger TCB and potential additional limitations on system call functionality.

### 3.5 Inter-process Communication

While BlackBox declassifies data to the OS passed in as system call arguments, it protects inter-process communication (IPC) among tasks running in the same enclaved container by encrypting the data passed into IPC-related system calls. This protects applications using IPC, which is transferred through and accessible to the OS. System calls that can create IPC-related file descriptors, such as pipe, and Unix Domain Sockets are interposed on and their returned file descriptors (FDs) recorded in per-process arrays marking them as related to IPC. When the CSM interposes on system calls that pass data through FDs, like write and sendmsg, it checks if the given FD is one related to IPC for that process. If it is, the CSM first uses authenticated encryption with a randomly generated symmetric key created during container initialization to encrypt the data before moving it into the task’s syscall buffer. A record counter, incremented on each transaction, is included as additional authenticated data to prevent the host from replaying previous transactions. Similarly, data is decrypted and authenticated when interposing on system calls like read and recvmsg before copying it to the calling process’s PPAS. With this mechanism, IPC communication is transparently encrypted and protected from the OS.

As mentioned in Section 3.4, to avoid trusting the OS’s memory allocations, memory pages that are used by more than one process in a container are marked read-only in the container’s PPAS unless the pages are known to belong to a shared memory mapping and are inherited during process creation. Shared memory regions created by a parent process through mmap with MAP_SHARED and faulted in prior to forking can be written to by both parent and child processes since the child’s address space is validated after fork, as discussed in Section 3.3. However, for simplicity, BlackBox does not allow for writable IPC shared memory via XSI IPC methods such as shmat and shm_open, which are no longer widely-used. Modern applications instead favor thread-based approaches for performance or shared mappings between child worker processes via mmap compatible with BlackBox.

Futexes are used among threads and processes to synchronize access to shared regions of memory. As part of the design of futex, the OS is required to read the futex value, which is in the process’s address space and included in the respective container’s memory. This direct access to container memory is incompatible with BlackBox’s memory isolation. To support futex, the OS makes a futex_read CSM call to obtain the value of a futex for container processes, rather than try and access the memory directly. The CSM ensures that only the futex address passed to futex can be read, and only if a futex call has been made.

Signals, used to notify processes of various events, present two issues for BlackBox. First, when delivering a signal to a process, a temporary stack for the signal handler is set up in the process’s memory. With enclaved containers, this memory is not accessible to the OS. To remedy this, the OS is modified to setup this stack in a region of memory outside of the container’s PPAS, which is then moved to the PPAS when the signal handler is executed and returned to the OS when the signal handler returns via rt_sigtreturn. Second, the OS has to adjust the control flow of the process to execute the signal handler instead of returning to where it was previously executing. BlackBox cannot allow the OS to adjust the control flow of an enclaved process without validating it is doing so properly. To achieve this, as part of the CSM interposing on system calls, it tracks signal handler installation via system calls such as rt_sigaction. Upon handling a signal, the CSM ensures that the process will be correctly returning to a registered handler.

### 3.6 Container File System

Files within a container can only be accessed through an OS’s I/O facilities making access to a container’s files inherently untrustworthy without additional protection. A userspace encrypted file system could potentially be used to provide transparent protection of file I/O, but this would likely signif-


incrementally increase the container’s TCB. BlackBox relies on applications to use encryption to fully protect sensitive data files within a container, and provides a simple mechanism to allow the OS to load encrypted executable binaries for execution.

As discussed in Section 3.2, container images for BlackBox are pre-processed. For example, ELF binaries, widely-used on Linux, have .text, .data, and .rodata sections that contain the executable code and data used by the code. These sections are combined into various segments when loaded into memory. In a BlackBox container image, the ELF headers are left unencrypted, but the .text, .data, and .rodata sections are encrypted then hashed, and their hash values are recorded along with the binaries. This enables BlackBox to validate the integrity and authenticity of executable binaries.

An ELF binary is executed by the OS as a result of a process calling exec, upon which the OS loads the binary by mapping its ELF headers into memory, reading the ELF headers to determine how to process the rest of the binary, then mapping the segments of the binary to memory. As discussed in Section 3.3, the OS is required to call task_exec, which passes the virtual addresses of the binary’s loaded segments containing the .text, .data, and .rodata sections to the CSM. During this call, the CSM moves the process’s pages, corresponding to the loaded binary, into the container’s PPAS, validates that the hashes of the encrypted .text, .data, and .rodata sections match the hashes for the given binary from the BlackBox container image to confirm the authenticity and integrity of the loaded segments, then decrypts the sections in memory. The virtual to physical address mappings of these binary segments are recorded for later use. Upon returning from task_exec, the OS will begin running the task whose binary is now decrypted within protected container memory. If checking the hashes or decryption fails, the CSM will refuse to run the binary within an enclaved container, ensuring only trusted binaries can run within.

For dynamically linked binaries, in addition to the binary segments the OS maps during exec, the OS also maps the segments of the loader in the process’s address space. These segments are verified in the same manner as the binary’s segments. Dynamically linked binaries load and execute external libraries that BlackBox must validate are as expected and trusted. During the container image creation process, as with executable binaries, library binaries are also encrypted preventing their use without the CSM. These libraries are loaded and linked at runtime in userspace by a loader that is part of the trusted container image. To do this, the loader, running as part of a process’s address space, mmap library segments into memory. The CSM intercepts these mmap by interposing on FD-related system calls, such as open. If an FD is created for one of the libraries within a container, as recorded during container image creation, the CSM marks that FD as associated with the given library. If this FD is then used with mmap, the CSM intercepts it. Based on the size of the mmap request and the protection flags used, the CSM can infer which segment the loader is mapping. If it is a segment containing one of the encrypted sections, the CSM performs the same hashing, decryption, and memory map recording as it does with executable binaries.

4 Implementation

We have implemented a BlackBox prototype by repurposing existing hardware virtualization support available on modern architectures, including a higher privilege level, usually reserved for hypervisors, and nested page tables (NPTs). NPTs, also known as Arm’s Stage 2 page tables and Intel’s Extended Page Tables (EPT), is a hardware-assisted virtualization technology that introduces an additional level of virtual address translation [8]. When NPTs are used by hypervisors, the guest OS in a VM manages its own page table to translate a virtual address to what the VM perceives as its physical address, known as a guest physical address, but then the hypervisor manages an NPT to translate the guest physical address to an actual physical address on the host. Hypervisors can thereby use NPTs to control what physical memory is available to each VM.

BlackBox uses hardware virtualization support to run the CSM in lieu of a hypervisor to support PPASes. The CSM runs at the higher hypervisor privilege level, so that it is strictly more privileged than the OS and is able to control NPTs. The CSM introduces an NPT for each container and the OS, such that a container’s PPAS is only mapped to its own NPT, isolating the physical memory of each container from the OS and each other. The CSM switches a CPU from one PPAS to another by simply updating its NPT base register to point to the respective container’s NPT. Similarly, the CSM uses NPTs to protect its own memory from the OS and containers by simply not mapping its own memory into the NPTs. The memory for the NPTs is part of the CSM’s protected memory and is itself not mapped into any NPTs so that only the CSM can update the NPTs. When the CSM runs, NPTs are disabled, so it has full access to physical memory.

Specifically, BlackBox uses Arm hardware virtualization extensions (VE) [16–19]. The CSM runs in Arm’s hypervisor (EL2) mode, which is strictly more privileged than user (EL0) and kernel (EL1) modes. EL2 has its own execution context defined by register and control state, and switching the execution context of EL0 and EL1 are done in software. The CSM configures Stage 2 page tables in EL2, and the System Memory Management Unit (SMMU), Arm’s IOMMU. The Linux kernel runs in EL1 and has no access to EL2 registers, so it cannot compromise the CSM. CSM calls are made using Arm’s hvc instruction from EL1.

Before and after every transition to the OS, BlackBox traps to the CSM, which in turn switches between container and OS NPTs. One might think that imposing two context switches to the CSM to swap NPTs for every one call to the OS would be prohibitively expensive, but we show in Section 5
that this can be done on Arm without much overhead. The
flexibility that Arm EL2 provides of allowing software
to determine how execution context is switched between
hypervisor and other modes turns out to be particularly
advantageous for implementing the CSM because it does
not lock its implementation into using heavyweight hardware
virtualization mechanisms to save and restore hypervisor
execution context that are not required for the CSM.

Trapping to the CSM before and after every transition to the
OS requires that the CSM interpose on all system calls, inter-
rupts, and exceptions. Hypervisors traditionally accomplish
similar functionality by trapping interrupts and exceptions to it-
self, then injecting virtual interrupts and exceptions to a VM.
BlackBox avoids the additional complexity of virtualizing
interrupts and exceptions by taking a different approach. The
CSM configures hardware so system calls, interrupts, and ex-
ceptions trap to the OS and modifies the OS’s exception vector
table for handling these events so that enter_os and exit_o
s CSM calls are always made before and after the actual OS
event handler. To guarantee these handlers are installed and
not modified by the OS at a later time, BlackBox requires the
OS to make a protect_vectors CSM call with the address of
the text section of the vector table during system initialization,
before any container may be enclaved. The CSM then
prevents the OS from tampering with the modified vector ta-
bale by marking its backing physical memory read only in the
OS’s NPT. Similarly, the vDSO region of memory is marked
read only to prevent malicious tampering of the region.

Figure 3 depicts the steps involved in interposing on transi-
tions between the containers and OS when repurposing virtu-
alization hardware. While running in a container, an exception
occurs transferring control to the protected OS exception
vector table (step 1). All entry points in the exception vector
table invoke the enter_os CSM call (step 2). During this, the
CSM switches to the OS’s NPT (step 3). The OS will there-
fore not be able to access private physical memory mapped
into container NPTs. For system call exceptions, system call
arguments are copied to an OS accessible syscall buffer (step
4). Control is transferred back to the OS (step 5) to perform
the required exception handling. When the OS has finished
handling the exception, the exit_os CSM call is made as
part of the return path of the exception vectors when returning
to userspace (step 6). For system call exceptions, OS updated
arguments are copied back to the original buffer (step 7). On
exit_os, the CSM verifies the exception return address to
ensure the call is from the trusted exception vectors, which the
OS cannot change, rejecting any that are not. The CSM then
checks if the running task belongs to an enclaved container,
in which case the CSM switches to the respective container’s
NPT so the task can access its PPAS memory state (step 8).
Control is restored to the container by returning from exit_o
s (step 9) and back to userspace (step 10). If exit_os is not
called, the CSM will not switch the CPU to use the container’s
PPAS, so its state will remain inaccessible on that CPU.

BlackBox protects a container’s memory by using separate
NPTs for the OS and each container, but still relies on the OS
to perform all complex memory management functions, such
as allocation and reclamation, to minimize the complexity
and size of the CSM. This is straightforward because unlike
hypervisors which virtualize physical memory using NPTs,
the CSM merely uses NPTs for access control so that the
identity mapping is used for all NPTs including the OS’s
NPT. The OS’s view of memory is effectively the same as
the actual physical memory for any physical memory
mapped into the OS’s NPT. Except for the CSM’s physical
memory, all physical memory is initially assigned to the
OS and mapped to its NPT. When the OS allocates physical
memory to processes in containers, the CSM can just unmap
the physical memory from the OS’s NPT and map it to the
respective container’s NPT at the same address. The CSM
does not need its own complex allocation functionality. The
CSM checks the OS’s NPT to make sure that the OS has
the right to allocate a given page of memory. For example,
should the OS attempt to allocate a physical page belonging
to the CSM, the CSM will reject the allocation and not update
the OS’s or container’s NPT. The CSM also checks that any
page allocation proposed by the OS for a container is not
mapped into the IOMMU page tables and will therefore not
be subject to DMA attacks, as discussed in Section 3.1.

Note that the OS is oblivious to the fact that its allocation
decisions for process page tables, Arm’s Stage 1 page tables,
are also used for Stage 2 page tables. Furthermore, since
Arm hardware first checks Stage 1 page tables before Stage
2 page tables, page faults due to the need to allocate physical
memory to a process all appear as Stage 1 page faults, which
are handled in the normal way by the OS’s page fault handler.
Since the CSM maps the physical memory to the respective
Stage 1 and Stage 2 page table entries at the same time, there
are no Stage 2 page faults for memory allocation.

As discussed in Section 3.4, BlackBox requires that pro-
cess page tables cannot be directly modified by the OS. At the
same time, commodity OSes like Linux perform many opera-
tions that involve walking and accessing process page tables.
To minimize OS modifications required to use enclaved con-
tainers, BlackBox makes the process page tables readable but
not writable by the OS by marking the corresponding entries
in the OS’s NPT read only. All existing OS code that walks
and reads process page tables can continue to function without modification, and only limited changes are required to the OS to use CSM calls for any updates to process page tables. A process’s page tables are also mapped to its respective container’s NPT, so they can be accessed by MMU hardware for virtual address translation while executing the process. BlackBox also maps tasks’ syscall buffers, used for passing system call arguments to and from the OS, to their Stage 1 page tables. This allows OS functions designed to copy data to and from buffers in the calling process’s address space to function correctly without modification. The tasks’ syscall buffers themselves are only mapped to the OS’s NPT, not the container’s NPT, as they are shared directly only by the CSM and OS.

To optimize TLB usage, physically contiguous memory can be mapped to an NPT in blocks larger than the default 4 KB page size. The BlackBox implementation supports transparent 2 MB stage 2 block mappings by first fully populating the last-level stage 2 page table with 4 KB mappings, then folding all 512 entries into a single entry. BlackBox checks that all 512 entries are contiguous in physical memory and that the first entry is aligned to a 2 MB boundary. BlackBox will unfold a block mapping if one of the original 512 entries is unmapped, such that all 512 entries are no longer contiguous in physical memory. Similarly, BlackBox will unfold a block mapping if there is a need to change the attributes of one of the original 512 entries, such as marking it read only while other entries remain writable. This approach is advantageous over just supporting huge pages allocated by the OS because it improves TLB usage even when the OS does not use huge pages.

Although BlackBox is designed to work using existing hardware virtualization support, the upcoming Armv9 architecture with its inclusion of the Arm Confidential Compute Architecture (CCA) [41] offers alternative mechanisms that may be used for implementing BlackBox. CCA introduces secure execution environments called Realms. The memory and execution state of these Realms are inaccessible to existing privileged software like OSes and hypervisors guaranteeing their confidentiality and integrity from them. Realms are supported by a separate Realm World and managed by a Realm Management Monitor (RMM) running in EL2 within the Realm World giving it full access to Realm memory and CPU state as well as control over their execution. Although Realms are currently only designed to support VMs, it may be possible to use them to support enclave containers by integrating the functionality of the CSM with the RMM and extending its ABI to encompass the CSM’s ABI.

BlackBox’s implementation is relatively small. The implementation is less than 10K lines of code (LOC), most of which is the 5K LOC for the implementation of Ed25519, ChaCha20, and Poly1305 from the verified HACL* crypto library [70]. Other than HACL*, BlackBox consisted of 4.9K LOC, all in C except for 0.4K LOC in Arm assembly. Table 2 shows a breakdown by feature. 0.3K LOC was for verifying the CSM was correctly booted and initialized. 1K LOC was for enclave management, including enclave creation and handling enclave metadata. 0.1K LOC was for switching between enclaves and the OS. 0.2K LOC was for protecting data in CPU registers. 1K was for system call interposition, including marshaling of arguments. The table used for determining how to marshal system calls and check return values is dynamically generated as a single line of C code at compile time. 2.3K LOC was for memory protection, including NPT management of PPAses, Iago and DMA protection, and handling and validating page table update requests. BlackBox’s CSM TCB implementation complexity is similar to other recently verified concurrent systems [39–41, 63], suggesting that it is small enough that it can be formally verified. Beyond the CSM itself, only 0.5K LOC were modified or added to the Linux kernel to support BlackBox.

Table 2 also compares the code complexity of BlackBox versus the Linux kernel and KVM hypervisor. This is a conservative comparison as the LOC for Linux and KVM only include code compiled into the actual binaries for one specific Arm server used for the evaluation in Section 5. Even with this conservative comparison, BlackBox is orders of magnitude less code, in part because its functionality is largely orthogonal to both OSes and hypervisors, which have much more complex functionality requirements.

### 5 Experimental Results

We quantify the performance of BlackBox compared to widely-used Linux containers, and demonstrate BlackBox’s ability to protect container confidentiality and integrity. Ex-
Table 3: Microbenchmarks and Application Workloads

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hackbench</td>
<td>hackbench [54] using Unix domain sockets and 100 process groups running in 500 loops.</td>
</tr>
<tr>
<td>Apache</td>
<td>Apache v2.4.46 server handling 100 concurrent requests from remote ApacheBench [64] v2.3 client, serving the 12 KB default Debian index.html.</td>
</tr>
<tr>
<td>HAProxy</td>
<td>HAProxy v1.8.19 server proxying 100 concurrent requests from remote ApacheBench [64] v2.3 client to remote Apache v2.4.29, serving the 82 KB index.html of the GCC 13.0.0 manual.</td>
</tr>
<tr>
<td>Kernbench</td>
<td>Compilation of the Linux 5.4 kernel using allnoconfig for Arm with GCC 8.3.0.</td>
</tr>
<tr>
<td>Memcached</td>
<td>memcached v1.6.9 using the memtier [51] benchmark v1.3.0 with default parameters.</td>
</tr>
<tr>
<td>MySQL</td>
<td>MariaDB v10.3.27, a MySQL fork, handling requests from remote YCSB [13] v0.17.0 client running workload A with 200 parallel transactions, recordcount=500K, and opcount=100K.</td>
</tr>
<tr>
<td>Netperf</td>
<td>netperf v2.6.0 [33] running netserver on the server and the client with default parameters in three modes: TCP_STREAM (receive throughput), TCP_MAERTS (send throughput), and TCP_RR (latency).</td>
</tr>
<tr>
<td>Nginx</td>
<td>Nginx v1.18.0 server handling 100 concurrent requests from remote ApacheBench [64] v2.3 client, serving the 12 KB default Debian index.html.</td>
</tr>
</tbody>
</table>

5.1 Performance Measurements

Figure 4 shows performance measurements for each microbenchmark and application workload for each container configuration normalized to native execution; lower numbers are better. Solid bars indicate results run on the Raspberry Pi and the overlaid outlined bars indicate results run on the AMD Seattle Arm server. BlackBox has the highest overhead relative to native execution on the null system call measurement, but most of the overhead is from Docker, due to its use of seccomp to configure and limit the system calls available in a container to reduce the available attack surface area. Although seccomp is used for all system calls, its overhead is most apparent for the null system call as its base cost is the lowest since it does no work. In contrast, the overhead due to BlackBox, from the two CSM calls that BlackBox makes on every system call, is small relative to seccomp. Although CSM calls require switching to and from Arm’s EL2 mode, it requires no more than EL2’s system register state to execute, eliminating the need to save and restore system registers when switching between EL1 and EL2; only general-purpose registers need to be saved and restored. Taking advantage of Arm’s architectural features makes CSM calls relatively inexpensive, enabling fine-grained container protection without significant overhead from system call interposition. The key aspect of Arm’s design that is crucial for the CSM is that software determines what state needs to be saved and restored. Running the CSM in the equivalent x86 hypervisor root mode would be much more expensive as it provides a hardware instruction that must be used to context switch to root mode that requires saving and restoring the entire CPU system state [17]. The x86 mechanism works well for hypervisors since they already require this operation, but poorly for the CSM which makes minimal use of CPU system state, and therefore does not need the expensive save and restore.

For the read, write, stat, open/close, and select system call measurements, BlackBox Enclaved is less than two times the cost of Docker. The overhead for the BlackBox configurations were used to quantify the cost of different protection mechanisms. BlackBox NS provides the same security as Docker, the only difference being that BlackBox NS runs the containers on BlackBox with the OS’s NPT enabled, to quantify NPT overhead. BlackBox NE provides stronger security by enclaving the container but without enabling IPC encryption, thereby quantifying BlackBox overhead without IPC encryption. BlackBox Enclaved is the same as BlackBox NE but with IPC encryption enabled. When using BlackBox, its DMA protection is not available on the Raspberry Pi 4 because it has no SMMU. Docker’s default seccomp policy is enabled for all configurations. Versions of libseccomp prior to v2.5 had a significant performance issue on policies like Docker’s default [65]. The Docker version we use incorporates this performance fix.
enclaved configurations is due to the need to copy system call arguments back and forth between the container PPAS and OS, since enclaved container memory is not accessible to the OS. open additionally incurs overhead as part of checking the path being opened to identify FDs associated with shared libraries as part of BlackBox’s binary decryption mechanism. For all system calls, the overhead on the AMD server, as indicated by the outlined bars, exceeds that of the Raspberry Pi’s. In most cases, this is due to the server hardware performing the CPU bound system call operations more quickly than the Raspberry Pi while their memory performance remains similar, resulting in the similar costs for BlackBox’s system call argument copying having relatively higher overhead.

fork and exec measurements show the highest overhead for BlackBox Enclaved versus Docker, less than three times the cost of Docker. This is due to validating that the new process’s address space matches its parent’s on fork, and additionally validating the address space against the new binary’s mappings on exec. Although the binary must be decrypted for exec measurements, it is only decrypted once and all subsequent iterations just confirm the mappings match the first’s, thereby amortizing the cost of the initial decryption.

Page fault measurements show the one microbenchmark for which there is noticeable overhead for BlackBox NS versus Docker. This is due to the added cost of using NPTs for the BlackBox NS configuration. This overhead then increases for enclaved containers due to needing to verify the fault resides within a known address mapping to protect the container from potential lago attacks from the OS. Although a page fault results in several context switches to the CSM, the context switches themselves are not a significant cost because they are relatively inexpensive on Arm.

Protecting container IPC communication through encryption imposes little cost for most workloads, but this overhead is noticeable for pipe, UNIX domain sockets (AF_UNIX), and hackbench measurements. These benchmarks represent worst-case overheads for IPC encryption because they all use IPC to read and write a single byte to signal other processes. When encrypting, this single byte is padded and written along with authentication data, significantly increasing the relative write size and affecting read/write latency measurements. In contrast, the context switch microbenchmark, in which a parent process spawns two child processes that communicate between each other with pipes, has almost no overhead. In this case, 4 byte reads and writes are used so the extra data that encryption adds, and therefore the time to complete the calls, is relatively less, and context switching and rescheduling dominates IPC encryption costs. The signaling microbenchmarks do not involve any encryption. BlackBox Enclaved overhead for signal installation is due to copying the sigaction struct in and out, and for signal delivery is due to verifying the control flow.

Apache, HAProxy, Kernbench, memcached, MySQL, and Nginx measurements show that BlackBox overhead is much less on realistic application workloads than microbenchmarks. In most cases, BlackBox Enclaved overhead versus native execution is less than 15% on both the Raspberry Pi and AMD server, demonstrating modest overhead across both Arm embedded and server hardware. As indicated by the BlackBox NS measurements, NPT usage is a source of overhead, though more so on the Raspberry Pi than the AMD server. Apache, HAProxy, and Nginx workloads measure latency in addition to throughput. In terms of latency, the overhead for these workloads for BlackBox Enclaved versus native execution is less than 15% on both the Raspberry Pi and AMD server. Furthermore, Netperf measurements show that BlackBox provides fast networking performance as it involves no I/O virtualization, in contrast to using VMs. Applications are able to make full use of the host’s networking capabilities. Although applications are expected to encrypt their network I/O to protect their data, we did not encrypt network connections for these measurements to avoid encryption costs obscuring BlackBox’s overhead.

Figure 5 quantifies the CPU utilization when running the application workloads, as a measure of computational overhead. Solid bars indicate results run on the Raspberry Pi and the overlaid outlined bars indicate results run on the AMD server. CPU utilization is generally lower on the AMD server than the Raspberry Pi, since the AMD server is more powerful with more CPUs. On the Raspberry Pi, the difference in CPU utilization between BlackBox Enclaved and native execution is less than 15% across all workloads, and less than 5% for all workloads except Apache and Memcached. On the AMD server, the difference in CPU utilization between BlackBox
Enclaved and native execution is less than 15% across all workloads, except Apache. Apache CPU utilization for BlackBox Enclaved is high because at higher throughput rates, the cost of extra copying to use syscall buffers, as discussed in Section 3.4, becomes dominant. The buffers are used to send data from a container's PPAS to the OS to perform network I/O. Other than Apache, the difference in CPU utilization between BlackBox Enclaved and native execution is quite modest across both Arm embedded and server hardware.

5.2 System Call Coverage

We evaluated the completeness of Linux system call support in the current BlackBox prototype implementation by running the Linux Test Project (LTP) [44] version 20210524 system call test suite. LTP consists of 1344 test cases designed to test for correct functionality across the entire Linux system call interface. We compared system call support results for running LTP in an enclaved container on BlackBox versus running it natively, in both cases using the Raspberry Pi. When running LTP natively, 1149 test cases pass and 195 fail. These failures are expected and are a combination of missing dependencies and unsupported features of the kernel and architecture used. For example, test cases for the 16-bit version of fchown are not supported on the platform. When running LTP using BlackBox, 1012 test cases pass and 332 fail, demonstrating support for almost 90% of test cases that passed when run natively. The additional 137 failed tests are due to the current prototype not yet supporting lesser used system calls like process_vm_readv.

5.3 Evaluation of Practical Attacks

We evaluated BlackBox's effectiveness against a compromised OS by analyzing CVEs related to the Linux kernel and various Linux container engines such as Docker. We considered 23 CVEs which could result in privilege escalation, code execution, and memory corruption in Linux capable of compromising the integrity and confidentiality of container data; we did not consider denial of service attacks, as BlackBox does not guarantee availability. Specifically, privilege escalation occurs if the exploit enables the attacker to gain root access or kernel privilege level, and code execution occurs if the exploit enables executing arbitrary code at the same privilege as the software with the bug.

Table 4 lists the CVEs considered. We considered both malicious containers and unprivileged host users who exploit bugs in the kernel and container engines to elevate privileges and compromise container data. In general, these CVEs exploit flaws in container runtime systems and the kernel that enable an attacker to obtain kernel-level or root-level access. Ordinarily, this level of access compromises all container data and integrity on the system. Linux and the relevant container engine do not fully protect against any of these compromises. In contrast, BlackBox protects against all of them.

6 Related Work

Various approaches have been explored to securing applications from untrusted OSes. Hardware-based trusted execution environments (TEEs) such as ARM TrustZone [2] and Intel Software Guard Extensions (SGX) [30] can protect application memory from higher privileged software, but require applications to be written or rewritten specifically for this purpose and may impose other functionality restrictions. Some systems have built on TEEs. Haven [7] aims to en-
clave Windows applications by porting a Windows library OS to run inside SGX, avoiding Iago attacks by trusting the library OS at the cost of a significant TCB. Other systems also propose running library OSes enclaved by SGX [50, 59, 66]. CubicleOS [57] is a library OS designed to be runnably within containers that makes use of Intel MPK hardware extensions to isolate apps. Scone [3] uses SGX to enclave Linux containers, requiring its own custom threading model and a modified C library within SGX to provide system call support and shielded I/O interfaces for interacting with the OS. TZ-Container [28] leverages a shield layer and a container manager inside TrustZone to protect containers, but relies on the OS not modifying the memory mappings used to protect containers by scanning the OS image to ensure it does not contain instructions capable of updating page tables. TrustShadow [24] introduces a runtime system within TrustZone so that a limited number of security-critical legacy apps operate on TrustZone memory isolated from the OS. Unlike these approaches, BlackBox does not rely on TrustZone or SGX and does not rely on a library OS or other significant runtime system running inside an enclaved execution environment, avoiding increasing TCB complexity. Unlike Haven, its small TCB comes with potentially greater susceptibility to Iago attacks by allowing applications to use the system call interface of the untrusted OS.

Commodity hypervisors have been modified to secure applications from an untrusted OS by restricting a guest OS in a VM to an encrypted view of application memory [4, 10, 11, 27, 35, 45, 67]. For example, InkTag [27] uses two NPTs as part of its isolation mechanism, one for the OS and the other for all applications, separating the plaintext memory of isolated applications from encrypted memory, but relying on paravirtualized page table updates to isolate applications from each other. Appshield [12] uses virtualization techniques to protect and isolate critical applications against OS-level malware attacks. Appshield’s memory protection model requirements are not compatible with Linux’s copy-on-write semantics and its limited system call interface is insufficient to support significant workloads. In contrast, BlackBox does not rely on a hypervisor or traditional memory virtualization, but instead introduces a new concept of protected physical address spaces implemented as part of a container security monitor, enabling it to have a much smaller TCB.

Various approaches reduce the hypervisor’s TCB. Microhypervisors [25, 34, 61] build new hypervisors from scratch with smaller TCBS, but at the cost of a significantly reduced feature set. BlackBox’s approach allows for a small TCB while still maintaining a significant feature set and the full hardware support available in a commodity OS. SeKVM [38–40, 63] retrofits KVM with a small verified TCB to provide VM data confidentiality and integrity. In contrast, BlackBox provides container-level isolation and does not require a hypervisor, introducing a new concept, the CSM, that avoids the cost and complexity of hypervisor-based virtualization.

X-Containers [60] targets securely isolating containers in the cloud. Its containers include an entire library OS based on Linux and run on top of a Xen hypervisor, providing a model more akin to nested virtualization. Unlike BlackBox, X-Containers have a large TCB from requiring both large library OSes and a commodity hypervisor.

Other approaches have looked at ways to harden traditional containers. gVisor [23] runs a limited userspace kernel within a container and beneath applications. System calls are intercepted to further isolate applications from the host OS through reduced interactions and potential attack surfaces. gVisor’s increased isolation comes at the cost of a increased TCB size in the container. Distrose images [22] aim to limit the contents of a container to precisely what is necessary for the target app to run, reducing what must be trusted and maintained within a container. Linux Container Hardening [42] aims to improve the security of Linux containers through improving the kernel subsystems and primitives used by containers to be more secure. These approaches are complementary to BlackBox, and although they improve container security, unlike BlackBox, they all must still trust the OS and its large codebase.

### 7 Conclusions

BlackBox is a new container architecture providing fine-grain protection of application data confidentiality and integrity without trusting the OS. BlackBox achieves this by introducing a container security monitor, a new software component that creates protected physical address spaces for containers. The monitor enforces protected address spaces to isolate container memory and CPU state from the OS and other containers. It facilitates the use of OS facilities via system calls by passing required data between protected address spaces and the OS, implicitly declassifying such data. This narrow purpose keeps it small and simple.

Unlike a hypervisor, the monitor performs no virtualization or resource management. Instead, it relies on the OS to provide complex functionality required to manage hardware resources, including CPU scheduling, memory management, file systems, and device management. We have implemented BlackBox by repurposing Arm hardware virtualization support. Our results demonstrate that BlackBox supports existing unmodified containerized application workloads with modest overhead while maintaining a trusted computing base orders of magnitude less than an OS or commodity hypervisor.

### 8 Acknowledgments

Shih-Wei Li and Xuheng Li helped with system implementation. Andrew Baumann, Christoffer Dall, Peter Pietzuch, and Nicolas Viennot provided helpful comments on earlier drafts. This work was supported in part by OPPO, a Guggenheim Fellowship, DARPA contract N66001-21-C-4018, and NSF grants CCF-1918400, CNS-2052947, and CCF-2124080.
References


[60] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-Containers: Breaking Down Barriers to Improve Performance and Isolation


