CAP-VMs: Capability-Based Isolation and Sharing for Microservices

Vasily A. Sartakov, Lluís Vilanova, David Eyers†, Takahiro Shinagawa‡, Peter Pietzuch
Imperial College London, †University of Otago, ‡The University of Tokyo
{v.sartakov,vilanova,prp}@imperial.ac.uk, †dme@cs.otago.ac.nz, ‡shina@ecc.u-tokyo.ac.jp

Abstract

Cloud stacks must isolate microservices, while permitting efficient data sharing between isolated services deployed on the same physical host. Traditionally, the MMU enforces isolation and permits sharing at a page granularity. MMU approaches, however, lead to cloud stacks with large TCBs in kernel space, and the page granularity requires inefficient OS interfaces for data sharing. Forthcoming CPUs with hardware support for memory capabilities offer new opportunities to implement isolation and sharing at a finer granularity.

We describe cVMs, a new VM-like abstraction that uses memory capabilities to isolate application components while supporting efficient data sharing, all without mandating application code to be capability-aware. cVMs share a single virtual address space safely, each having only capabilities to access its own memory. A cVM may include a library OS, minimizing its dependency on the cloud environment. cVMs efficiently exchange data through two capability-based primitives assisted by a small trusted monitor: (i) an asynchronous read/write interface to buffers shared between cVMs; and (ii) a call interface to transfer control between cVMs. Using these two primitives, we build more expressive mechanisms for efficient cross-cVM communication. Our prototype implementation using CHERI RISC-V capabilities shows that cVMs isolate microservices (Redis and Python) with low overhead while improving data sharing.

1 Introduction

Cloud environments face a need for application compartmentalization. Today isolation between application components is enforced using virtual machines (VMs) [9, 32, 63] and containers [17, 40], either separately or in combination. Yet, microservice architectures push the limits of these mechanisms in terms of performance and security: microservices rely heavily on communication with each other, which means that multiple microservices are often co-located to minimize data sharing overheads. Current isolation approaches such as VMs and containers add substantial overheads when co-located microservices exchange data, and may rely on large trusted computing bases (TCBs).

VMs provide strong isolation through a relatively narrow hardware interface. Since a guest VM has its own OS kernel, its TCB can be reduced to a relatively small hypervisor, which multiplexes VM access to the hardware [57]. Efficient inter-VM data sharing, however, is challenging to achieve due to performance and page granularity trade-offs [16, 69].

In contrast, containers isolate processes into groups [17] and provide faster inter-process communication (IPC) primitives, including pipes, shared memory and sockets. Similar to VMs, containers face the same problems of a page-level sharing granularity and overheads due to frequent user/kernel transitions. Furthermore, their richer IPC primitives for data sharing come at the cost of a larger TCB: a shared OS kernel implements both namespace isolation between process groups and complex IPC primitives, increasing the likelihood of security vulnerabilities.

Existing cloud stacks thus face a fundamental tension when microservices are compartmentalized but must communicate: microservices must either copy data or rely on page table modifications, both of which are expensive operations that involve a privileged intermediary, such as the hypervisor or OS kernel, and lead to coarse-grained interfaces designed around page granularity.

In this work, we explore a different route to designing a cloud stack that isolates application components such as microservices, while supporting efficient sharing: if the hardware supported dynamic, low-overhead sharing of arbitrary-sized memory regions between otherwise isolated regions, how would this impact the cloud stack design?

We describe CAP-VMs (cVMs), a new VM-like abstraction for executing isolated components and sharing data across them. cVMs are enforced by a small TCB that uses hardware support for CHERI memory capabilities [68] to isolate and share data between compartments efficiently. Through the use of hybrid capabilities [65], cVMs avoid the need to port microservices to use capability instructions, circumventing compatibility problems that typically plague memory capabilities.
Memory capabilities [23, 68] impose flexible bounds on all memory accesses, which allows software components to be isolated without page table modifications or adherence to page boundaries. This offers a new opportunity to design memory sharing primitives between isolated compartments with zero-copy semantics.

Using memory capabilities as part of a cloud stack for microservices, however, raises new challenges: the cloud stack must (i) support existing capability-unaware microservices without cumbersome code changes, bespoke compiler support, or manual management of capabilities across isolation boundaries; (ii) remain compatible with existing OS abstractions, e.g., POSIX interfaces, all while keeping the TCB small; and (iii) offer efficient IPC-like primitives for otherwise untrusted components to share data safely and take advantage of the potential zero-copy sharing enabled by capabilities.

To address the above challenges, cVMs make the following design decisions:

(1) **Strong isolation through capabilities.** Multiple cVMs share a single virtual address space safely through capabilities. Each cVM is sandboxed by a pair of default capabilities, which confine the accesses of all instructions inside a cVM to its own memory boundaries. To avoid having to port existing microservices to a capability architecture, cVMs allow them to execute unmodified by using CHERI’s hybrid capability architecture [65], which integrates capabilities with a conventional MMU architecture. In addition, cVMs strictly limit how CHERI capabilities can be used to avoid known capability revocation overheads: cVMs are not permitted to store or export capabilities, and the transitions of communication capabilities is controlled by a trusted component.

(2) **Bespoke OS support through a library OS.** cVMs are self-contained with a small TCB, reducing reliance on the external cloud stack, while providing POSIX compatibility. They include a bespoke library OS with POSIX interfaces for, e.g., filesystem and network operations with cryptography for transparent protection, which is protected from application code using capabilities. In the library OS, each cVM thus implements its own namespace for filesystem objects, virtual devices, cryptographic I/O keys etc. Only low-level resources, e.g., execution contexts for threads and I/O device operations, are shared and provided by an external host OS kernel.

(3) **Efficient data sharing primitives.** cVMs offer two low-level primitives to share data efficiently without exposing application code to capabilities, which are hidden behind a small, trusted monitor Intravisor: (i) a CP_File API allows application code to share arbitrary buffers through an asynchronous read/write interface. Under the hood, the cVM implementation uses capability-aware instructions to exchange the rights to safely access each other’s memory, and read/write data at byte granularity at the cost of a single memory copy (whereas traditional file-oriented IPC would require two copies); and (ii) a CP_Call API transfers control between cVMs, which, e.g., can be used to implement synchronization mechanisms. By combining these two primitives, higher-level APIs are possible: (iii) a CP_Stream API supports efficient stream-oriented data exchange between cVMs, again introducing a single memory copy whereas traditional IPC would require two.

We implement cVMs on the CHERI RISC-V64 architecture, executable on FPGA hardware with CHERI support and multi-core RISC-V hardware. Our evaluation shows that cVMs provide a practical isolation abstraction with efficient data sharing: using the CP_Stream API for inter-cVM communication reduces latency for Redis by up to 54% compared to classical sockets-based interfaces, and reduces its standard deviation by up to 2.1×. When isolating a cryptography component of a Python-based service, cVMs introduce an overhead of up to 12% compared to the monolithic baseline.

## 2 Hardware Isolation Support

Next we survey the design space for isolation and sharing in cloud environments in more detail (§2.1), provide background on capability support on modern hardware (§2.2), and describe our threat model (§2.3).

### 2.1 Isolation and sharing in the cloud

In this paper, we argue that VMs and containers are two extremes of component isolation. VMs virtualize hardware interfaces such as page tables, instructions, traps, and physical device interfaces to manage both isolation and communication; containers virtualize pure software interfaces such as processes, files, and sockets for the same purposes.

**Compatibility.** Both VMs and containers are highly compatible with existing applications, which is critical for adoption in cloud environments. VMs can execute an unmodified guest OS on top of the host hypervisor, making virtualization transparent to applications inside VMs. Conversely, containers execute unmodified applications on top of the same host OS kernel that manages other containerized and non-containerized applications. In both cases, OS interfaces and semantics used by the virtualized applications remain unmodified compared to a non-virtualized environment.

But the compatibility offered by these technologies lowers communication performance, which is often exacerbated as we try to achieve better isolation between components.

**Isolation.** Despite strict isolation between the memory of containers, there is a lack of isolation of the TCB that manages the virtualization mechanism itself. Conventional container platforms, e.g., Linux containers [17], share privileged state, as they employ namespace virtualization: the OS kernel creates separate process identifiers, devices, filesystem views etc., which offer the illusion that a process group exists in isolation. In reality, containers share kernel data structures, and privilege escalation inside one container may lead to the compromise of all containers [1, 4]. In comparison, VMs are virtualized
through narrower interfaces, resulting in a conceptually simpler hypervisor that is harder to compromise [14, 57]. Unfortunately, stronger isolation comes at a performance price from both known hardware inefficiencies [13, 41, 61] as well as less flexible mechanisms for data sharing.

Sharing. Cloud services typically use networking as a means of communication. Even if microservices are co-located on the same host, they use a reliable network transport protocol, e.g., TCP. While this helps with scalability and elasticity, it adds a high overhead for co-located microservices, making optimizations based on direct memory sharing attractive. Nevertheless, both VMs and containers use page-based memory isolation, which limits the granularity and performance of memory sharing: mechanisms must be aware of page boundaries to avoid leaking sensitive data, and page table modifications for on-demand sharing are known to be expensive [62].

Co-location of microservices brings two avenues for performance improvements: (1) sharing can transparently accelerate communication of co-located microservices [45, 48]; and (2) new communication interfaces tailored towards sharing between co-located microservices can improve efficiency.

In microservices architectures with many services [26], traditional network-based communication shows its performance limits between tightly-coupled components [33]. Therefore, we argue that co-location of tightly-coupled microservices is preferable, and motivates us to design a cloud stack with novel, efficient isolation and communication interfaces and mechanisms for co-located microservices. This requires, however, new hardware support for isolation and sharing that is free of the “MMU tax” of page-level privileged memory protection.

2.2 CHERI capability architecture

Memory capabilities [18] are a different protection and sharing mechanism supported by the hardware. For example, the CHERI architecture [64, 68] implements capabilities as an alternative to traditional memory pointers. Each capability can be stored in memory or registers, and encodes a memory address range and permissions to operate in it, e.g., a read-only buffer, or a function to call.

CHERI protects capabilities by enforcing three properties: (1) provenance validity ensures that a capability can only be “derived”, i.e., constructed, from another valid capability, i.e., we cannot cast a sequence of bytes to a capability; (2) capability integrity means that capabilities stored in memory cannot be modified. CHERI achieves capability integrity through transparent memory tagging [68]; and (3) capability monotonicity requires that, when a capability is stored in a register, it is only possible to reduce its bounds and permissions, e.g., a read-only capability cannot be turned into a read-write one.

Building capability-based compartments. CHERI capabilities can be used to compartmentalize software components, e.g., plugins or libraries in a program, by giving each different capabilities to separate memory regions. The above properties enforced by CHERI ensure that compartments can coexist in the same address space, and will be isolated as long as their initial set of capabilities points to disjoint data and code in memory. The main application can, of course, grant each compartment extra capabilities, e.g., to allow particular cross-compartment memory accesses or function calls.

Pure- and hybrid-cap code. CHERI distinguishes two execution modes [65]: (i) in the pure-cap mode, all pointers must be capabilities (CHERI has separate registers for regular data and capabilities), and code must use a new set of capability-aware instructions; and (ii) in the hybrid-cap mode, code can mix ordinary and capability-aware instructions, which allows the coexistence of capability-unaware and pure-cap code via wrapping functions. This facilitates the incremental adoption of capabilities in software.

When accessing memory, pure-cap code must use new instructions that use capability registers instead of regular registers. CHERI also provides a CInvoke instruction to securely call functions using a pair of capabilities: the target function address, and an arbitrary value that is only meaningful to the callee function (e.g., an identifier for an object managed by the callee). A callee thus first “seals” the two capabilities using a CSeal instruction, passes them to any relevant callers, and unseals them when correctly called via CInvoke.

In contrast, hybrid-cap code relies on two new capability registers, the default data capability (ddc) and the program counter capability (pcc), which are used implicitly by capability-unaware instructions. The host OS starts all processes by setting their ddc and pcc to the entire virtual address space. Capability-aware code can then create new capabilities from these two registers, preserving CHERI’s provenance, integrity and monotonicity properties.

Pure-cap code thus introduces substantial compatibility challenges. For example, the three CHERI properties break typical memory allocators, preventing them from placing metadata before allocation, and capability alignment rules impose allocation and data layout restrictions. CHERI advocates for a trusted, system-wide garbage collector for memory deallocation and capability revocation [65]. This would be a disruptive change in cloud environments, potentially resulting in resource reclamation delays and increases in tail latencies.

Removing the need to use capability-aware code is important in cloud environments with limited control over tenant code. Therefore, we want to explore a design for a cloud stack that compartmentalizes microservices in CHERI’s hybrid-cap mode, without the disadvantages of capability-aware code.

2.3 Threat model

Cloud environments support multiple, isolated microservices, and thus we consider attacks in which an attacker controls a malicious service that interferes with another service by probing interfaces or trying to escape its sandbox. We assume that the attacker has full control over the microservice, its application components and library OS, e.g., by exploit-
ving vulnerabilities inside the compartment or by executing arbitrary code that includes capability-aware instructions.

Our TCB includes the underlying host OS kernel on the cloud server, but the entire application stack (program, libraries and library OS) is considered untrusted. We assume that the CHERI hardware is correct. We do not analyse side-channel attacks against CHERI, which is an important, yet orthogonal, consideration that affects both architectural and micro-architectural levels [66].

3 cVM Design

cVMs are a new virtualisation and compartmentalisation abstraction for cloud application components, such as microservices. Such components can often be co-located and exchange data, and cVMs add them with support for low-overhead data exchange using CHERI capabilities. The design of cVMs has the following features:

Separate namespaces. Unlike containers, cVMs do rely on a shared OS kernel for namespace isolation. They use capabilities to add a new userspace-level isolation boundary, moving OS kernel functionality from a privileged to an unprivileged layer. cVMs only use the host OS kernel for execution contexts, synchronisation, and I/O, bringing them closer to VMs.

Bypassed communication. cVMs are mutually untrusted, but communication bypasses the host OS kernel for performance. They use capabilities for on-demand access to memory regions used for communication, without compromising neighbouring memory.

Low-overhead isolation. cVMs use capabilities for low-overhead isolation of both process and program modules. For example, cVMs can isolate shared libraries with minimal changes to the calling interface.

Compatibility. cVMs use CHERI’s hybrid-cap mode. Capabilities are thus hidden from application code, which only needs changes to use new communication APIs.

3.1 Architecture overview

Fig. 1 shows the architecture of cVMs. Each cVM is an application component, such as a process or library, and has three parts: (i) program binaries and their libraries; (ii) a standard C library; and (iii) a library OS.

cVMs add two new isolation boundaries, enforced through capabilities. The Intravisor boundary separates the Intravisor from all cVMs, and cVMs from each other. The Intravisor is responsible for the lifecycle and isolation of cVMs, allows safe communication between them, and provides other primitives that cannot be implemented inside the unprivileged library OS (e.g., storage and networking I/O, time, threading and synchronisation). It has access to the memory of all cVMs, but not the other way around.

The Program boundary separates programs from the library OS that provides them the namespace for all OS primitives. A single library OS instance can thus host multiple, mutually-isolated programs with their own code and data (left-most cVM in Fig. 1).

These isolation boundaries are enforced by CHERI capabilities; compartmentalised content cannot access memory beyond its boundary, except through the controlled interfaces described next. Finally, there is a classical separation from the host OS, using CPU rings and MMU-based isolation.

3.2 Isolation boundaries

We now describe how cVM are isolated in more detail (see Fig. 2). Each program compartment contains the code and data of its binary, its dependencies (shared libraries), and the standard C library; the cVM also contains the library OS, which provides the OS functionality.

Isolation boundaries are enforced by giving each its own default CHERI capabilities using the pcc and dcc registers (see §2.2) with non-overlapping address ranges; compartmentalised code thus cannot load, store or jump into memory outside that granted by the capabilities that it holds. To allow program → libOS and libOS → Intravisor calls, cVMs have four handlers, synchronisation). It has access to the memory of all cVMs, but not the other way around.

The Program boundary separates programs from the library OS that provides them the namespace for all OS primitives. A single library OS instance can thus host multiple, mutually-isolated programs with their own code and data (left-most cVM in Fig. 1).

These isolation boundaries are enforced by CHERI capabilities; compartmentalised content cannot access memory beyond its boundary, except through the controlled interfaces described next. Finally, there is a classical separation from the host OS, using CPU rings and MMU-based isolation.

3.2 Isolation boundaries

We now describe how cVM are isolated in more detail (see Fig. 2). Each program compartment contains the code and data of its binary, its dependencies (shared libraries), and the standard C library; the cVM also contains the library OS, which provides the OS functionality.

Isolation boundaries are enforced by giving each its own default CHERI capabilities using the pcc and dcc registers (see §2.2) with non-overlapping address ranges; compartmentalised code thus cannot load, store or jump into memory outside that granted by the capabilities that it holds. To allow program → libOS and libOS → Intravisor calls, cVMs have four handlers, synchronisation). It has access to the memory of all cVMs, but not the other way around.

The Program boundary separates programs from the library OS that provides them the namespace for all OS primitives. A single library OS instance can thus host multiple, mutually-isolated programs with their own code and data (left-most cVM in Fig. 1).

These isolation boundaries are enforced by CHERI capabilities; compartmentalised content cannot access memory beyond its boundary, except through the controlled interfaces described next. Finally, there is a classical separation from the host OS, using CPU rings and MMU-based isolation.
uses these capabilities to invoke system calls on the library OS through the CInvoke instruction, while the rest of the application remains capability-unaware. The library OS has full access to the programs that it manages.

cVMs also need to implement the equivalent of guest/host (or VM/hypervisor) separation using CHERI capabilities in userspace. When creating a cVM, the Intravisor installs capabilities to its own host system call handlers on the new library OS instance; in turn, the library OS uses CInvoke to invoke Intravisor operations.

### 3.3 Creation and communication API

cVMs combine compatibility and flexibility when isolating cloud services. They support the execution of complete microservices using a process isolation abstraction, but also that of individual library components.

Tab. 1 shows the cVM API. New cVMs are created by cp_cvm_make(); similar to fork/exec, it accepts a disk image file, a program binary to load into the cVM, and a function in that binary to launch. If a cVM isolates a standalone library, cp_call() invokes functions in the library.

cVMs use CHERI capabilities for efficient inter-cVM communication. The Intravisor exchanges an initial set of capabilities between cVMs to allow communication.

**CP_File**. This primitive introduces a file-like API to access memory from another cVM at arbitrary granularity; the use of capabilities in CP_File permits bypassed access to memory without repeated mediation by the Intravisor.

A donor cVM registers a memory region with the Intravisor to share with other cVMs via cp_file_make(); a recipient cVM calls cp_file_get with the same key to obtain access. The cVMs then access data in the memory region via cp_file_read/write(). Internally, the library OS uses capability-aware code to copy data directly between the cVMs (using capcy; see §4).

To support asynchronous data transfers, cp_file_wait() and cp_file_notify() allow callers to wait for and notify events on a CP_File, respectively. Finally, the donor cVM calls cp_file_destroy() to destroy it, revoking all access.

**CP_Call**. This primitive invokes functions outside the calling cVM, e.g., a callback function in the library OS, or a function in a shared library. cVMs manage CP_Calls as follows: cp_call_make() registers a function in the donor that recipients can look up using cp_call_get() and then call with cp_call(). The call is received by the Intravisor, which creates a new thread in the donor’s cVM, sets it to execution to the target function with given arguments and, optionally, waits for its completion (based on the async argument).

**CP_Stream**. By composing the CP_Files and CP_Calls APIs, it is possible to construct more complex communication mechanisms. For example, we have built a stream-oriented API for inter-cVM communication in which the sender does not need to know where data is copied.

A recipient cVM calls cp_stream_recv() to register buffers for incoming messages (internally, a list of CP_Files); a sender cVM calls cp_stream_send() to copy data into any of the buffers available in the recipient. The recipient is then informed of data transfers when calling cp_stream_poll().

### 3.4 Capability management

The use of CHERI capabilities introduces two problems that cVMs must avoid: avoiding the need for application code to become capability-aware and performance problems when revoking capabilities.

As explained in §2.2, making application fully capability-aware requires non-trivial code changes. The design of cVMs avoids this by limiting the use of capability-aware code to a small portion of the standard C library, the library OS and the Intravisor, which explicitly handle the CP_Files and CP_Calls abstractions through syscall trampolines.

In the cVM design, we want to avoid a centralized trusted mechanisms for capability revocation (see §2.2), as this goes against our goal of minimizing overheads and TCB size. Therefore, only the Intravisor is permitted to store CHERI capabilities in memory: all capabilities that are passed by the Intravisor to cVMs have the CAP_STORE permission withheld.
Instead of having to perform expensive garbage collection, revocation can now be done by clearing a small number of capability registers. This can be done efficiently when programs call the cVM API to avoid interrupting execution.

4 Implementation

Next we report implementation details of cVMs on the CHERI RISC-V64 platform. Our implementation consists of 5,200 lines of C code and 100 lines of assembly for the Intravisor, and 1,800 lines of C code and 200 lines of assembly for the Init service, the Hostcall interface and CAP Devices. It uses the Linux Kernel Library (LKL) v4.17.0 [36] as the library OS and the musl standard C library v1.2.1 [42]. As the host OS kernel, we use CheriBSD [25].

4.1 cVM lifecycle

Initialisation. The boot process of a cVM is trigged by the Intravisor. It receives a deployment configuration for the cVM, which includes the heap size, the disk image location, the permitted interfaces, etc. It also defines the version and location of an Init service (see below) and the library OS binaries. The Intravisor first allocates memory for the cVM binary, stack and heap. It also allocates memory for the thread stack pool. Our implementation of cVMs cannot change the size of heap and stack at runtime, but this is a minor limitation given the size is in terms of virtual memory, and is only committed to physical memory on demand. Just as Cloud providers prefer re-instantiating VMs over use of memory balloons, we expect large resource size changes would re-instantiate cVMs.

All threads must be created inside a compartment’s memory, thus the Intravisor pres-allocates memory for future thread stacks. After that, the Intravisor deploys the image of the Init service into the cVM and spawns the initial thread in the context of the cVM. This thread prepares the hostcall callback tables, and enters the cVM via the CInvoke-based interface created by the Intravisor.

The Init service (Fig. 2) is responsible for initializing all components at deployment, and creates the communication interface between the library OS and the host system. It is part of the library OS isolation layer, which means that it can access the memory of the application component. It initialises the library OS, builds the syscall interface for the program (or library), deploys its binary and calls the entry function (e.g., c_start()). For an executable binary, it launches the program; for a library, the entry function initializes a CP_Stream and registers the public library functions with the Intravisor.

Execution. cVMs use the Linux kernel library (LKL) [36] as a library OS that provides a Linux-compatible environment. LKL processes system calls and requests the host OS kernel to perform actions as needed.

LKL’s storage and networking backends implement lean interfaces for hardware I/O devices: disk I/O has three hostcalls (disk_read/write(), disk_getsize()); networking uses only net_read/write(). The disk_read/write functions are applied to a file descriptor of the disk image; the network functions are invoked on a TAP device. The remaining functions in the hostcall interface are straightforward: they offer support for time and timer functions, debug output, threading and locking, and management of CAP Devices (see §4.3).

Threading. For simplicity, cVMs use a 1-to-1 threading model. When a cVM creates a thread, the pthread library requests an execution context from LKL, which in turn, requests a new thread from the host OS kernel. This requires the integration of the pthread implementations inside the cVM and the host—both must maintain their own thread-local stores, pointers to thread structs, etc.

When LKL requests a thread, it prepares a structure with an address of the entrance function, and a pointer to the arguments. This is passed to the host OS kernel, and the Intravisor creates a new thread with the provided arguments: it allocates a stack for the thread from the thread stack pool, pre-allocated at boot. After that, the new thread is ready to enter the cVM using CInvoke and capabilities are created by the hostcall interface. Prior to entering, the Intravisor switches the thread pointer tp register. Inside a cVM, threads have LKL TP values; when processing hostcalls, they have host ones.

4.2 Calls between nested compartments

cVMs use the CInvoke instruction to call functions between isolation layers, both (i) from an outer to an inner layer (ICALL), e.g., when the Intravisor invokes Init; and (ii) from an inner to an outer layer (OCALL), e.g., when performing a syscall or hostcall.

CInvoke takes two sealed capabilities (see §2.2) as arguments: (i) one with a new Program Counter Capability (pcc) value and another that points to a memory region that becomes accessible after the instruction execution. The pcc is replaced by the first unsealed capability; the second capability moves to the ctt6 (C31) register in the unsealed form.

Next we explain how CInvoke is used to implement both ICALLs and OCALLs:

ICALLs. Fig. 3 shows the switching mechanism for ICALLs. In this example, the Intravisor in the outer layer calls Init in the inner layer. To make the call, the caller prepares the first capability that points to the entry point inside the compart-
ment. This capability, together with the corresponding data
capability, defines the default capabilities of the inner compart-
ment. Inside the compartment, these capabilities, COMP.DDC
and ENTRY, PCC become ddc and pcc, respectively. While the
ENTRY.PCC capability can be passed as the first argument
of CInvoke, COMP.DDC must be loaded by the caller prior to
switching (see Fig. 3).

To return from the compartment or grant permission to
invoke functions in the outer layer from the inner layer, fur-
ther capabilities are needed: these are stored in memory by
the Intravisor before CInvoke is called, in a structure that
we call the Affix. They include a sealed ddc of the outer
layer (MON.DDC.sealed). Without this capability, the Intravi-
sor could not change ddc from the inner to the outer layer on
return in order to access the Intravisor’s data. This capability
can only be fetched from the inner layer—the accessible
memory is restricted by the ddc of the inner layer.

The Affix also includes RET.sealed and OCALL.sealed,
which are two sealed pcc capabilities to entry functions in
the outer layer. The former is used to return from the com-
partment; the latter points to an entry function, which is used
when the inner layer calls a function of the outer layer (e.g.,
printf()) and returns to continue execution inside the com-
partment. This is used for the syscall and hostcall interfaces.
Capabilities in the Affix are created by the Intravisor and
stored on the stack and inside per-compartment private stores.

OCALLs share many similarities with ICALLs. The caller
prepares a sealed capability of the return address. After the
end of a function, the callee uses CInvoke and the execu-
tion of the caller continues from the desired address. To-
gether with CInvoke, the callee passes the sealed capability
MON.DDC.seal1, which was passed originally inside the Affix.
It is put into ddc after the function returns.

4.3 Communication mechanisms

The data sharing API between cVMs from §3.3 is also based
on capabilities. Data referenced by capabilities, however, can
only be manipulated by capability-aware instructions, which
do not exist in hybrid-cap code. To resolve this issue, we me-
diate the interaction between hybrid-cap code and capabilities
using virtual devices called CAP Devices.

The CP_Files, CP_Calls, and CP_Streams primitives are
implemented using character devices, which are created by the
library OS and Intravisor. A program can read/write from/to
these devices, and the corresponding operations are performed
by capability-aware code inside drivers.

This design has two advantages: (i) despite its one mem-
ory copy, it is faster than traditional communication inter-
faces (see §6.5); and (ii) it offers a simple mechanism to
revoke capabilities. A remote cVM can inform the Intravisor
of the revocation, and the Intravisor requests the library OS
to destroy the corresponding CAP Device. To revoke capa-
bilities without device drivers in pure-cap code, a Intravisor
would have to stop the cVM execution and destroy capabilities
manually.

CP_Files support regular POSIX file operations. In contrast
to ordinary files, the content of CP_Files is not cached by the
page cache, and read/write operations can be unaligned.

Fig. 4a shows the implementation. A donor cVM advertises
one or more memory regions defined by keys, and a recipient
cVM probes the Intravisor for a given key. The Intravisor
verifies the access control list and builds a CAP Device for the
target CP_File (e.g., /dev/cf0). For the donor cVM to revoke
access, it uses its own CAP Device to request revocation,
and the Intravisor, together with the library OS, destroy the
CP_Files (cf0) driver along with its capabilities.

When the recipient cVM issues a cp_file_read() call,
the driver uses capcpy to copy data. For cp_file_read(), it
uses ld.cap to read data from a remote cVM and store it via
sd; a cp_file_write() does the reverse.

CP_Calls. To expose a function, a cVM creates an ICALL
entry and registers it with the Intravisor (see Fig. 4b). The
Intravisor maintains a table of exported functions for each
cVM, called cVM-RPCs. It consists of access control records
with capabilities, name identifiers and permissions. Application
components can interact with the cVM-RPCs via CAP
Devices, the capability management interface (/dev/cf0), and
the Intravisor.

Any function can be invoked by CP_Calls including ones
inside the library OS. This enables the use of CP_Calls as a
notification mechanism between CP_Files. The donor blocks
execution until the recipient cVM reads data. It makes the
wait() call with the driver, the driver puts the execution
thread in the work queue and waits for the signal. Prior to
blocking, it registers a wake-up CP_Call with the Intravisor.
The recipient cVM, in turn, finishes its operations with the
CP_Files, and notifies the donor via this CP_Call.

These basic operations can be composed to create higher-
level protocols, and a single CAP device can handle multiple
memory regions. For example, for Redis (see §6.3), we use
a series of read/write operations with a single notification as
well as batched reads with different capabilities.

**CPStreams.** In contrast to CPFiles, when sending data, the destination for CPStreams is unknown, and cp_stream_send() only knows the source. Therefore, one side of the communication pre-registers one or more destination buffers via cp_stream_recv(), and uses cp_stream_poll() to block. The remote side uses CP_Call to enter the remote compartment, atomically fetches one destination buffer from a pre-registered queue of buffers, and copies into this buffer data via capcyp. It then wakes up the poll queue and returns.

### 4.4 Capability revocation

Data transfers (capcyp) are performed by the drivers of CAP Devices without direct involvement of the Intravisor, which enhances performance and reduces the TCB. This, however, means that the driver must have access to the capabilities provided by the donor. We do not consider the driver trusted, thus it may be compromised by an adversary who obtains access to capabilities and memory outside the cVM after the end of a communication session. To mitigate against this threat, cVMs support a revocation mechanism. It guarantees that, once the donor cVM revokes capabilities, they are destroyed, and a recipient cVM cannot use them.

First, cVMs or communication capabilities are not created with the PERMIT_STORE_CAP permission. Code inside a cVM thus cannot store capabilities to memory: it can load them, modify, create new capabilities, but it fails on ST. The communication capabilities are stored once by the Intravisor, when the communication is established, and destroyed at the end. Second, the revoked capabilities in the CPU context are destroyed after a context switch by the host OS kernel.

### 5 Security Analysis

According to our threat model from §2.3, an attacker can gain control over a cVM. However, we guarantee that they cannot escape the compartment or access memory beyond its boundary due to the CHERI architectural properties (see §2.2): the ddc and pcc capabilities always apply, are non-extensible, and are controlled by the Intravisor.

Hybrid-cap code may be vulnerable to attacks that attempt to break execution flow. An adversary may inject capability-aware instructions (e.g., CLD/CSD, CInvoke) to access data and code outside of the compartment. To do this, the adversary requires capabilities, which they cannot construct from the available data inside a cVM.

To escape a compartment, an adversary must obtain appropriate capabilities. Each cVM, however, only maintains a few capabilities: a compartment (i) receives three sealed capabilities via Affixes, which can be inspected by an adversary but not unsealed to create new capabilities; and (ii) may receive capabilities used by CPFiles and CPStreams. These capabilities can be exploited by an adversary after gaining full control over the library OS. Since these are data capabilities, they cannot be used to create code capabilities, which are needed to escape the compartment. The adversary also cannot store these capabilities due to their permissions. Finally, they also cannot be exported outside of the compartment via the hostcall interface, because the interface does not handle capabilities and instead corrupts them.

Hybrid-cap code may contain security flaws, but an adversary cannot escape confinement, unless a flaw in the outer level provides them with unsealed capabilities. In our design, this is unlikely due to the Intravisor’s small TCB. The adversary cannot export or import capabilities via the hostcall interface or use them beyond a communication session. Vulnerable hybrid-cap code also cannot abuse system calls of the host OS kernel, escalate privileges or attack other cVMs, because the host OS kernel ignores all direct system calls from cVMs.

### 6 Evaluation

We now explore the performance of cVMs and the proposed communication interfaces. We begin with the overview of evaluation platforms and workloads (§6.1), then we compare the performance of microservices deployed with cVMs and Docker containers. In §6.3 we validate the efficiency of inter-cVM communication mechanisms, while in §6.4 we explore the overhead of the use of cVMs for components compartmentalisation. Finally, §6.5 concludes the section with a quantitative comparison of inter-cVM communication mechanisms with existing kernel mechanisms.

#### 6.1 Experimental environment

The CHERI architecture is under active development and, while ARM’s Morello board with CHERI support has been announced [8], it is unavailable at the time of writing. Therefore we use two evaluation platforms: (1) a single-core FPGA-based CHERI implementation [21]; and (2) a multi-core SiFive RISC-V implementation without CHERI support. **FPGA CHERI.** We synthesize an FPGA image from DARPA’s CHERI FETT program [22] (agf1-826d853003d6c433a), that ships with a single-core RISC-V64 CHERI system based on the FLUTE core (5-stage, in-order pipeline, running at 100 MHz) [50], and execute it on AWS F1 [7]. We use CheriBSD as the host OS kernel, compiled as a hybrid-cap system with LLVM version 11.0.0 and cheribuild [15].

The FPGA implementation enables a quantitative evaluation of cVMs, but introduces several limitations: (i) it has a single-core CPU with low clock frequency; (ii) its peripheral devices, in particular storage devices, are emulated by the host; and (iii) DRAM latency is disproportionately low compared to the CPU clock speed. As a consequence, we cannot realistically execute typical cloud workloads that are memory and I/O bound and use multiple cores, and we cannot eliminate system noise by pinning tasks to separate cores.

**SiFive RISC-V.** To avoid the abovementioned limitations, we also evaluate cVMs on a HiFive Unmatched RISC-V
board [30], which has 4 RISC-V64 (dual-issue, in-order) CPU cores running at 1.2 GHz. The CPU does not have CHERI support, and we instead emulate it by using CLang-13 in “sim” mode, which replaces all CHERI instructions with native RISC-V instructions. Our applications execute on Ubuntu v20.04 with Linux v5.11.0 and the RISC-V Docker port [49] with Alpine containers [6]. IPC micro-benchmarks execute in FreeBSD-14, since the FPGA uses Cheribsd and we run them on both platforms.

This approach allows us to realistically execute cloud applications; we execute CHERI-equivalent code and data paths while remaining fully compatible with existing RISC-V platforms, even though security is not enforced (e.g., with replacing capability load and store with ordinary ld and st instructions, CInvoke with jr, etc.).

**Application workloads.** We explore cVMs using several cloud applications and micro-benchmarks to evaluate their performance and isolation requirements:

**NGINX/Redis (§6.2).** This is a two-tier microservice deployment that evaluates the YCSB benchmark [70] using the NGINX [44] web server and the Redis [47] key/value store. NGINX acts as an API gateway and translates REST requests into Redis queries. When co-located, these services have a substantial amount of communication between them. We demonstrate that the cVMs interfaces, CP_Files and CP_Streams, significantly reduce overhead, using the SiFive platform to compare cVMs against a deployment using Docker containers [40].

**Redis (§6.3).** We execute a single-core Redis instance [47] and measure the latency of fixed-size GET and SET operations, comparing sockets and the equivalent cVM interface with CP_Streams. This experiment validates our previous results by also comparing the FPGA and SiFive environments.

**Python/Library (§6.4).** We measure the cost of using cVMs to isolate the components of a simple cryptographic application in Python, by deploying the Python runtime [59] and the PyCrypto cryptographic library [3] in mutually isolated cVMs that use the CP_Call and CP_File interfaces to communicate. This experiment runs on the FPGA environment.

### 6.2 Multi-tier deployment with NGINX/Redis

We first compare the benefits of using cVMs to co-locate communicating microservices, compared to a traditional deployment with Docker containers [40].

The computational limitations of our FPGA and SiFive platforms make it unfeasible to execute a complete microservices benchmark suite such as DeathStarBench [26]. We instead deploy a representative YCSB benchmark [70] (workloadb, 1 KB records, read/update ratio 95/5) in the SiFive platform with two-tiers, where the NGINX web server [44] acts as an API gateway that redirects incoming HTTP requests to the Redis key/value store [47], which acts as a cache for frequently used data. We use wrk2 [5] to generate requests to NGINX through a 1 GbE network, measuring the latency of various

---

**Fig. 5:** Control/data flow in multi-tier deployment (NGINX/Redis)

![Multi-tier deployment](fig5.png)

These microservices benefit from host co-location due to the frequent interaction between the (NGINX) API gateway and its (Redis) cache. Fig. 5 compares the Docker and cVM deployments. Docker incurs multiple data copies between the microservices and the TCP/IP network stacks—for example, Fig. 5a shows that Redis copies values into a send buffer that is passed to the TCP/IP stack, which NGINX then copies into an output buffer that is, in turn, passed to the TCP/IP stack for the client (for a total of 4 copies, including the kernel’s TCP/IP stack). In contrast, cVMs are able to reduce the number of copies—for example, Fig. 5b shows that the CP_Stream primitive reduces the total number of memory copies to 2, since Redis values are always directly copied into NGINX’s output buffer. To support this optimization, NGINX and Redis replace their sockets with CP_Streams, NGINX registers the output buffer with a CP_Stream, and the CP_Stream write in Redis uses capabilities to directly copy data into the output buffer (which NGINX can then send to the client).

Fig. 6 shows the median and 95th latency of the four YCSB queries under various throughput regimes, comparing the baseline Docker deployment with cVMs. We can see that cVMs is clearly more efficient; it has with lower latencies on all cases (e.g., 20-40% for median latency), and substantially higher throughput without shooting latency above 5 ms (e.g., 33-50% for median latency).

**Conclusion:** In a typical deployment with multiple microservices, cVMs can provide service isolation while bringing both lower latency and higher throughput, compared to traditional containers. This performance gain is driven by a reduced number of memory copies (via CP_Stream), using very fast calls to the capability-hiding TCB in cVMs (via CP_Call within CP_Streams). Furthermore, cVMs come with a smaller TCB...
compared to containers, and we would expect cVMs to outperform VMs because of their overheads in memory virtualization (especially in memory-bound applications) and in communication mechanisms (we will have unnecessary data copies by the guest OS and/or hypervisor, or cross-VM copies via PCIe with directly assigned devices).

### 6.3 Platform validation with Redis

Before looking at more use-cases, we now validate our results by comparing the FPGA and SiFive platforms. We use Redis with a single connection that measures the latency of 1000 GET or SET operations with fixed-size keys (1 B) and values (100 B), and a simple client application that is co-located with it. The baseline system uses separate processes and TCP/IP sockets, whereas we use separate cVMs for each application and CP_Stream for communication (similarly to §6.2).

Fig. 7 shows the latency distribution of the GET and SET requests on all configurations. These results validate our observations from the multi-tier YCSB benchmark in §6.2; cVMs show lower latencies with less deviation on both platforms, compared to a native system with TCP/IP sockets—90% of cVM requests take 14–19 ms, while the baseline takes 19–35 ms in the FPGA platform. The SiFive platform shows the same conclusions, albeit with different absolute numbers—this is because the FPGA runs at a much lower frequency, and two processes must be co-scheduled on the same core (with both the baseline and cVMs).

**Conclusion** The CP_Stream primitive in cVMs is reliably better on both the FPGA and SiFive platform, delivering lower communication latencies across the throughput spectrum. We can thus conclude that our end-to-end evaluation in §6.2 is reliably representing how cVMs would perform in a real-world CHERI-enabled CPU. Furthermore, we later re-validate this conclusion by comparing cVMs against various IPC primitives across all platforms in §6.5.

### 6.4 Process compartmentalisation with Python/Library

Next we explore the overhead of compartmentalising a shared library from a cryptographic application we wrote in Python. In this case, we harden the security of a hypothetical cloud application by mutually isolating the Python runtime and the native cryptographic module library, PyCryptodome [3]; by using separate cVMs, we can safeguard the application against malicious interference by package managers [43], or safeguard the library against unauthorized access to cryptographic keys [2].

In this case, Python creates CP_Files for the in/out buffers it needs to pass to the PyCryptodome library, uses CP_Call to transfer control into the library indicating the CP_Files used as input/output arguments (instead of the original version where we pass raw input/output buffer pointers); PyCryptodome then uses these CP_Files to read its input and encrypt/decrypt it into the output buffer (using AES-128) and, finally, uses CP_Call to return execution to the Python runtime.

Fig. 8 shows the mean encrypt/decrypt throughput of this application using various buffer sizes in the baseline (non-isolated) system and cVMs, using the FPGA platform. Note that the low absolute numbers and measurement variance (shown as shaded areas) are due to the platform limitations described in §6.1 (§6.3 already showed that the same trends hold in a more realistic platform where such limitations would not be present).

We clearly see that cVMs have a negligible performance
impact. Throughput grows very rapidly until its peak at 32 KB buffers, where the encryption and decryption rate of cVMs is only 7% and 12% lower than the baseline, respectively, and overheads become even smaller as buffer sizes grow larger (as expected).

Our analysis also shows that CP_Call and TCB calls into the cVMs Intravisor are reasonably efficient. For reference, the mean execution time for the AES cryptographic code with a 16 B buffer is comparable to the time for a C binding invocation in Python. At such sizes, CP_Call invocations account for half of the performance overhead (which sits at 97% and 101% for encryption and decryption, respectively), only slightly above a simple C binding invocation in Python, and go down to 7% with larger buffer sizes.

**Conclusion:** cVMs is very effective at hardening cloud applications by isolating some of their components, such as shared libraries. The required changes are minimal and do not change the semantics of the application interfaces, since the CP_File and CP_Call primitives follow well-understood memory copy and function call semantics, respectively (as a reminder, CP_Streams are built on top of these). Furthermore, the cost of this added isolation is very small at even small buffer sizes, and becomes negligible as the amount of work performed between cVMs-enabled operations increases.

### 6.5 Comparison between cVMs and other primitives

Finally, we compare cVMs to other existing IPC primitives in the baseline system, and re-validate our performance results across our two platforms (FPGA and SiFive). The baseline system uses two threads of a single process instead of cVMs themselves (the FPGA otherwise shows very low TLB performance), and we measure the performance of CP_Files and CP_Streams, pipes (PIPE), unix sockets (UNIX) and TCP/IP sockets (TCP). For comparison, we also measure a raw local memcpy (MEMCPY; 4 instructions, aligned data and double-word load/store operations) as our upper bound in performance, but do not evaluate CP_Calls since there is no direct equivalent in the baseline kernel.

Fig. 9 shows the results of all these experiments using various buffer sizes on both the FPGA and SiFive platforms. First of all, the peak performance of MEMCPY in the FPGA platform is severely limited and fluctuates because of the TLB size and simple indexing function of its Flute CPU (problems that carry onto the other primitives too).

The overhead of CP_Files is just 6% compared to MEMCPY on the FPGA platform and negligible for SiFive, while it significantly outperforms all baseline IPC mechanisms. This is because we do a simple cross-cVM memcpy using CHERI’s ld.cap and cincoffsetimm instructions to perform the memory access and to increment the capability offset, respectively. These results also show that domain transitions via CInvoke are very efficient, since every CP_File operation requires two capability calls and their returns (user→library OS→Intravisor, and then back).

All baseline IPC primitives have 2× overhead or more, since they perform more data copies than MEMCPY and CP_Files, which closely follow ideal performance. Interestingly, CP_Streams show low performance in the FPGA platform (despite its lower number of copies), whereas they show close-to-CP_File performance in SiFive. This is because CP_Streams offer an asynchronous communication primitive where two concurrent processes time-share a single CPU in the FPGA platform when using the cVM API (same reason why all IPC primitives have lower relative performance in the FPGA platform compared to SiFive). It is also worth noting that UNIX sockets are the closest to CP_Streams (both are bi-directional, support more than two parties and have sequenced packet modes), but exhibit only 10% and 54% of the

---

1. This represents 0.79 MB/s and 0.96 MB/s for the baseline, and 0.74 MB/s and 0.85 MB/s for cVMs.
performance of CP_Streams for 4 MB buffers on the FPGA and SiFive platforms, respectively.

**Conclusion:** When combining CHERI support with a realistic multi-core implementation (similar to what the Morello board is supposed to provide), we would expect the combined results to be close to those of the SiFive platform, with a minor performance decrease (similarly to the difference between memcpy and CP_Files in Fig. 9a). This potential performance degradation is significantly smaller than measured performance improvements, which range between 2-4 for the multi-core SiFive platform against the best baseline primitives, to 2× and an order of magnitude speedup for the single-core FPGA platform depending on the mechanism and buffer size.

### 7 Related Work

**Intra-process compartments.** Various projects apply intra-process isolation or introduce isolation primitives. CubeleOS [51] isolates components of a user-level library OS using Intel MPK, but unlike cVMs, it cannot readily and efficiently support legacy POSIX calls. Shreds [20], Janus [28], Erim [60], Hodor [29] and Donkey [54] also use page tag-based isolation (ARM Domains, Intel MPK or a custom RISC-V implementation) to implement protection domains and communication. In cases where tags can be directly operated by user code (such as Intel MPK with the wrpkrusu instruction), the system also needs a trusted toolchain or program verifier, which cVMs do not require. Furthermore, page tags limit the number of compartments and communication buffers, as well as their granularity, which is not a problem for cVMs as it uses CHERI capabilities.

Similarly, works such as NaCl [71] and WASM [27] fall onto similar problems, since they require obsolete Intel segmentation and/or proof-carrying code that must be verified by either a toolchain or loader. ConfLLVM [11] also uses MPK to isolate trusted/untrusted code inside a process, but only supports two domains with asymmetric data exchange: trusted code can only interact with untrusted code. cVMs do not limit the number of protection domains, and inter-cVM communication can be symmetric.

LwCs [37] are an OS abstraction of intra-process protection, but their granularity is a page, and switching domains comes at the cost of switching page tables. XFI [24] provides fine-grained memory protection and control flow integrity by extending software-based fault isolation, but software-based protection incurs runtime overhead and are error-prone due to their complexity.

**Compartmentalisation frameworks.** cVMs enable the deployment of isolated shared libraries. Prior work proposes frameworks for compartmentalisation: Wedge [10] identifies code parts that can be isolated; PrivTrans [12] is a source-code partitioning tool that separates trusted and non-trusted components; Glamdring [35] does the same for trusted execution. These approaches are orthogonal to cVMs, and they could be used to generate application components for cVMs.

**Trusted execution.** Intel SGX [31, 38, 39] provides enclaves as an intra-process isolation primitive. Enclaves are part of processes and cannot be accessed by privileged software or other enclaves. Frameworks such as Graphene-SGX [19], SGX-LKL [46], Panoply [56], and the Spons and Shield Framework [53] deploy programs inside enclaves together with a library OS. This design decreases the possible impact of the untrusted kernel on enclave software.

cVMs also use a library OS and share design features with these frameworks, but provide effective data sharing that cannot be implemented in enclaves. Enclaves can only share untrusted memory and cannot access each other, which is necessary for fast inter-cVM communication. Since enclaves do not trust the host, they must use encryption, which impacts performance [52]. Therefore, an interface similar to CP_Files cannot be implemented with enclaves.

**Library OSs** can be used to de-privilege OS kernel components or create user-level containers. µKontainer [58] is a container kernel based on the LKL library OS [36]. Williams et al. [67] show that library OSs can be efficiently executed on top of processes instead of bare VMs. X-Containers [55] offer a cloud platform using library OSs. cVMs share the idea of a user-level library OS-based containers but extend them with strong isolation and a secure and efficient communication mechanism using CHERI capabilities.

**Machine and process isolation.** We have discussed the performance and TCB size shortcomings of traditional process-based isolation when compared to cVMs, as well as the limitations of the pure-cap CHERI system. One could envision using virtualization and Intel’s vmfunc to strike a similar balance between shared TCB size and communication performance [34] but, unfortunately, the use of virtualization would introduce well-known I/O and memory translation overheads that are critical in a cloud stack, and not present in cVMs.

### 8 Conclusions

cVMs are a new VM-like abstraction for cloud applications that use memory capabilities for secure isolation. cVMs include library OSs to minimize how much of the cloud environment is within the TCB. Multiple cVMs safely share an address space, allowing more efficient interaction of application components than when crossing current VM/container boundaries. Their asynchronous read/write interface and synchronous call interface allow capability-unaware, legacy code to run within cVMs. These secure inter-cVM communication primitives are shown to be efficient in microbenchmarks, and boost the throughput of Redis.

**Acknowledgements**

This work was partially funded by the UK Government’s Industrial Strategy Challenge Fund (ISCF) under the Digital Security by Design (DSbD) Programme.
References

[1] CVE-2013-6441. Available from MITRE, CVE-ID CVE-2013-6441, December 2 2013.

[2] CVE-2014-016021284. Available from MITRE, CVE-ID CVE-2014-0160, December 2 2014.

[3] A self-contained cryptographic library for Python. https://github.com/Legrandin/pycryptodome, 2021. Last accessed: December 14, 2021.

[4] CVE-2021-21284. Available from MITRE, CVE-ID CVE-2021-21284, December 2 2021.

[5] A constant throughput, correct latency recording variant of wrk. https://github.com/giltene/wrk2, 2021. Last accessed: December 13, 2021.

[6] Alpine Linux. https://alpinelinux.org, 2021. Last accessed: December 13, 2021.

[7] Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/, 2021. Last accessed: December 14, 2021.

[8] Arm morello program. https://developer.arm.com/architectures/cpu-architecture/a-profile/morello, 2021. Last accessed: December 14, 2021.

[9] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP ’03, page 164–177, New York, NY, USA, 2003. Association for Computing Machinery.

[10] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08). USENIX Association, April 2008.

[11] Ajay Brahmakshatriya, Piyus Kedia, Derrick P. McKee, Deepak Garg, Akash Lal, Aseem Rastogi, Hamed Nemati, Anmol Panda, and Pratik Bhatu. Conflvm: A compiler for enforcing data confidentiality in low-level code. In Proceedings of the Fourteenth EuroSys Conference 2019, EuroSys ’19. ACM, 2019.

[12] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In 13th USENIX Security Symposium (USENIX Security 04), San Diego, CA, August 2004. USENIX Association.

[13] Jeffrey Buell, Daniel Hecht, Jin Heo, Kalyan Saladi, and R Taheri. Methodology for performance analysis of vmware vsphere under tier-1 applications. VMware Technical Journal, 2(1):19–28, 2013.

[14] Reto Buerki and Adrian-Ken Rueegsegger. Muen-an x86/64 separation kernel for high assurance. University of Applied Sciences Rapperswil (HSR), Tech. Rep, 2013.

[15] Building system for CHERI software. https://github.com/CTSRD-CHERI/cheribuild, 2021. Last accessed: December 14, 2021, commit a37f5cc.

[16] Anton Burtsev, Kiran Srinivasan, Prashanth Radhakrishnan, Kaladhar Voruganti, and Garth R. Goodson. Fido: Fast Inter-Virtual-Machine Communication for Enterprise Appliances. In 2009 USENIX Annual Technical Conference (USENIX ATC 09), San Diego, CA, June 2009. USENIX Association.

[17] Canonical. Linux containers. https://linuxcontainers.org, 2021. Last accessed: December 14, 2021.

[18] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. SIGPLAN Not., 29(11):319–327, nov 1994.

[19] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In 2017 USENIX Annual Technical Conference (USENIX ATC 17), pages 645–658, Santa Clara, CA, July 2017. USENIX Association.

[20] Yaohui Chen, Sebassujeen Reymonjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-Grained Execution Units with Private Memory. In 2016 IEEE Symposium on Security and Privacy (SP), pages 56–71, 2016.

[21] CHERI-modified versions of the Flute processor. https://github.com/CTSRD-CHERI/Flute, 2021. Last accessed: December 14, 2021.

[22] Darpa FETT Bug Bounty Program. https://fett.darpa.mil, 2021. Last accessed: December 14, 2021.

[23] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. SIGOPS Oper. Syst. Rev., 42(2):103–114, March 2008.

[24] Úlfar Erlingsson, Martín Abadi, Michael Vrabie, Mihai Budiu, and George Necula. Xfi: Software guards for system address spaces. pages 75–88, 01 2006.

[25] FreeBSD adapted for CHERI-MIPS, CHERI-RISC-V, and Arm Morello. https://github.com/CTSRD-CHERI/cheribsd, 2021. Last accessed: December 14, 2021.
[26] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. SIGPLAN Notices, 52(6):185–200, June 2017.

[28] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael Scott, Kai Shen, and Mike Marty. Janus: Intra-process isolation for high-throughput data plane libraries. Technical report, Technical Report UR CSD/1004, 2018.

[29] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In USENIX ATC, pages 489–504, Renton, WA, July 2019. USENIX Association.

[30] HiFive Unmatched. https://www.sifive.com/boards/hifive-unmatched, 2021. Last accessed: December 13, 2021.

[31] Simon P. Johnson. Scaling towards confidential computing. https://systex.ibr.cs.tu-bs.de/systex19/slides/systex19-keynote-simon.pdf, 2019.

[32] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In Proceedings of the Linux symposium, volume 1, pages 225–230, 2007.

[33] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making rpcs first-class datacenter citizens. In USENIX Annual Technical Conf. (ATC), July 2019.

[34] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In Proceedings of the Twelfth European Conference on Computer Systems, European Conference on Computer Systems (EuroSys), page 437–452, 2017.

[35] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O’Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzschke, David Eyers, Rüdiger Kapitza, et al. Glamdring: Automatic application partitioning for intel [SGX]. In 2017 {USENIX} Annual Technical Conference ({USENIX}){ATC} ’17, pages 285–298, 2017.

[36] Linux Kernel Library. https://github.com/lkl, 2021. Last accessed: December 14, 2021.

[37] James Litton, Anjo Vahldiek-Oberwagner, Eslam El-nikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-weight contexts: An OS abstraction for safety and performance. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pages 49–64. USENIX Association, November 2016.

[38] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel® Software Guard Extensions (Intel® SGX) Support for Dynamic Memory Management Inside an Enclave. In Proceedings of the Hardware and Architectural Support for Security and Privacy 2016, pages 1–9, 2016.

[39] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. HASP@ ISCA, 10, 2013.

[40] Dirk Merkel et al. Docker: lightweight linux containers for consistent development and deployment. Linux journal, 2014(239):2, 2014.

[41] Gal Motika and Shlomo Weiss. Virtio network paravirtualization driver: Implementation and performance of a de-facto standard. Computer Standards & Interfaces, 34(1):36–47, 2012.

[42] musl libc. https://musl.libc.org, 2021. Last accessed: December 14, 2021.

[43] new npm malware with Bladabindi trojan spotted. https://blog.sonatype.com/bladabindi-njrat-rat-in-jdb.js-npm-malware, 2021. Last accessed: December 14, 2021.

[44] nginx – HTTP and reverse proxy server. https://nginx.org, 2021. Last accessed: December 13, 2021.

[45] Fengfeng Ning, Chuliang Weng, and Yuan Luo. Virtualization i/o optimization based on shared memory. In IEEE Intl. Conference on Big Data, 2013.
[46] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. Sgx-lkl: Securing the host os interface for trusted execution. arXiv preprint arXiv:1908.11143, 2019.

[47] Redis is an in-memory database that persists on disk. https://github.com/redis/redis, 2021. Last accessed: December 14, 2021.

[48] Yi Ren, Ling Liu, Qi Zhang, Qingbo Wu, Jianbo Guan, Jinzhu Kong, Huadong Dai, and Lisong Shao. Shared-memory optimizations for inter-virtual-machine communication. ACM Computing Surveys, February 2016.

[49] RISC-V bring-up tracker. https://github.com/carlosedp/riscv-bringup, 2021. Last accessed: December 7, 2021.

[50] RISC-V CPU, simple 5-stage in-order pipeline. https://github.com/bluespec/Flute, 2021. Last accessed: December 14, 2021.

[51] Vasily Sartakov, Lluís Vilanova, and Peter Pietzuch. Cubicleos: A library os with software componentisation for practical isolation. In Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’21, pages 575–587. ACM, 2021.

[52] Vasily A. Sartakov, Stefan Brenner, Sonia Ben Mokhtar, Sara Bouchenak, Gaël Thomas, and Rüdiger Kapitza. EActors: Fast and Flexible Trusted Computing Using SGX. In Proceedings of the 19th International Middleware Conference, Middleware ’18, pages 187–200, New York, NY, USA, 2018. ACM.

[53] Vasily A. Sartakov, Daniel O’Keeffe, David Eyers, Lluís Vilanova, and Peter Pietzuch. Spons & shields: Practical isolation for trusted execution. In Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2021, page 186–200, New York, NY, USA, 2021. Association for Computing Machinery.

[54] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys – efficient in-process isolation for risc-v and x86. In 29th USENIX Security Symposium (USENIX Security 20), pages 1677–1694. USENIX Association, August 2020.

[55] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Baggasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 121–135, 2019.

[56] Shweta Shinde, DL Tien, Shrutí Tople, and Prateek Saxena. Panoply: Low-TCB Linux Applications With SGX Enclaves. In Proceedings of the Annual Network and Distributed System Security Symposium (NDSS), page 12, 2017.

[57] Udo Steinberg and Bernhard Kauer. Nova: A microhypervisor-based secure virtualization architecture. In Proceedings of the 5th European Conference on Computer Systems, EuroSys ’10, page 209–222, New York, NY, USA, 2010. Association for Computing Machinery.

[58] The Python programming language. https://github.com/python/cpython, 2021. Last accessed: December 13, 2021.

[59] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In 28th USENIX Security Symposium (USENIX Security 19), pages 1221–1238. USENIX Association, August 2019.

[60] Lluís Vilanova, Nadav Amit, and Yoav Etsion. Using smt to accelerate nested virtualization. In Proceedings of the 46th International Symposium on Computer Architecture, ISCA ’19, page 750–761, New York, NY, USA, 2019. Association for Computing Machinery.

[61] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. SIGOPS Oper. Syst. Rev., 36(SI):181–194, December 2003.
[64] Robert NM Watson, Peter G Neumann, Jonathan Woodruff, Michael Roe, Jonathan Anderson, David Chisnall, Brooks Davis, Alexandre Joannou, Ben Laurie, Simon W Moore, et al. Capability hardware enhanced risc instructions: Cheri instruction-set architecture (version 8). Technical report, University of Cambridge, Computer Laboratory, 2019.

[65] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In 2015 IEEE Symposium on Security and Privacy, pages 20–37. IEEE, 2015.

[66] Robert NM Watson, Jonathan Woodruff, Michael Roe, Simon W Moore, and Peter G Neumann. Capability hardware enhanced risc instructions (cheri): Notes on the meltdown and spectre attacks. Technical report, University of Cambridge, Computer Laboratory, 2018.

[67] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. Unikernels as processes. In Proceedings of the ACM Symposium on Cloud Computing, SoCC ’18, page 199–211, New York, NY, USA, 2018. Association for Computing Machinery.

[68] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), pages 457–468, 2014.

[69] Xen project. Event channel internals. https://wiki.xenproject.org/wiki/Event_Channel_Internals, 2021. Last accessed: December 14, 2021.

[70] Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB, 2021. Last accessed: December 13, 2021.

[71] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In Intl. Symp. on Security and Privacy (S&P), pages 79–93. IEEE, 2009.