2025 Design, Automation & Test in Europe Conference (DATE 2025)

Enabling Security on the Edge: A CHERI Compartmentalized Network Stack

Donato Ferraro*†, Andrea Bastoni†‡, Alexander Zuepke†‡, Andrea Marongiu*
University of Modena and Reggio Emilia*, Minerva Systems†, Technical University of Munich‡

***Abstract*—The widespread deployment of embedded systems in critical infrastructures, interconnected edge devices like autonomous drones, and smart industrial systems requires robust security measures. Compromised systems increase the risks of operational failures, data breaches, and—in safety-critical environments—potential physical harm to people. Despite these risks, current security measures are often insufficient to fully address the attack surfaces of embedded devices. CHERI provides strong security from the hardware level by enabling fine-grained compartmentalization and memory protection, which can reduce the attack surface and improve the reliability of such devices. In this work, we explore the potential of CHERI to compartmentalize one of the most critical and targeted components of interconnected systems: their network stack. Our case study examines the trade-offs of isolating applications, TCP/IP libraries, and network drivers on a CheriBSD system deployed on the Arm Morello platform. Our results suggest that CHERI has the potential to enhance security while maintaining performance in embedded-like environments.**

***Index Terms*—Security, Network, CHERI, Operating Systems**

I. Introduction

Nowadays, interconnected, powerful embedded devices are widely used "at the edge," such as in drones (UAVs), autonomous vehicles, smart industrial plants, and in many components of critical infrastructures. Although these devices are relatively powerful, compared to high-performance data center systems, edge devices are more resource-constrained in terms of processing power and memory. They also typically operate in critical environments with real-time requirements.

To improve performance, reduce size, and save space, many of these devices lack security features such as Memory Protection Units (MPUs) and Memory Management Units (MMUs). As a result, all applications typically run within a single address space. For example, the NuttX/PX4 framework [9], [10], widely used in both autonomous and remote-controlled drones, runs all software components—network stacks, drivers, and applications (e.g., actuation)—without isolation.

This lack of isolation increases the risk that a vulnerability in one component could compromise the entire system. For instance, a buffer overflow in the network stack could allow an attacker to take full control of a drone, potentially causing physical damage (e.g., making the drone crash) or leaking sensitive data, such as video surveillance footage. As embedded systems are increasingly used in sensitive and highly connected environments, the need to ensure their effective security is becoming more critical [21]. Unfortunately, even when MPUs and MMUs are available, their coarse-grained isolation can lead to performance degradation and reduced predictability [1], [2]. This issue also affects systems that use lightweight isolation among threads to minimize the overhead of task-switches.

Capability Hardware Enhanced RISC Instructions (CHERI) [14] represents an advancement in addressing the aforementioned security challenges. CHERI extends conventional CPU architectures with a *capability*-based security model, providing fine-grained memory protection at byte level, by associating each memory reference with a *capability*—a protected token that specifies bounds and access rights to it. This mechanism allows systems to compartmentalize software components efficiently and prevents them from accessing memory regions outside their designated bounds. Unlike traditional memory isolation techniques that rely on MPUs or MMUs, CHERI enforces security at the hardware level with potentially lower overhead, making it a promising candidate for embedded systems where resources are constrained, and enhanced memory protection is critical.

However, CHERI is still an emerging area of research, and aside from FPGA deployments, there is limited availability of CHERI-enabled hardware. The CHERI RISC-V Sonata development platform [12] was only unveiled in 2024, and the Arm Morello [11] platform is available on request only. The Arm Morello platform is currently the only Arm-based prototype supporting CHERI. Although it is more powerful than typical microcontrollers and small embedded systems, it provides a valuable real-world testbed for exploring the potential and the trade-offs of CHERI, even for embedded-like devices. For example, the platform has been used in the AutoCHERI project [16] to evaluate CHERI's feasibility in enhancing security for autonomous vehicles.

Given the central role of the network stack in interconnected edge devices, it is unsurprising that network components are major targets of security attacks [25]–[27], with memory-related issues being one of the main intrusion vectors. For instance, recent vulnerabilities like CVE-2023-52370 [5] and CVE-2023-6951 [6] exploit buffer overflows in the network stack, while CVE-2024-38951 [7] leverages unchecked buffer

A. Bastoni and A. Zuepke were supported by the Chair for Cyber-Physical Systems in Production Engineering at TUM and the Alexander von Humboldt Foundation. This work has also been supported by Secure Systems Research Center, Technology Innovation Institute (TII). The work was also supported by the European Union under the NextGenerationEU Programme within the Plan "PNRR - Missione 4 "Istruzione e Ricerca" - Componente C2 Investimento 1.1 "Fondo per il Programma Nazionale di Ricerca e Progetti di Rilevante Interesse Nazionale (PRIN)" by the Italian Ministry of University and Research (MUR)". Project Title: "Simplifying Predictable and energy-efficient Acceleration from Cloud to Edge (SPACE)", Project code: E53D23007800006. MUR D.D. financing decree n. 959, 30th June 2023.

978-3-9826741-0-0/DATE25/© 2025 EDAA

limits to mount a Denial-of-Service (DoS) attack on the MAVLink [39] protocol of PX4. Compartmentalizing the elements of the network stack can prevent vulnerabilities in one component from compromising the entire system.

The network stack is typically composed by (i) an Ethernet driver, responsible for managing the physical Network Interface Card (NIC); (ii) a TCP/IP protocol library; and, (iii) applications. In this work, we analyze how to compartmentalize these components by leveraging CHERI fine-grained protection. Specifically, we aim to evaluate the trade-offs and performance of encapsulating the different components into CHERI compartments. Due to the mentioned hardware limitations, to obtain representative measurements, we conduct our evaluation on the Arm Morello platform using the CheriBSD operating system (OS) [13]. We assess the behavior of single address space systems using the Data Plane Development Kit (DPDK) [4], a framework that allows direct access to network hardware from user space, bypassing the networking stack of the operating system. Both DPDK and the F-Stack TCP/IP library [24] were ported to CHERI Morello. A modified version of the CAP-VM Intravisor [8] was used to experiment with different isolation configurations. This work highlights two key points: (i) CHERI is a promising architecture for enhancing security in low-end embedded systems, as well as in scenarios where thread isolation is required to minimize overhead, and (ii) the overhead introduced by this architecture is minimal.

II. Background

A. CHERI

CHERI [14] is a technology that enhances Instruction Set Architectures (ISAs)—e.g., RISC-V, Arm—with *capability*-based primitives, improving software security and preventing vulnerabilities. CHERI introduces two key concepts.

Fine-grained code protection using capabilities. CHERI replaces traditional pointers with "capability pointers". These pointers include metadata such as bounds, access permissions, and integrity tag—which carry information on the validity of a *capability*. These metadata are stored within the pointer, doubling its size. Hence, the size of the *capability* is twice the size of the native platform register size—i.e., a 64-bit platform has *capabilities* of 128-bit. The use of *capabilities* restricts memory access and permissions at a granular level, ensuring that software can only access what it is authorized to. *Capabilities* can only derive from valid *capabilities* (valid provenance) and new *capabilities* are produced with less or equal privilege as their parent *capability* (monotonicity), i.e., read *capability* cannot form a read/write *capability*. CHERI provides two different types of compilation: *hybrid* mode, where the programmer specifies which pointers are *capabilities* using keywords in the C code, and *pure* mode, where every pointer is treated as a *capability*. This dual-mode approach allows legacy code to run unmodified while gradually porting software to a fully *capability*-based protection system.

Secure software compartmentalization. *Capabilities* in CHERI are used to enforce isolation within the same address space. Different components of a program can be isolated from each other, with each *compartment* having its own set of permissions and memory bounds, making it harder for malicious code to affect other parts of the system.

In *hybrid* architectures, compartmentalization is preserved by new special *capability* registers: the Default Data Capability (*DDC*), which defines the boundaries of the compartment, and the Program Counter Capability (*PCC*), which defines the *capability* of the program counter. This means that, if an application tries to execute a load or store instruction that goes outside its DDC, the architecture will generate a signal decoded as *Capability Out-of-Bounds exception*.

CHERI *sealing* implements robust compartmentalization, where *capabilities* are restricted to specific tasks or regions of code. Through sealing, a *capability* can be locked (or sealed) to a specific code or data and can only be unlocked through a special mechanism.

B. CAP-VMs

CAP-VMs [8], is an example of a dynamic approach to compartmentalization with CHERI. CAP-VMs provide VM-like abstractions to isolate system applications—in a single-address-space environments—by leveraging CHERI memory *capabilities*. The CAP-VM approach comprises three main elements. (i) A host OS Kernel: a CHERI-aware kernel that provides the primitives for synchronization, execution contexts, and I/O operations. Similarly to CAP-VM [8], we used CheriBSD. (ii) The *Intravisor*: a process responsible to manage configuration, isolation, and lifecycle of *capability*-VMs (cVMs). It distributes memory *capabilities* to cVMs and has access to all cVM memory regions. (iii) *capability-VM* (cVM): an isolated application component (*hybrid* or *pure* mode) that run as a thread of the Intravisor.

The Intravisor is in charge of the configuration of system *capabilities* among the various cVMs, i.e., of the definition of the isolated memory ranges. For cVMs, the Intravisor also acts as a proxy between the cVM *capability* world and CheriBSD. cVMs do not have direct access to the host OS syscalls, but must use instead a *trampoline* proxy table provided by the Intravisor that correctly handles the *capabilities* and mediates the access to the OS.

In this work, we leverage the minimal trusted computing base (TCB) of the Intravisor, which makes it practical for integration into embedded systems.

C. DPDK

The *Data Plane Development Kit* [4] is an Open Source set of libraries and drivers designed to accelerate packet processing in high-performance networking applications. DPDK bypasses the traditional kernel network stack and interacts directly with network hardware. This leads to better performance, but less isolation with the application. DPDK also operates in polling mode to reduce the latency caused by interrupt-triggered context switches. To detach the NIC from the kernel-space, DPDK uses a kernel-module.

Using DPDK, the management of network devices can be encapsulated within a cVM, thus isolating network operations from other system components and leveraging CHERI's *capability*-based security to ensure compartmentalization.

DPDK does not provide any TCP/IP protocol implementation, but only focuses on packet I/O and network hardware acceleration. Applications using DPDK must either implement their own protocol stack or integrate an existing user-space networking stack like the Open-Source F-Stack library [24]. In this paper, we ported F-Stack on our CHERI architecture and integrated it in our DPDK cVMs. Compared to other TCP/IP libraries built on top of DPDK, the existing integration of F-Stack with the FreeBSD network stack facilitated a more efficient porting process to CheriBSD.

We selected *iperf3* [31] as an application for the evaluation of our compartmentalized network. *iperf3* allows to define a server-client connection to measure the maximum bandwidth achievable.

III. Network Stack Compartmentalization Options

Different alternatives exist for the compartmentalization of a network stack using CHERI [20], [42], [43]. Each alternative entails trade-offs regarding its intrusiveness (i.e., the amount of changes required), overheads, and potential scalability. To this end, this work focuses on two possible system designs and compares them with a baseline scenario that does not use CHERI.

While we target resource-constrained systems that lack additional memory protection mechanisms, the trade-offs presented are also applicable to user-space device management with strong inter-thread isolation. This design is particularly efficient when low overhead is essential, leveraging multi-threading solutions to achieve this balance.

An overview of our CHERI-enabled designs—namely, Scenario 1 and Scenario 2—is presented in Figure 1 and Figure 2. Additionally, we define a *Baseline* scenario for comparison. This serves as a reference point for evaluating the two CHERI-enabled scenarios.

We run our experiments on the Arm Morello on top of the CheriBSD host OS. For the two CHERI-enabled scenarios, the Intravisor is responsible for the configuration of the compartments and mediates the syscalls between compartments and CheriBSD. Since the Morello built-in Ethernet device is not supported by DPDK, we chose a PCI card *Intel 82576 Gigabit Network Connection* with two Ethernet ports.

A. Scenarios

Baseline. *Baseline* consists of a non-CHERI full-network stack. In this configuration, DPDK, F-Stack, and *iperf3* run on two different processes (comparison with Scenario 1), and as a single process (Scenario 2). This setup represents a system that uses the MMU to isolate the two processes, which limits the ability to achieve fine-grained isolation.

Scenario 1. In Scenario 1, each compartment (*cVM1*, *cVM2*) contains one network application (*iperf3*), the F-Stack TCP/IP library, and the DPDK user-space network layer. The components run in *hybrid* mode and are linked against a modified *musl libc* library that provides the *trampoline* wrappers towards the Intravisor. Each *cVM* works as independent network processing unit, with its own dedicated stack and Ethernet interface ("eth0" and "eth1"). This configuration provides a complete compartmentalization approach, ensuring that any security breach does not affect the other *cVMs*. Moreover, it also serves as a baseline for comparing performance and isolation. The *trampoline* function is only executed when *musl libc* syscalls are executed.

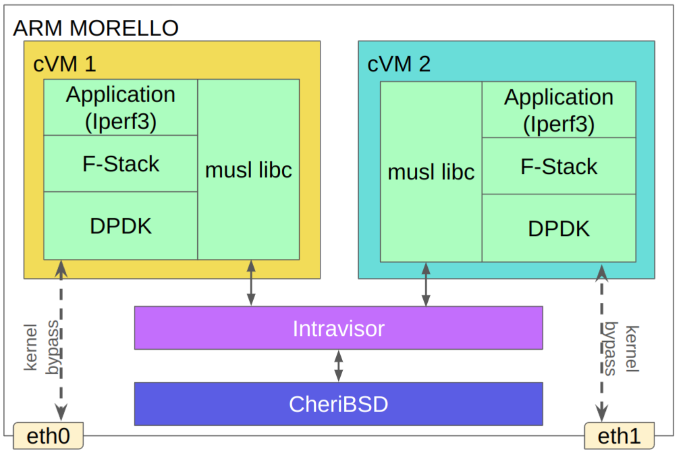


Fig. 1. Scenario 1: Replication of the entire stack into two different *cVMs*. The arrow shows where the *trampoline* function is executed.

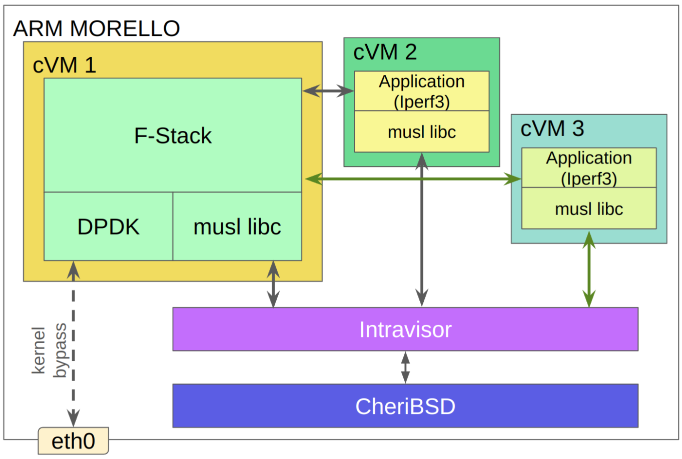


Fig. 2. Scenario 2: Isolation of two distinct applications *cVMs* from a F-Stack and DPDK cVM in the contented variant.

Scenario 2. This scenario separates the application (*iperf3*) from F-Stack and DPDK, thus allowing for potential space optimization in constrained environments. Here, *cVM1* runs the TCP/IP library and DPDK stack, while *cVM2* (and *cVM3*) execute (separate instances of) the application. All compartments link against the modified *musl libc* library to interact with the Intravisor. *cVM1* is responsible to read/write from/to the Ethernet driver queues and sort to the right file descriptor, when requested. This scenario requires a *mutex* to coordinate the execution of the F-Stack API functions and the main-loop execution, which creates a potential contention issue for this shared resource. Therefore, we have evaluated this scenario in two configurations (see Section IV) with one application (uncontented) or two compartmentalized applications (contented) respectively. Figure 2 depicts the contented configuration, i.e., with two compartmentalized applications.

B. Implementation

Intravisor. In CAP-VMs [8], *hybrid cVMs* encapsulating applications use a combination of *musl libc* [28] and Linux Kernel Library (LKL) [29]. LKL turns the Linux kernel into a library, providing kernel services—such as file systems and networking—to be run in user-space, making the resulting application highly portable to various environments—e.g., virtualized and bare-metal. However, this may increase the attack surface, and introduces overhead by emulating kernel syscalls in user-space. Conversely, our DPDK and F-Stack are designed to fully execute in user-space and interact with the kernel only at boot time. They can therefore be streamlined within *cVMs*, removing the additional LKL layer. We directly connected *musl libc* in the Intravisor substituting supervisor call instructions (`svc`) with dedicated *trampoline* functions. Specifically, a *trampoline* passes through the syscall ID and arguments, stores register states. It also loads the correct PCC and DDC, and use them to jump into the cVM/Intravisor using CHERI specific instruction (e.g., `blrs` for the Arm Morello). Avoiding LKL helps reducing overhead as well as the overall footprint and attack surface, making the *cVMs* potentially more secure. Similarly to FreeBSD, CheriBSD native libc is not fully compatible with *musl libc*. We adapted the Intravisor proxy function to properly translate *musl libc* calls into CheriBSD libc equivalents. For instance, *musl libc* uses *futex* for thread synchronization, while CheriBSD uses *umtx* [30].

DPDK. DPDK was ported to the CHERI Morello using *hybrid* mode [3]. However, the existing support is very limited and does not initialize nor use any NIC interface. We implemented the module that detaches the NIC from kernel-space and attaches it to user-space, ensuring that the memory allocations it requests are performed with the correct permission flags. Minor adjustments were required to enable execution using *musl libc*.

F-Stack. F-Stack leverages the FreeBSD network stack, but, prior to this work, no available implementations of F-Stack for CHERI existed. The first step in our implementation involved porting F-Stack to work with CheriBSD. This required adaptations on both OS and F-Stack sides. CheriBSD was modified to accommodate the specific requirements of F-Stack, by streamlining the use of the network stack. DPDK and F-Stack operate in polling mode, meaning that the application must explicitly request to read/write from/to the device. After an initialization phase, a main-loop is executed, with the key tasks being: (i) process the ring buffers of the DPDK Ethernet driver; and, (ii) execute a user-defined function where calls to F-Stack API functions can be made. F-Stack also required adaptations to work with *capabilities*. We extended its data structures to use *capabilities* and we have extended its Application Programming Interface (API).[1] Applications can integrate with F-Stack through its API, which closely resembles the BSD socket API. For instance, F-Stack exposes `ff_socket()` and `ff_write()` functions that are the equivalent of `socket()` and `write()`. This allows developers to adapt existing network applications for F-Stack with minimal changes. To work with *capabilities*, the signatures of such APIs has to be modified. For instance, the signature of the `ff_write()` has been modified as follow:

```
-    ssize_t ff_write(int fd, const void *
     buf, size_t nbytes);
+    ssize_t ff_write(int fd, const void *
     __capability buf, size_t nbytes);
```

The total amount of modified lines of code (LoC) to achieve our objectives is shown in Table I. Since the FreeBSD version originally supported by F-Stack is v13.0, while we used CheriBSD v23.11 based on FreeBSD v14.0, the modifications in Table I include not only a transition to a CHERI-enabled library, but also the adjustments needed to adapt the native library to the version of CheriBSD. Overall, this is in line with the usually expected modifications to port a library [17].

TABLE I
NUMBER OF LINE OF CODES ADDED/MODIFIED

Library	LoC	global amount in percentage
F-Stack	152	0.99%

Application. To work with our setup, we initially ported *iperf3* to work with the F-Stack API. Next, we replaced the `select` function, with the `epoll` mechanism, which adapts better to F-Stack. For Scenario 2, we also implemented the wrapper functions to the API of F-Stack to do the cross-compartment jump between the running application and the *cVM1*.

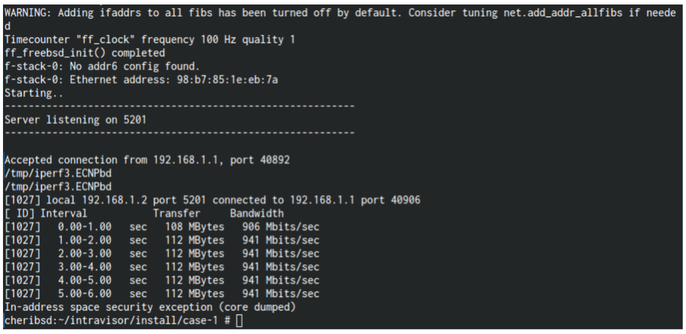
IV. EVALUATION

In this section, we evaluate the performance and security benefits of the system architecture and isolation configurations implemented on the Morello platform. In addition to verifying the security achieved through CHERI compartmentalization, the evaluation focuses on two main aspects: performance in terms of TCP bandwidth and execution time cost of a representative function, the `ff_write()`. To measure these values, we use `clock_gettime()` with CLOCK_MONOTONIC_RAW. The `ff_write()` execution time serves as a meaningful performance indicator because this function has a direct impact on the overall system efficiency. Note that, since DPDK and F-Stack operate in polling mode to minimize latency, CPU utilization is not a meaningful indicator.

Under CHERI, each network-stack compartment is restricted to its DDC *capability*. We verified the effectiveness of compartmentalization modifying applications to access memory ranges outside their valid boundaries. As expected, CHERI triggers a CAP-out-of-bound exceptions, as shown in Figure 3.

For the network-related evaluations, we executed our modified *iperf3* to calculate the maximum bandwidth in both server (receiver) and client (sender) modes. We also computed the *efficiency*, as the resulting bandwidth divided by the maximum

[1] The code for both DPDK and F-Stack extensions is available at https://github.com/donato-ferraro/dpdk-morello-capvms and https://github.com/donato-ferraro/fstack-cheri



```
WARNING: Adding ifaddrs to all fibs has been turned off by default. Consider tuning net.add_addr_allfibs if neede
d
Timecounter "ff_clock" frequency 100 Hz quality 1
ff_freebsd_init() completed
f-stack-0: No addr6 config found.
f-stack-0: Ethernet address: 98:b7:85:1e:eb:7a
Starting..
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------

Accepted connection from 192.168.1.1, port 40892
/tmp/iperf3.ECNPbd
/tmp/iperf3.ECNPbd
[1027] local 192.168.1.2 port 5201 connected to 192.168.1.1 port 40906
[ ID] Interval           Transfer     Bandwidth
[1027]   0.00-1.00   sec   108 MBytes   906 Mbits/sec
[1027]   1.00-2.00   sec   112 MBytes   941 Mbits/sec
[1027]   2.00-3.00   sec   112 MBytes   941 Mbits/sec
[1027]   3.00-4.00   sec   112 MBytes   941 Mbits/sec
[1027]   4.00-5.00   sec   112 MBytes   941 Mbits/sec
[1027]   5.00-6.00   sec   112 MBytes   941 Mbits/sec
In-address space security exception (core dumped)
cheribsd:~/intravisor/install/case-1 # ▯
```

Fig. 3. Applications accessing memory outside their boundaries cause exceptions under CHERI.

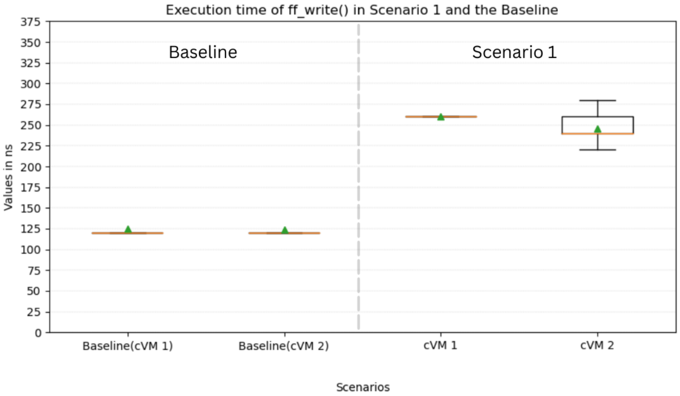


Fig. 4. Execution time of the `ff_write()` function in Scenario 1, compared with *Baseline*.

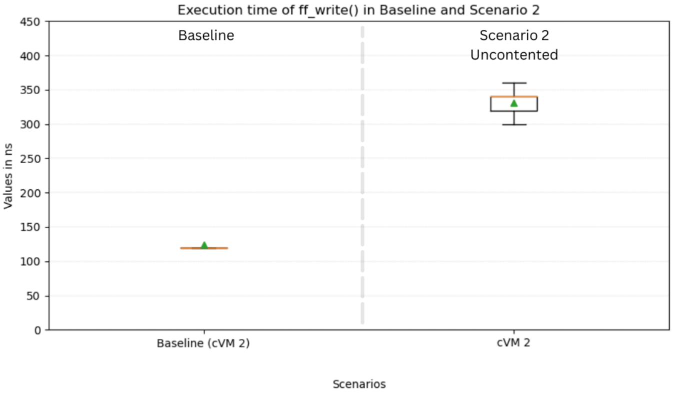


Fig. 5. Execution time of the `ff_write()` function in Scenario 2, compared with *Baseline*.

bandwidth *theoretically* achievable (1Gbps for each Ethernet port). In Scenario 1 and *Baseline*, where both Ethernet ports are in use, we are not achieving high efficiency due to the hardware limitations imposed by the PCI NIC. In all scenarios, as shown in Table II, we reached the maximum bandwidth possible with our hardware setup. In the contented Scenario 2, we notice an unbalance among the achieved bandwidth, which can be attributed to the lack of mechanisms for fairness control. We defer the investigation of Quality-of-Service (QoS) approaches or the integration of DPDK QoS features to future works.

In the evaluation of the execution time of `ff_write()`, each test measures the execution of 1 million iterations, and Figures 4-6 present averages and standard deviations as box plots. Moreover, in all the experiments, outliers ($\approx$ 10% of the iterations) are removed with a standard IQR strategy. Figure 4 compares Scenario 1 with *Baseline*. The first two boxes from the left depict the execution time for *Baseline*, while the other two the execution time of Scenario 1. Note that, in cVMs we can't directly access the timers of the system, the execution time always includes a cross-compartment jump to the Intravisor, the execution of the syscall in CheriBSD, and the return from the Intravisor to access them. As seen in Figure 4, the impact of the CHERI compartment is minimal, and amounts to approximately 125 ns, corresponding to the additional indirections required by the musl - Intravisor mechanism. In both *Baseline* (*cVM1* and *cVM2*) and *cVM1* for Scenario 1, the 25th and 75th percentiles are identical, leading to the absence of a visible box in the box plot. This indicates that more than 50% of the results were identical. The experiments were repeated several times, consistently yielding the same outcomes.

TABLE II
RESULTS OF TCP BENCHMARKS IN THE THREE SCENARIOS, BOTH SERVER AND CLIENT SIDES. VALUES EXPRESSED IN MBIT/S

Baseline				
Modes	Server	Efficiency	Client	Efficiency
Baseline (*cVM1*)	658	65.8%	757	75.7%
Baseline (*cVM2*)	658	65.8%	757	75.7%
Scenario 1				
Modes	Server	Efficiency	Client	Efficiency
cVM1	658	65.8%	757	75.7%
cVM2	658	65.8%	757	75.7%
Baseline				
Modes	Server	Efficiency	Client	Efficiency
Baseline (*cVM2*)	941	94.1%	941	94.1%
Scenario 2 (uncontented)				
Modes	Server	Efficiency	Client	Efficiency
cVM2	941	94.1%	941	94.1%
Scenario 2 (contented)				
Modes	Server	Efficiency	Client	Efficiency
cVM2	470	94%	531	106.2%
cVM3	470	94%	410	82%

In Scenario 2, the cost of the cross-compartment jump and return in the cVM is included in the measurements. Moreover, due to the required synchronization between the main-loop of F-Stack and the execution of the `ff_write()` (see III), the measures also include the time necessary to acquire the mutex. Figure 5 shows the execution time measurements of the `ff_write()` of the uncontented setup compared with *Baseline*. In this case, we increased the interval between two consecutive `ff_write()` to reduce the possibility to be blocked for a long time by the mutex. Compared to Scenario 1, despite the additional indirection to jump between the two *cVMs* and the handling of the mutex, the experienced slow down is contained to approximately 200 ns.

The effect of the mutex contention is evident in the contented Scenario 2, where three cVMs can acquire and wait on the mutex. As shown in Figure 6, operations on the mutex result in

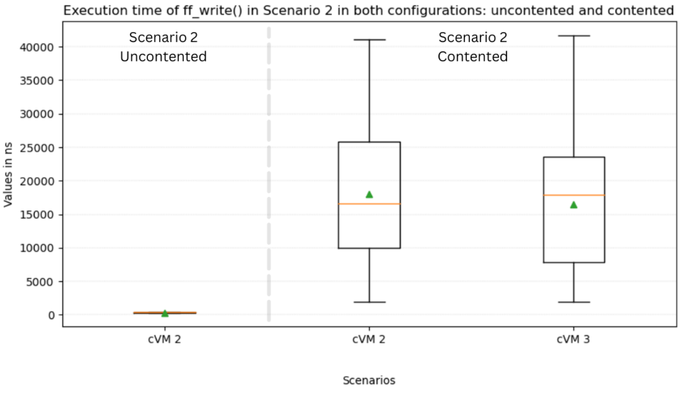


Fig. 6. Execution time of the `ff_write()` function in Scenario 2 in uncontented and contented configurations.

an overhead of around 19.000 ns (152× slowdown). Nonetheless, despite the overhead introduced for handling the mutex, the contented Scenario 2 reaches the maximum bandwidth supported by the hardware in both server (receiver) and client (sender) modes (see Table II). As future work, we plan to investigate in details the impact of different locking strategies to further reduce the overhead of our designs.

V. Related Work

Several works have addressed different aspects of ensuring security in embedded systems [21]–[23]. These efforts span the entire embedded system stack, from designing robust and energy-efficient hardware to developing secure, lightweight operating systems [37] and strong encryption protocols. For instance, the OS plays a critical role in securing UAVs, as it must implement countermeasures specific to embedded systems to defend against different security attacks [18]. Prior studies on compartmentalizing network stacks in OSs, like FreeRTOS [47] and RIOT [46], have relied on coarse-grained (task-level) isolation using MPUs or MMUs, still leaving systems vulnerable to memory-related attacks such as buffer overflow [40], [41]. CHERI can mitigate these vulnerabilities by providing fine-grained memory access control, replacing or cooperating with MPU/MMU-based solutions [42]. Other hardware-based solutions also address compartmentalized software design by isolating critical trusted security functions, like authentication, from the main untrusted OS. Nevertheless, they still rely on MPU-like mechanisms for isolation. For instance, Arm TrustZone [45] uses Secure Attribution Unit (SAU) or Implementation Defined Attribution Unit (IDAU) to separate memory into secure and insecure partitions. As a consequence, CHERI would allow enhanced internal memory protection, mitigating attacks from the untrusted OS [44].

CHERI-enabled architectures, help addressing mixed-criticality environments by isolating vulnerable components. One notable work on CHERI compartmentalization in embedded systems is CompartOS [20]. The authors leverage CHERI *capabilities* to isolate linkage modules by using a CHERI extension to the library *libdl*. However, as also documented in this work, this approach introduces some overhead compared to static solutions, which are typically more common in the safety-critical embedded systems. CAP-VMs [8] was also extended with a new mechanism to reduce the duplication of the libraries running in the cVMs has been implemented [36]. This mechanism could also be used as a future integration for our work to reduce the memory utilization.

DPDK has been routinely used in all those contexts requiring low network overhead and high bandwidth (e.g., [33]–[35]). Among those, F-Stack is the predominant TCP/IP library used, with good scalability and performance [32]. In addition, the high-performance DPDK has been integrated into embedded systems. One example of this integration involves modifying LwIP [38]—a lightweight TCP/IP stack designed for embedded systems—to run on top of DPDK, exploring the use of DPDK also for constrained environments [19].

VI. Conclusion & Future Works

This work has presented the potential of CHERI to provide strong security properties through compartmentalization of the software components of a network stack. Specifically, we presented different trade-offs for designs that isolate DPDK, F-Stack, and a network benchmark application using Intravisor-based *cVMs*. Our evaluation on the CHERI-enabled Arm Morello platform has validated the effectiveness of CHERI compartmentalization and has shown the low overhead potential of our *cVM*-based design. However, despite achieving full bandwidth, the evaluation also underlined the impact of synchronization among different *cVMs*.

Further research will focus on isolating all the components within the system, including finer-grained compartmentalization of the network stack and peripheral drivers. Additional scenarios include: (i) the separation of DPDK from F-Stack and the application; and, (ii) the separation of the entire stack. Future analysis will consider the design and potential impact of different synchronization strategies among compartments.

References

[1] Runyu Pan and Gabriel Parmer. 2019. ”MxU: Towards Predictable, Flexible, and Efficient Memory Access Control for the Secure IoT”. ACM Trans. Embed. Comput. Syst. 18, 5s, Article 103 (October 2019), 20 pages. https://doi.org/10.1145/3358224

[2] D. Dasari, B. Akesson, V. Nélis, M. A. Awan and S. M. Petters, ”Identifying the sources of unpredictability in COTS-based multicore systems,” 2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES), Porto, Portugal, 2013, pp. 39-48, doi: 10.1109/SIES.2013.6601469.

[3] spdk-morello/dpdk: Data Plane Development Kit; https://github.com/spdk-morello/dpdk/tree/morello

[4] Home - DPDK. https://www.dpdk.org/

[5] CVE-2023-52370 — CVE. https://www.cve.org/CVERecord?id=CVE-2023-52370

[6] CVE-2023-6951 — CVE. https://www.cve.org/CVERecord?id=CVE-2023-6951

[7] CVE-2024-38951 — CVE. https://www.cve.org/CVERecord?id=CVE-2024-38951

[8] Vasily A. Sartakov and Lluís Vilanova and David Eyers and Takahiro Shinagawa and Peter Pietzuch. 2022. ”CAP-VMs: Capability-Based Isolation and Sharing in the Cloud”. OSDI ’22. isbn: 978-1-939133-28-1. https://www.usenix.org/conference/osdi22/presentation/sartakov

[9] Open Source Autopilot for Drones - PX4 Autopilot. https://px4.io/

[10] L. Meier, D. Honegger and M. Pollefeys, ”PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms,” 2015 IEEE International Conference on Robotics and Automation (ICRA), Seattle, WA, USA, 2015, pp. 6235-6240, doi: 10.1109/ICRA.2015.7140074.

[11] R. Grisenthwaite, G. Barnes, R. N. M. Watson, S. W. Moore, P. Sewell and J. Woodruff, ”The Arm Morello Evaluation Platform—Validating CHERI-Based Security in a High-Performance System,” in IEEE Micro, vol. 43, no. 3, pp. 50-57, May-June 2023, doi: 10.1109/MM.2023.3264676.

[12] Unveiling Sonata: Affordable CHERI Hardware for Embedded Systems. https://www.design-reuse.com/news/55539/sonata-cheri-risc-v-prototype-board.html

[13] CheriBSD. https://www.cheribsd.org/

[14] Robert N. M. Watson, et al., ”Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)”, Technical Report UCAM-CL-TR-987, Computer Laboratory, September 2023. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-987.pdf

[15] Introduction - MAVLink Devoloper Guide. https://mavlink.io/en/

[16] AutoCHERI. https://autocheri.tech/

[17] R. N. M. Watson et al. ”CHERI: Hardware-Enabled C/C++ Memory Protection at Scale” in IEEE Security & Privacy, vol. 22, no. 4, pp. 50-61, July-Aug. 2024, doi: 10.1109/MSEC.2024.3396701.

[18] S. Iqbal, ”A Study on UAV Operating System Security and Future Research Challenges,” 2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC), Las Vegas, NV, USA, 2021, pp. 0759-0765, doi: 10.1109/CCWC51732.2021.9376151.

[19] R. Rajesh, K. B. Ramia and M. Kulkarni, ”Integration of LwIP Stack over Intel(R) DPDK for High Throughput Packet Delivery to Applications,” 2014 Fifth International Symposium on Electronic System Design, Surathkal, India, 2014, pp. 130-134, doi: 10.1109/ISED.2014.34.

[20] Hesham Almatary et al. ”CompartOS: CHERI Compartmentalization for Embedded Systems,” 2022.https://arxiv.org/abs/2206.02852

[21] H. U. Rehman, M. Asif and M. Ahmad, ”Future applications and research challenges of IOT,” 2017 International Conference on Information and Communication Technologies (ICICT), Karachi, Pakistan, 2017, pp. 68-74, doi: 10.1109/ICICT.2017.8320166.

[22] Srivaths Ravi, Anand Raghunathan, Paul Kocher, and Sunil Hattangady. 2004. ”Security in embedded systems: Design challenges. ACM Trans. Embed. Comput. Syst. 3, 3 (August 2004), 461–491. https://doi.org/10.1145/1015047.1015049

[23] A. Aldahmani, B. Ouni, T. Lestable and M. Debbah, ”Cyber-Security of Embedded IoTs in Smart Homes: Challenges, Requirements, Countermeasures, and Trends,” in IEEE Open Journal of Vehicular Technology, vol. 4, pp. 281-292, 2023, doi: 10.1109/OJVT.2023.3234069.

[24] F-Stack — High Performance Network Framework Based On DPDK. https://www.f-stack.org/

[25] D. Papp, Z. Ma and L. Buttyan, ”Embedded systems security: Threats, vulnerabilities, and attack taxonomy,” 2015 13th Annual Conference on Privacy, Security and Trust (PST), Izmir, Turkey, 2015, pp. 145-152, doi: 10.1109/PST.2015.7232966.

[26] A. C. Panchal, V. M. Khadse and P. N. Mahalle, ”Security Issues in IIoT: A Comprehensive Survey of Attacks on IIoT and Its Countermeasures,” 2018 IEEE Global Conference on Wireless Computing and Networking (GCWCN), Lonavala, India, 2018, pp. 124-130, doi: 10.1109/GCWCN.2018.8668630.

[27] P. Rahimi, A. K. Singh, X. Wang and A. Prakash, ”Trends and Challenges in Ensuring Security for Low-Power and High-Performance Embedded SoCs,” 2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), Singapore, Singapore, 2021, pp. 226-233, doi: 10.1109/MCSoC51149.2021.00041.

[28] musl libc. https://musl.libc.org/

[29] O. Purdila, L. A. Grijincu and N. Tapus, ”LKL: The Linux kernel library,” 9th RoEduNet IEEE International Conference, Sibiu, Romania, 2010, pp. 328-333.

[30] _umtx_op(2) - FreeBSD Manual. https://man.freebsd.org/cgi/man.cgi?query=_umtx_op&sektion=2&n=1

[31] iPerf - The TCP, UDP and SCTP network bandwidth measurement tool. https://iperf.fr/

[32] A. B. Narappa, F. Parola, S. Qi and K. K. Ramakrishnan, ”Z-Stack: A High-Performance DPDK-Based Zero-Copy TCP/IP Protocol Stack,” 2024 IEEE 30th International Symposium on Local and Metropolitan Area Networks (LANMAN), Boston, MA, USA, 2024, pp. 100-105, doi: 10.1109/LANMAN61958.2024.10621881.

[33] Abhishek Bhattacharyya, Shunmugapriya Ramanathan, Andrea Fumagalli, Koteswararao Kondepu. An end-to-end DPDK-integrated open-source 5G standalone Radio Access Network: A proof of concept. Computer Networks. https://doi.org/10.1016/j.comnet.2024.110533.

[34] V. Bode, C. Trinitis, M. Schulz, D. Buettner and T. Preclik, ”Adopting User-Space Networking for DDS Message-Oriented Middleware,” 2024 IEEE International Conference on Pervasive Computing and Communications (PerCom), Biarritz, France, 2024, pp. 36-46, doi: 10.1109/PerCom59722.2024.10494460.

[35] J. Umeike, S. Agarwal, N. Lazarev and M. Alian, ”Userspace Networking in gem5,” 2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Indianapolis, IN, USA, 2024, pp. 179-191, doi: 10.1109/ISPASS61541.2024.00026.

[36] Vasily A. Sartakov, Lluis Vilanova, Munir Geden, David Eyers, Takahiro Shinagawa, Peter Pietzuch. ”ORC: Increasing Cloud Memory Density via Object Reuse with Capabilities”. In 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23) (pp. 573-587). USENIX Association.

[37] Zuepke, Alexander, et al. ”For Safety and Security Reasons: The Cost of Component-Isolation in IoT” Logistics, Supply Chain, Sustainability and Global Challenges, vol. 7, no. 1, Sciendo, 2016, pp. 41-50. https://doi.org/10.1515/jlst-2016-0004

[38] LwIP - A Lightweight TCP/IP stack - https://savannah.nongnu.org/projects/lwip/

[39] Introduction - MAVLink Developer Guide - https://mavlink.io/en/

[40] Al-Boghdady, Abdullah, Khaled Wassif, and Mohammad El-Ramly. 2021. ”The Presence, Trends, and Causes of Security Vulnerabilities in Operating Systems of IoT’s Low-End Devices” Sensors 21, no. 7: 2329. https://doi.org/10.3390/s21072329

[41] G. Mullen and L. Meany, ”Assessment of Buffer Overflow Based Attacks On an IoT Operating System,” 2019 Global IoT Summit (GIoTS), Aarhus, Denmark, 2019, pp. 1-6, doi: 10.1109/GIOTS.2019.8766434.

[42] H. Xia et al., ”CheriRTOS: A Capability Model for Embedded Devices,” 2018 IEEE 36th International Conference on Computer Design (ICCD), Orlando, FL, USA, 2018, pp. 92-99, doi: 10.1109/ICCD.2018.00023.

[43] S. Amar et al. ”CHERIoT: Complete Memory Safety for Embedded Devices”, 2023, In Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO ’23). Association for Computing Machinery, New York, NY, USA, 641–653. https://doi.org/10.1145/3613424.3614266

[44] T. Van Strydonck et al., ”CHERI-TrEE: Flexible enclaves on capability machines,” 2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P), Delft, Netherlands, 2023, pp. 1143-1159, doi: 10.1109/EuroSP57164.2023.00070.

[45] A. Arm, “Arm TrustZone Technology for the Armv8-M Architecture” ARM Limited, 2018, https://developer.arm.com/documentation/100690/.

[46] E. Baccelli et al., ”RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT,” in IEEE Internet of Things Journal, vol. 5, no. 6, pp. 4428-4440, Dec. 2018, doi: 10.1109/JIOT.2018.2815038

[47] Fei Guan, Long Peng, Luc Perneel, and Martin Timmerman. 2016. Open source FreeRTOS as a case study in real-time operating system evolution. J. Syst. Softw. 118, C (August 2016), 19–35. https://doi.org/10.1016/j.jss.2016.04.063