RESEARCH-ARTICLE

Secure Scripting with CHERIoT MicroPython

DUNCAN LOWTHER, University of Glasgow, Glasgow, Scotland, U.K.

DEJICE JACOB, University of Glasgow, Glasgow, Scotland, U.K.

JACOB TREVOR, University of Glasgow, Glasgow, Scotland, U.K.

JEREMY SINGER, University of Glasgow, Glasgow, Scotland, U.K.

PDF Download
3708493.3712694.pdf
04 March 2026
Total Citations: 0
Total Downloads: 518

Published: 25 February 2025

Citation in BibTeX format

CC '25: 34th ACM SIGPLAN International Conference on Compiler Construction
March 1 - 2, 2025
NV, Las Vegas, USA

Conference Sponsors:
SIGPLAN

Secure Scripting with CHERIoT MicroPython

Duncan Lowther
University of Glasgow
Glasgow, United Kingdom
duncan.lowther@glasgow.ac.uk

Dejice Jacob
University of Glasgow
Glasgow, United Kingdom
dejice.jacob@glasgow.ac.uk

Jacob Trevor
University of Glasgow
Glasgow, United Kingdom
j.trevor.1@research.gla.ac.uk

Jeremy Singer
University of Glasgow
Glasgow, United Kingdom
jeremy.singer@glasgow.ac.uk

Abstract

The lean MicroPython runtime is a widely adopted high-level programming framework for embedded microcontroller systems. However, the existing MicroPython codebase has limited security features, rendering it a fundamentally insecure runtime environment. This is a critical problem, given the growing deployment of highly interconnected IoT systems on which society depends. Malicious actors seek to compromise such embedded infrastructure, using sophisticated attack vectors. We have implemented a novel variant of MicroPython, adding support for runtime security features provided in the CHERI RISC-V architecture as instantiated by the CHERIoT-RTOS system. Our new MicroPython port supports hardware-enabled spatial memory safety, mitigating a large set of common runtime memory attacks. We have also compartmentalized the MicroPython runtime, to prevent untrusted code from elevating its permissions and taking control of the entire system. We perform a multi-faceted evaluation of our work, involving a qualitative security-focused case study and a quantitative performance analysis. The case study explores the full set of five publicly reported MicroPython vulnerabilities (CVEs). We demonstrate that the enhanced security provided by CHERIoT MicroPython mitigate two heap buffer overflow CVEs. Our performance analysis shows a geometric mean runtime overhead of 48% for secure execution across a set of ten standard Python benchmarks, although we argue this is indicative of worst-case overhead on our prototype platform and a realistic deployment overhead would be significantly lower. This work opens up a new, secure-by-design approach to IoT application development.

CCS Concepts: • **Software and its engineering** → **Interpreters**; **Runtime environments**; • **Security and privacy** → *Embedded systems security*.

Keywords: CHERI, Capabilities, Cybersecurity, Python

ACM Reference Format:
Duncan Lowther, Dejice Jacob, Jacob Trevor, and Jeremy Singer. 2025. Secure Scripting with CHERIoT MicroPython. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction (CC '25), March 1–2, 2025, Las Vegas, NV, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3708493.3712694

1 Introduction

There are billions of commodity IoT devices deployed in the field. In order to make IoT system prototyping and development more accessible, many embedded systems support the *MicroPython* variant of the Python programming language. It is a user-friendly, cut-down, bytecode interpretive implementation of Python. The MicroPython virtual machine (VM) is designed primarily for execution on resource-constrained target devices. MicroPython ships with a range of domain-specific libraries for IoT applications, such as a lightweight Bluetooth low energy driver (`uble`).

Numerous high-profile cybersecurity incidents have clearly demonstrated the requirement for secure-by-design systems in the IoT domain [13, 32]. The recently released *CHERIoT* system [1, 2] is an IoT solution that incorporates the CHERI [28, 34] concept of capability-based runtime security via direct hardware support. In this paper, we present our work on adapting the MicroPython runtime for CHERIoT. The compelling advantage of this combination is the low-level architectural support for memory safety in managed execution environments.

We demonstrate effective use of CHERI hardware features. Specifically, we leverage the *spatial memory safety* properties of capabilities to mitigate CVE-2023-7158 [20] and CVE-2024-8948 [21]. Our MicroPython port also makes use of the *lightweight compartmentalization* facilities of CHERI capabilities, to address library-based software supply chain vulnerabilities. Compartments mitigate against vulnerabilities by reducing the attack surface as well as by limiting the damage from an exploit.

While we acknowledge that CHERI performance benchmarking is particularly fraught with difficulty [29], we feel it is important to present wallclock execution time results. We compare the CHERIoT RISC-V based Sonata board with a non-CHERI 32-bit embedded RISC-V configuration running on the same reconfigurable board. We report a 48% geometric mean performance overhead compared to the non-CHERI RISC-V when executing MicroPython benchmarks from its standard benchmark suite. We argue this is a high-watermark overhead score, given the non-CHERI interpreter is running on bare-metal rather than an RTOS. In addition, the FPGA model for CHERIoT on Sonata is continually being improved and optimized. The performance evaluation indicates significant potential for CHERI to be incorporated in typical IoT platforms.

Our work follows a case study based methodology, with a multi-faceted exploration of CHERI adoption in a real-world scenario. Specifically, we seek to address the following three research questions in a qualitative manner:

- **RQ1**: Is the CHERIoT-RTOS software ecosystem an appropriate environment for developing and deploying a lean programming language interpreter like MicroPython?
- **RQ2**: Does the security afforded by the capability model of CHERIoT-RTOS mitigate any vulnerabilities of the MicroPython interpreter within a standard threat model scenario?
- **RQ3**: How does the Arm Morello system compare with the RISC-V CHERIoT system, in terms of two deployment platforms at high technology readiness levels (TRLs) that support the CHERI concept?

2 Technical Background

2.1 CHERI

Capability Hardware Enhanced RISC Instructions (CHERI) is an extension for conventional processor architectures that enables memory safe program execution. The key characteristic of CHERI is the extension of pointer values into *capabilities*, which are unforgeable fat pointers containing runtime metadata including bounds, permissions and object type. Each capability is *tagged* with a one-bit validity tag that enforces capability monotonicity. This ensures that valid capabilities can only be derived from pre-existing capabilities, transitively depending on a primordial 'root' capability owned by the initial OS process at boot time. CHERI monotonicity enforcement ensures that derived capabilities constructed by user code can only possess permissions or bounds that do not exceed those of the source capability.

CHERI capabilities are generally sized as $2n+1$ bits, where n is the length of an architectural word. The additional bit is used for the validity tag which is stored out-of-band. The validity tag is maintained by the architecture and not directly addressable by user code. The n-bit pointer is stored in the first n bits of a capability value, and the associated metadata (bounds, permissions, object type) are stored in a compressed format in the second n bits of the capability [33].

Practical instantiations of the CHERI concept in realistic architectures include a variant of MIPS (now deprecated), Arm and RISC-V [31]. These generally work on 64-bit architectures with 129-bit compressed capabilities. CHERI models also run under emulation (generally using QEMU) and on FPGA hardware. The most prevalent high-performance CHERI implementation is the Arm Morello platform [10], which is a server-class quad-core 2.5GHz AArch64 processor. The Morello platform runs commodity OSs including CHERI variants of FreeBSD and Linux [26].

2.2 CHERIoT

CHERIoT [1, 2] is a variant of the CHERI micro-architectural extension, specialized for embedded hardware devices. The major adaptation is a scaling down to a 32-bit RISC-V embedded platform, with 65-bit capabilities.

CHERIoT is instantiated as an adaptation of the 32-bit RISC-V Ibex microcontroller core. The CHERIoT version implements capability extensions in addition to the standard RV32EMCB instruction set. The 65-bit capability format used by CHERIoT imposes greater constraints on metadata compared to architectures with 129-bit capabilities. In particular, the capability offset is always non-negative: if the address value in the capability drops below the base address, its validity tag is immediately cleared with no 'buffer'. Permissions are not encoded as individual bitflags but are linked to allow compression to 6 bits and `W^X` is architecturally enforced. The object type bitfield is only 3 bits wide, supporting a maximum 8 distinct object types. CHERIoT also supports tight control for software compartments.

CHERIoT has been co-designed with a prototype real-time operating system (CHERIoT RTOS) which provides a minimal trusted computing base (TCB). The TCB includes four main components:

1. the *loader*, which executes initially and configures capabilities for the rest of the system
2. the *context switcher*, for switching between threads (units of compute) and compartments (units of data)
3. the *allocator*, which allocates runtime memory from a shared global heap
4. the *scheduler*, which chooses the next runnable thread

Other components are adapted from open-source code, e.g. FreeRTOS network stack. User applications are compiled and statically linked with CHERIoT-RTOS, using a custom Clang/LLVM toolchain and an Xmake build configuration. This is the workflow we use to deploy our MicroPython interpreter on CHERIoT-RTOS, (Section 4).

2.3 Sonata

The Sonata system [15], shown in Figure 1, is a low-cost microcontroller board featuring a CHERIoT Ibex RISC-V CPU implemented as an FPGA soft-core. The system supports an extensive range of peripherals including I^2C, SPI, GPIO, and

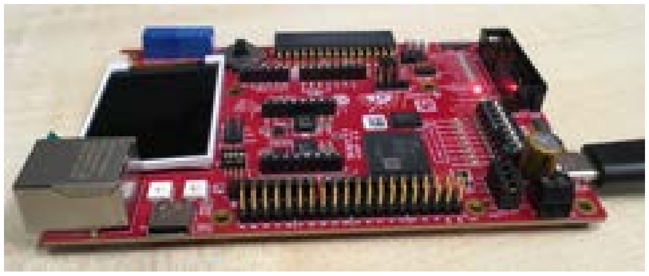

Figure 1. Sonata development board for CHERIoT

USB. The soft-core is clocked at 30 MHz. It is a 32-bit RISC-V (RV32EMCB) processor with CHERI extensions for hardware capability support (65-bit capabilities).

The main goal of Sonata is to be an evaluation board, to enable embedded systems developers to explore the utility of CHERI in various application contexts. The Sonata microcontroller board is a deliverable from the Sunburst project [16] funded by UKRI as part of the Digital Security by Design (DSbD) programme.

2.4 MicroPython

MicroPython compiles Python source code into a bytecode representation which is executed with a runtime interpreter. The system includes a number of core libraries to enable well known Python modules to be utilised in embedded applications. MicroPython is primarily intended to target a wide range of microcontrollers including ESP32, Arm Cortex-M, and Arduino. There is also a Unix process variant of MicroPython, which has been adapted for CHERI in prior work on the 64-bit Arm Morello platform [17, 18], featuring spatial memory safety enforced by tight bounds.

MicroPython is implemented in C, with the codebase having around 750 kSLoC in total. However, the core bytecode interpreter is only 50 kSLoC with the remainder of the codebase comprising library implementation code and platform-specific porting code.

3 Threat Model

In a microcontroller environment with direct physical addressing, there is no hardware memory protection of the kind generally provided in a high-level OS with virtual memory. For this reason, spatial memory safety via bounds-checked capabilities is an important security feature. CHERIoT's low-overhead compartmentalization can ensure separation between mutually distrusting application components.

To assess the effectiveness of CHERI features in our threat model, we need to define an attacker's methods that can be used against the target system, along with potential knowledge and resources the attacker has at their disposal.

We are working with a high-level memory safe programming language (i.e. Python) executed by a relatively mature open-source interpretive runtime (i.e. MicroPython). We assume that an attacker is able to:

- inspect, but not modify, the MicroPython source code with a view to identifying potential vulnerabilities.
- provide arbitrary Python code to be interpreted in MicroPython.
- provide compromised C libraries (either at build time for static linkage or at execution time for dynamic linkage).

We anticipate possible threat vectors will include:

- exploitation of known security vulnerabilities, such as publicly disclosed Common Vulnerabilities and Exposures (CVEs) in the MicroPython codebase.
- manipulation of low-level memory using the MicroPython `uctypes` raw memory interface.
- manipulation of low-level memory via external C library code.

In the remainder of this paper, we aim to show that our CHERIoT MicroPython runtime incorporates mitigations against these particular threat vectors.

4 CHERIoT Modifications to MicroPython

In this section, we briefly describe the source code changes that were required to build and execute MicroPython on the CHERIoT-RTOS platform.

4.1 Build System Integration

MicroPython utilizes a moderately complex GNU Make build system to compile various ports and modules. This allows the build system to separate various porting efforts and optionally include modules into the MicroPython image file. However, CHERIoT-RTOS uses Xmake [23], a Lua-based cross-platform build system that generates build files to be invoked by native build systems such as *Make* and *Ninja*.

Integrating MicroPython's build system with the CHERIoT-RTOS build systems involved substantial re-engineering work. Our pragmatic solution was to invoke CHERIoT's *XMake* build system from the MicroPython *Make* based build system after mapping the various VM components into compartments. The MicroPython build system compiles the sources of MicroPython to object files and maps them to their designated compartments. A declarative *xmake.lua* build script with directives to link MicroPython object files along with CHERI-RTOS object files into a single software image is auto-generated from templates and invoked from the MicroPython build system.

The schematic in Figure 2 illustrates this workflow. Note that there are two separate processes: the FPGA softcore deployment (on the right hand side, which is unmodified) and the CHERIoT-RTOS and MicroPython deployment, (on the left hand side, as described above). There are some limitations with this approach, e.g. the build `CFLAGS` setting needs to be

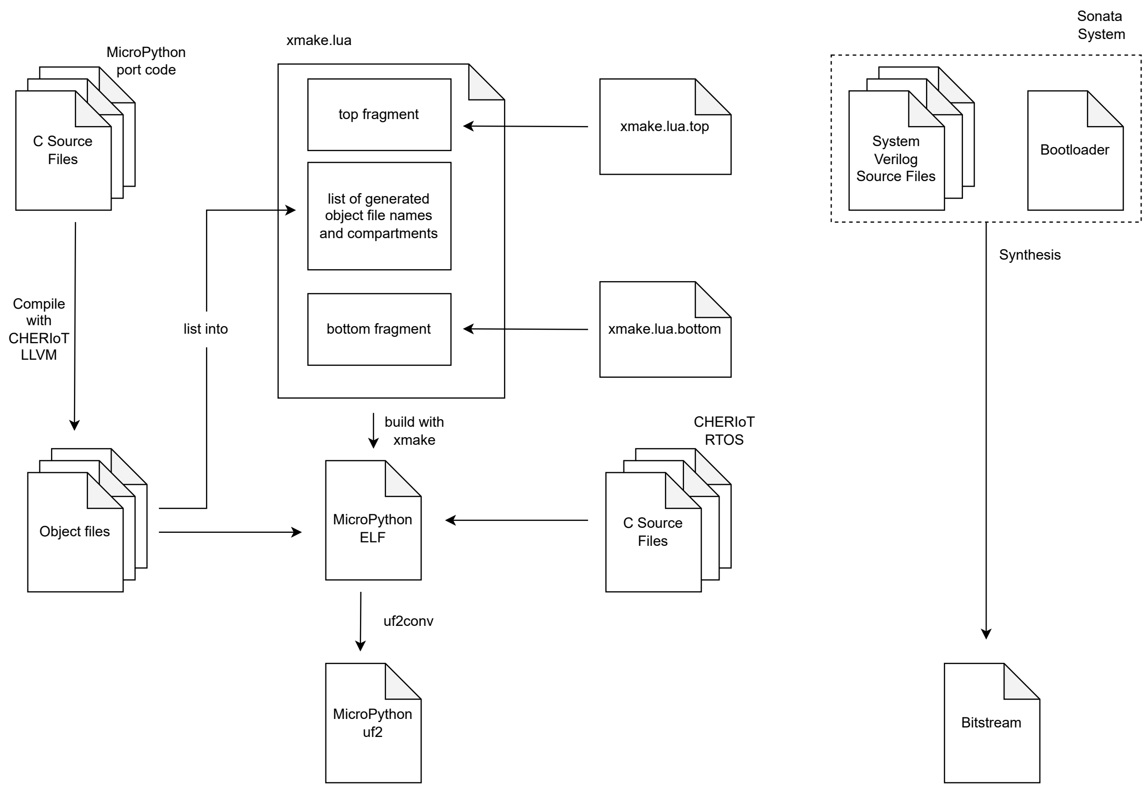

Figure 2. Build workflow for MicroPython on CHERIoT-RTOS. The left hand side shows MicroPython source code compilation and integration with the CHERIoT-RTOS build process. A software UF2 image which can be flashed to the pre-configured Sonata FPGA is generated. The right hand side shows how the Sonata/CHERIoT RTL is synthesized to an FPGA bitstream for deployment on the FPGA.

pulled manually from Xmake and the board descriptions and then injected into the Makefile.

There are a few other minor modifications to the build process. For instance, we need to add the CHERIoT header file `cheriintrin.h` to enable usage of CHERI intrinsics in our MicroPython codebase. We also need to add custom inline assembler blocks for MicroPython non-local returns (NLR). Finally, a C++ 'interface' file is required to call RTOS APIs from our C codebase.

4.2 Global/Local Memory Segregation

CHERI capabilities without `global` permissions are called `local` capabilities. These are written into memory through a capability that holds the `store_local` permission. CHERIoT uses the `store_local` permission to write to stacks and the register-save area. CHERIoT enforces a strict `global` ⊕ `store_local` permission invariant on writes to different areas of memory. MicroPython's NLR-based exception mechanism (cf. setjmp/longjmp(3) buffers on POSIX) are allocated on the stack because an NLR jump is only valid if it targets a buffer whose parent function has not yet returned. The NLR mechanism must faithfully store the stack pointer, which is a local capability, as well as two saved registers which may or may not contain local capabilities.

Active NLR buffers are stored as a linked list. Each NLR buffer also holds a pointer to the previous buffer which can be legally accessed via `local` capabilities as all of these are local. However, MicroPython stores the head of NLR buffer linked-list in a member of the `mp_state_thread_t` struct, which is declared as a global variable for the main thread. Stores using this variable cause an exception on CHERIoT due to the lack of a `store_local` permission for global variables.

Local capabilities may only be stored in registers or on the stack. Resolution of this issue is difficult due to the strict enforcement of the permissions invariant by CHERIoT hardware. One workaround at the level of CHERI-C source code would require every MicroPython C function that can take an exception to be refactored to accept thread-state context as an argument.

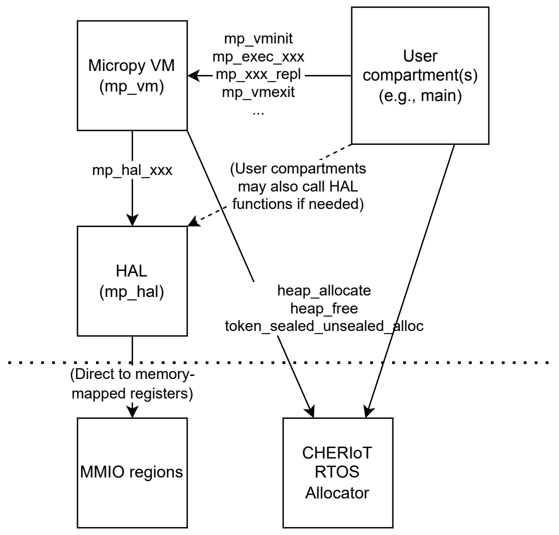
Micropy VM (mp_vm)
mp_vminit
mp_exec_xxx
mp_xxx_repl
mp_vmexit
...
User compartment(s) (e.g., main)
mp_hal_xxx
(User compartments may also call HAL functions if needed)
HAL (mp_hal)
heap_allocate
heap_free
token_sealed_unsealed_alloc
(Direct to memory-mapped registers)
MMIO regions
CHERIoT RTOS Allocator

Figure 3. Relationship between compartments. Arrows represent cross-compartment function calls mediated by the CHERIoT-RTOS compartment switcher (not shown). Everything above the dotted line is in 'userspace', below the line is the RTOS itself.

Instead, we take advantage of the CHERIoT compiler leaving register `c4` (aka `ctp`) untouched. In the standard RISC-V ABI this is the thread-locals pointer. However, thread-local variables are not supported by CHERIoT-RTOS. On entry into the MicroPython VM compartment, a structure of type `mp_state_thread_t` is allocated on the stack. A pointer to this struct is stored in `ctp`. As this register is otherwise unused, it remains accessible throughout MicroPython execution. However, this register is zeroed by CHERIoT-RTOS on return from any cross-compartment call. Correct usage of this register would require its contents to be spilled to the stack before any calls out of the VM compartment. This is achieved with preprocessor macros resolving to inline assembly.

There were a few other places where the global/store local restrictions required refactoring. However, in these cases it was possible to simply allocate the objects on the heap or declare them as globals.

4.3 Compartmentalization

One of the key principles of CHERIoT is *compartmentalization.* As isolation between compartments is strongly enforced by the CHERI hardware, the compartmentalization scheme is an important design consideration. Division of various VM components into compartments is complicated because MicroPython was designed to be a tightly coupled monolithic VM. For example, every Python object (with the exception of 'unboxed' objects like small integers) contains a pointer to a 'type' object. These Python 'type' objects themselves contain several function pointers which are invoked by the core VM loop to perform various operations on the object.

Figure 3 shows the current compartmentalized structure of CHERIoT MicroPython. There are five compartments shown, with compartment execution and switching managed by the CHERIoT-RTOS compartment switcher. External C libraries can be isolated as distinct userspace compartments.

To prevent invasive changes to the object structure, two options were considered. Applying the `cheri_ccallback` attribute to function pointers within the 'object' type would allow them to be called across compartments. However, this would result in a large number of compartment switches for all objects including core builtin types. In addition to the overhead, the `mp_state_thread_t` struct (Section 4.2) would not be accessible in the callee, preventing exceptions from being taken. The alternative design choice is to place the bulk of the VM in a single compartment. This would necessitate that Python objects are never accessible outside the VM compartment (although sealed handles for such objects may be passed in).

Congruently, VM components that do not require access to 'object' internals, such as the hardware access layer (HAL), are moved to a separate compartment. Ideally the VM compartment should always access drivers and peripherals via the HAL layer and never directly.

The core VM compartment exposes various entry points to be called from other compartments. The API includes functions to initialise the MicroPython VM, execute Python code strings or frozen modules, execute Python code interactively over UART, or call a named Python function with arguments converted from C/C++ arguments and the subsequent return value converted back into a C/C++ object. Each of these entry points must set up the thread-state structure mentioned above.

Error handling is a major design consideration in the CHERI compartmentalization model. The design should allow a compartment to react to CHERI capability faults occurring within the compartment. In addition, the error handler should also be able to handle a 'forced unwind' due to a capability fault in a callee function in another compartment that fails to handle its own fault. To allow for flexible handling of such exceptions, they are converted to Python exceptions. This is done by setting the setting the frame `pcc` register to `nlr_jump()`. As this function never returns normally, the immediate state of the faulting function is not a concern.

If there are no active NLR buffers when an Python exception occurs, a forced unwind is appropriate. However, a forced unwind cannot be directly triggered except within the compartment error handler itself. (The `nlr_jump()` is *not* part of the error handler – it is installed by the error handler returning, or invoked directly when an exception is raised). The error handler can be intentionally triggered in a number of ways, (*e.g.* by executing the EBREAK instruction). In each case, the error handler needs to differentiate and trigger a forced-unwind instead of installing the nlr_jump() context in this case.

The garbage collector (GC) needs to be able to scan every thread stack in the VM compartment. On non-CHERI systems this is trivial in the single-threaded case and on multi-threaded systems involves signalling other threads to scan their stacks. While CHERIoT does provide a mechanism to interrupt other threads, it is limited to threads within the same compartment. In the presence of any calls *out* of the VM compartment, the GC can only be implemented in a design that uses a single thread in a compartment called from the VM compartment.

The compartmentalization model can be extended to a design where the VM is 'owned/called from' another compartment or one where multiple non-interfering VMs can run. To aid this, all VM global state (largely consisting of two structures, mp_state_vm_t and mp_state_mem_t) is bundled into a dynamically allocated VM context structure. The mp_vm_init() function creates this context structure and returns a sealed capability handle to it. All other entry points taking the context handle as an argument are internally unsealed and stored in the thread-state structure for use throughout the compartment. Each VM context is entirely separate, so it is not a violation of the single-threading constraint for two threads to be executing in the VM compartment concurrently when using different VM contexts.

4.4 Minor Bug Fixes

Our port to CHERIoT-RTOS exposed two latent bugs in the MicroPython codebase, which we were able to identify and fix.

1. The nlr_push function requires a return-twice attribute
2. The mp_quicksort function has a pointer which goes out-of-bounds, leading to undefined behaviour. This is caught by CHERIoT since the capability value becomes invalid when it is taken out-of-bounds.

4.5 Peripheral Support

Development boards running MicroPython generally support a wide range of low-level protocols, for interfacing with external hardware components. The Sonata board is particularly well-equipped—given its status as a demonstrator platform, it has a much richer set of connectors than normal.

We added support for GPIO pins, SPI, I^{2}C and UART, all exposed via standard MicroPython idioms, e.g. machine.Pin for GPIO, machine.SPI for SPI and machine.I2C for I^{2}C. In this way, we provide high-level facilities for interfacing with commodity third party sensors, common breakout boards, etc.

The key point to note is that, in CHERIoT, such devices are memory-mapped, and must be addressed by valid *capabilities* which are automatically synthesized from the declarative board specification JSON file which defines the physical memory map. Each I/O device block is allocated a 4 KiB range in the address space. The spatial memory safety property of CHERI means it is not possible to offset from one I/O block into another, to access a different I/O device. The bounded capabilities prevent this, in the C source code underlying the MicroPython machine interface.

5 Evaluation

In this section, we describe our multifaceted evaluation, which includes an assessment of the extent of source code modifications required to support a CHERIoT MicroPython port (Section 5.1), a qualitative consideration of real-world security involving CVE mitigation (Section 5.2), and a quantitative performance evaluation across a set of Python benchmarks (Section 5.3).

5.1 Code Modifications

Our port of MicroPython to CHERIoT-RTOS was based on the pre-existing Morello port [17, 18]. Many of our changes involve the addition of target-specific files for the Sonata board. A simple git diff reports 2455 additional and 190 deleted lines of code, relative to the Morello port.

The ports/cheriot-rtos directory contains the majority of changes, specifically 23 new files comprising 2134 new source lines of code (SLoC). This is a relatively small port for MicroPython. By way of comparison, the ESP32 port directory contains 17.5 kSLoC and the STM32 port contains 99.6 kSLoC. One key consideration for CHERIoT is that the binary object file for the interpreter along with memory allocated for heap space is constrained to fit within the 256 KiB internal SRAM, limiting the MicroPython libraries and features that can be supported.

5.2 Security Mitigation

To assess the security of MicroPython on CHERIoT-RTOS, we examine all public vulnerabilities disclosed for MicroPython (Table 1). We find two specific vulnerabilities are prevented by CHERI, CVE-2023-7158 [20] and CVE-2024-8948 [21]. Two others (CVE-2023-7152 and CVE-2024-8946) are not applicable since our minimal MicroPython build does not include the buggy libraries. Finally, CVE-2024-8947 involves a temporal safety bug; at present, we have not implemented temporal safety in the MicroPython heap.

CVE-2023-7158 [20], classified as 'critical', is a heap buffer overflow bug in MicroPython. A minimum working example of Python code to exploit this vulnerability is shown in Listing 4. The underlying issue is that the indices function expects an integer value but we pass in a floating-point value.

Table 1. Vulnerabilities mitigated by CHERIoT

CVE	*Bug*	*Severity*	*Mitigated?*
2024-8948	heap buffer overflow	medium	yes
2024-8947	use after free	medium	no
2024-8946	heap buffer overflow	high	n/a
2023-7158	heap buffer overflow	high	yes
2023-7152	use after free	medium	n/a

```
class A:
    def __getitem__(self, idx):
        return idx

print(A()[:].indices(.0))
```

Figure 4. Python code to exploit CVE-2023-7158

The `indices` function checks to see if the bitpattern it receives is an unboxed small integer (i.e. an integer encoded directly in a pointer field). If not, it *assumes* the value is a boxed integer and dereferences relevant fields in the boxed integer object type to calculate the integer value. When the object is a float, it is only 8 bytes long (4 bytes for the type tag capability and another 4 bytes for the single-precision float bitpattern). However when we treat it as a boxed integer, we read values beyond the end of the 8 bytes, since the boxed integer data structure is longer. Reading past the end of the allocated object results in a heap buffer overflow.

In CHERIoT MicroPython, this type confusion problem is indirectly mitigated by the hardware bounds checking. floating-point object capabilities `*MP_OBJ_FLOAT` have a hard-coded length of 8 bytes, so any dereference of a value beyond this upper limit will result in a capability protection error, causing Python interpretation to halt.

A MicroPython patch has since been issued to address CVE-2023-7158 [8]. The patch involves an explicit type check to ensure the parameter supplied to the `indices` function is of the appropriate integer type. We observe that the CHERIoT-RTOS version of MicroPython, even without this mitigation, did not suffer from heap buffer overflow problems due to the hardware bounds checking.

CVE-2023-8948 [21] is a heap buffer underflow bug. Listing 5 shows a minimal working example to trigger the exploit. When the “`big`” integer is converted back to bytes, Python should raise an `OverflowError` exception. However, executing the `to_bytes()` method on the integer caused a heap buffer underflow and trampled memory.

The underlying cause is a bounds check for buffer underflow, which erroneously uses a “$(len < 0)$” guard condition to raise an exception. When the argument passed to the method is *0*, this guard condition is bypassed and control eventually passes to a loop copying data starting at the end

```
b = bytes(range(20))
ib = int.from_bytes(b, "big")
print(ib.to_bytes( 0, "big"))
```

Figure 5. Python code to exploit CVE-2023-8948

of the buffer to the beginning. The loop terminates when the pointer used to copy is equal to the buffer start. In the vulnerable code, the terminating condition used a pre-decrement operator `*(--ptr)`. When the initial length is zero, the first pointer underflows causing the exploit.

CHERIoT mitigates against the underflow attack by protecting against memory dereferences through an out-of-bounds capability, before any data is leaked. A fix for CVE-2023-8948 has now been merged into MicroPython [7].

5.3 Performance Characterization

The CHERIoT core implements the CHERI extensions on a RISC-V `RV32E` [27] profile. `RV32E` uses only 16 registers instead of the 32 registers used by the `RV32I` instruction set. As the floating-point registers typically used in an `RV32I` CPU have been removed, a *soft-float* calling convention is required. Soft-float support is added into the image in the MicroPython VM. The performance evaluation is performed on the Sonata board version 0.4. The CPU core is configured to execute at 30 MHz on a Xilinx ARTIX 7^{TM} XC7A50T FPGA [3].

To understand the overheads incurred by CHERIoT MicroPython, we compare its performance against a baseline single threaded MicroPython interpreter ported to execute on a single core RISC-V (`RV32E`) processor. To ensure that the hardware is closely matched, the CHERI features are disabled in the CHERIoT RTL before the FPGA bitstream is loaded onto the FPGA rather than using a different RISCV R32E implementation. This has the effect of executing `loads`, `stores` and pointer arithmetic using 32-bit integer pointers in the baseline `RV32E` platform while the CHERIoT-MicroPython interpreter uses 64-bit capabilities for the same. The ICache and SRAM sizes in both instances are maintained at 4 KiB and 256 KiB respectively.

In the case of the baseline `RV32E` MicroPython, the interpreter loop is executed on bare metal, without the support of any underlying RTOS. The evaluation utilizes benchmarks that do not require support from OS-level IO or system calls. Since both the baseline and CHERIoT MicroPython interpreters are single-threaded, any performance overhead measured would be the upper bound of the differences between the two instances.

The MicroPython interpreter is compiled with a heap size of 64 KiB. This was the maximum heap size that could be provided after the RTOS, MicroPython interpreter core and basic MicroPython modules required were added to the image.

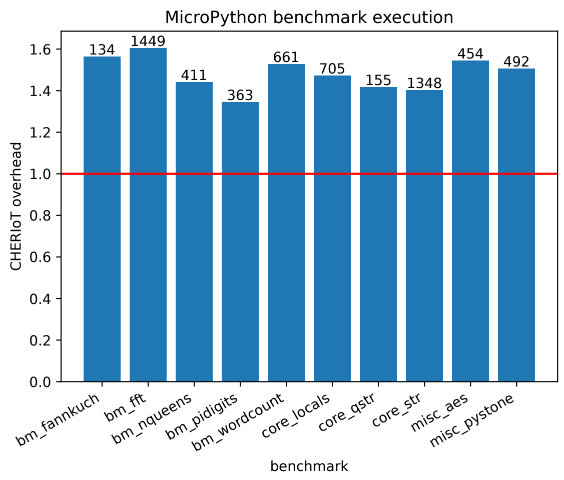

Figure 6. Overheads of CHERI execution for MicroPython benchmarks. Baseline (non-CHERI) interpreter is depicted as 1.0x. Arithmetic mean absolute execution time for each benchmark (in ms) is shown at the top of each bar.

The largest parameters for each benchmark that would terminate and finish execution given the constraint of the 64 KiB heap size was chosen. Benchmark results are the arithmetic mean of 20 iterations. Each iteration was performed after re-booting the Sonata board to prevent any anomalies from caching or string interning.

Execution time for each benchmark is measured from the start of execution of compiled bytecode. The time taken by the MicroPython compiler to compile code to bytecode is excluded from the measurements. The minimal time spent in garbage collection (GC) during execution of the actual benchmark is included. Time spent during a final GC phase executed by the REPL after benchmark completion is excluded. We take 20 measurements for each benchmark and record the arithmetic mean. Variance is minimal, since there is limited hardware noise or OS interaction. We report relative performance as the ratio of the arithmetic means for each benchmark, on the CHERI and non-CHERI configurations of the Sonata board.

The MicroPython benchmark suite [9] is a mixture of string manipulation programs along with integer and floating-point numerical computation benchmarks. Among the programs in the suite, `fannkuch`, `nqueens`, `pidigits`, `aes` and `pystone` are integer computation benchmarks while `bm_fft` is a floating-point computation benchmark. The `core_qstr`, `core_str`, `core_locals` and `wordcount` benchmarks involve string manipulation and search.

Figure 6 shows the relative performance of MicroPython on CHERIoT compared to the baseline RV32E baseline. Measured slowdowns in CHERIoT MicroPython range from 1.35x (`pidigits`) to 1.61x (`fft`). The geometric mean slowdown across all benchmarks compared to the baseline is 1.48x.

Benchmarks for CHERIoT MicroPython were executed with software revocation enabled for temporal safety of C/C++ allocations. We also ran the benchmarks without revocation but there was minimal observable difference. The worst case was 14 µs (0.004%) with seven out of ten benchmarks showing a difference of less that 1 µs (the resolution of our measurements). Software revocation is not a major performance factor for the MicroPython VM since it allocates a single large chunk of memory for the heap, which is then managed internally by the MicroPython runtime allocator.

These performance benchmark results should be interpreted while keeping in mind that CHERI hardware platforms in general the CHERIoT platform in particular are experimental hardware. They are under active development and many optimizations are being implemented to increase performance. Lowther et al. [17, 18] report similar normalised benchmark overheads of 1.5x–2x on the MicroPython *unix* port executing on the ARM based Morello [10] platform which is a desktop class processor. A separate study by Bramley et al. [4] compiled C/C++ benchmarks targetting memory allocator performance on the ARM Morello platform also report mean overheads of 1.56x – 1.61x.

Watson et al. [29] report certain CHERI hardware related overheads and attribute these to factors such as conservative branch prediction required because of unavailable bounds information of the PC. Their experiments point to a best estimate of 1.8% – 3.0% overheads in an optimized design, which is an order of magnitude lower than our measured overheads for CHERIoT MicroPython.

6 Platform Reflections

In this section, we consider the maturity of the CHERIoT platform in terms of its toolchain. We also compare CHERIoT to the other high TRL CHERI system, i.e. Arm Morello.

6.1 Ecosystem Maturity

The CHERIoT-RTOS/Sonata system consists of multiple components in varying states of maturity, all under active development throughout the duration of our work. These include the CHERIoT Clang/LLVM C/C++ compiler, the several components of the CHERIoT-RTOS, and the firmware for the Sonata board itself.

The compiler is a fork of the CTSRD CHERI LLVM fork used for Morello. Aside from the obvious backend changes to handle the CHERIoT ISA and ABI, there seem to be two significant new features. The first is the addition of a set of attributes to annotate functions which may be the targets of direct cross-compartment calls (`cheri_compartment`), indirect cross-compartment calls (`cheri_ccallback`), and calls out of a compartment to a library (`cheri_libcall`). Any function which does *not* bear one of these attributes is only callable from the compartment in which it is defined. The second is a series of changes to the linker to suit the CHERIoT-RTOS compartmentalization model. In our experience, the compiler seems to be relatively stable, despite the lack of an official release schedule. On the one occasion we encountered a bug, it was fixed within a day of our bug report.

The CHERIoT-RTOS itself consists of an Xmake-based build system, C and C++ header files (no headers shipped with Clang are used), core RTOS components (allocator, switcher, loader, and scheduler), and libraries.

The header files and libraries are minimal: the supported subsets of the C and C++ standard libraries focus on only the 'core' features expected to be used in an embedded setting. For instance, there is no floating-point support or `math.h`, and C++ exceptions are not supported either. It is also notable that the macros defined in the CHERIoT `cheri.h` header is completely incompatible with the header of the same name on previous CHERI systems. Macros wrapping CHERI Clang builtins are instead defined in `cheri-builtins.h`, and the names do not correspond with the standard CHERI names for the same operations (e.g. `cgetlen()` instead of `cheri_length_get()`). It is unclear why the CHERIoT developers changed the macros wrapping these builtins.

When we started the CHERIoT-RTOS/Sonata port of MicroPython, the Sonata board had just been released with firmware v0.1. The firmware was (as expected) very incomplete. Since then, several newer versions have been released. All evaluation in this paper is done on firmware version v0.4.1. The firmware (and the Sonata-specific CHERIoT-RTOS components) have certainly seen the most changes of any component over the span of our project. It is not clear that they could be considered stable. However, as of publication of this paper, v1.0 has been released which is labelled as stable in the release notes. Breaking changes to the Sonata system that we have had to adapt to include a change to loadable binary format used by the bootloader (raw binary-in-UF2 to ELF-in-UF2) and changes to the MMIO layout.

In general, the CHERI ecosystem is fragmented as it has evolved over the last decade. MIPS, Arm and RISC-V variants of CHERI are all available, often with subtly different feature sets and header files. This fragmentation can be confusing for developers coming into the community, but is expected in a quickly improving project.

CHERIoT is much more narrowly focused as there is a single implementation with less time for legacy material to accumulate [1, 2]. The documentation is up-to-date, appropriately detailed, and user-friendly, e.g. see the CHERIoT Programmers' Guide [5].

6.2 Comparison with Morello

The CHERIoT platform is highly constrained compared to Morello. This is to be expected, given CHERIoT is an embedded platform whereas Morello is server-class. Despite this, there is an obvious conceptual coherence and elegance to CHERIoT, in terms of the design decisions and the retention of the core CHERI principles.

The most striking thing when comparing CHERIoT-RTOS on Sonata to CheriBSD on Morello are the constraints imposed both at the OS and architectural levels. At the architectural level, most of these constraints are a direct result of the limited space for capability metadata (32 bits rather than 64). The compression of capability permissions has not affected our work with MicroPython. However, the reduction in bounds mantissa bits leads to coarser capability bounds compared to Morello. Validity tag clearance when pointer arithmetic goes out-of-bounds is eager, whereas on Morello validity tag clearance occurs when dereferencing the out-of-bounds capability. This relaxation better fits with loop computation idioms. More striking is the three-bit otype field and consequent reservation of 'real' sealing capabilities to the RTOS. User compartments request 'virtual' sealing capabilities (from the RTOS) to allocate a buffer with a sealed 'token' to it, or to unseal a 'token'. This differs from CheriBSD/Morello where user code can request a sealing capability and then use the architectural seal/unseal instructions directly.

The IoT domain is ideal for CHERI-fication [6] due to the additional security vulnerabilities that occur in such resource constrained bare-metal software execution environments. The user-friendly ecosystem and low-cost development boards suggest that CHERIoT may have a greater deployment footprint than the desktop or server class Morello platform.

7 Related Work

In a recent and comprehensive taxonomy of hardware security presented by Zhao et al. [35], the features provided by the CHERIoT system are *extensions* that support *runtime protection*.

In terms of porting to CHERI, the *MicroPython* interpreter was originally adapted for the Morello/CheriBSD platform [17, 18]. In our work, we have modified this CHERI port to deal with the 32-bit RISC-V embedded architecture on which CHERIoT is based—somewhat different to the 64-bit server-class Arm architecture of Morello. Further, we have integrated our code fully with the compartment model of CHERIoT-RTOS, whereas the Unix process-based Morello MicroPython interpreter executes in a single, process-wide compartment by default.

The *Microvium* JavaScript interpreter [12] has been ported to CHERIoT [2]. Microvium supports a cut-down subset of JavaScript, with heap snapshots for partial evaluation. Unlike our MicroPython work, the Microvium port required no source code changes in the interpreter codebase, since the runtime engine is implemented in an entirely platform-neutral dialect of C [22]. A single representative IoT benchmark application is characterized on CHERIoT, controlling an on-board LED via MQTT. No large scale performance benchmarking of JavaScript applications is reported.

Gutstein [11] evaluates the feasibility of porting a language runtime to CHERI, with a case study of the *JavaScript-Core* engine. Relevant changes, which we also needed to consider for CHERIoT MicroPython, include (i) the need to extend the core `JSValue` runtime object representation to accommodate double word capabilities, and (ii) changes to the C/C++ codebase to handle the change in the size and semantics of capability pointers. Gutstein considers server-class CHERI platforms but presents no performance results due to lack of hardware availability.

MSWasm [19] is an extension to WebAssembly (Wasm) that enforces runtime memory safety within the linear memory region. Given Wasm is often presented as appropriate for IoT deployment, MSWasm seems to be a reasonable comparison point with MicroPython on CHERIoT. The authors report geometric mean overheads of around 50% for executing PolyBenchC with spatial and temporal memory safety, for ahead-of-time compiled Wasm code, relative to a baseline with no runtime safety checks. Again, this is roughly in line with our overhead figures for MicroPython execution on CHERIoT.

There is much active research on compartmentalization for CHERI, including the *FlexOS* work from Manchester [14] and the *Cap-VMs* work from Imperial [24]. Both these systems are evaluated with real-world applications on server-class Morello hardware. They show the performance advantages of fine-grained lightweight compartmentalization supported natively in the CPU, which avoids paying a 'memory management unit tax' for isolation. In contrast, we evaluate compartments in the embedded domain. Here, where direct physical addressing is common, compartmentalization mechanisms are essential for secure runtime isolation.

Ullah and Rashid [25] present a systematic analysis of transitioning legacy systems software to Morello, emphasizing a need for 'heightened developer vigilance' since many vulnerabilities may be introduced or preserved when porting C code to CHERI platforms.

Watson et al. [30] describe a three-month project to port a major open-source codebase to Morello, involving a large user-centric desktop system. However their work explicitly excludes language runtimes, which (they assert) would 'require both non-trivial changes, and also would benefit from greater understanding as to how CHERI can improve their robustness'. Our work shows the necessary changes are significant but achievable, and that CHERI does enhance runtime system resilience.

8 Conclusion

In summary, the MicroPython port from Morello to CHERIoT is relatively straightforward from a technical perspective. Our open-source fork is available via github[1]. While challenges arise in the specific details of porting, the groundwork laid by the existing Morello CHERI implementation eases the transition to CHERIoT. This adaptability suggests that CHERI enhancements could be helpful for porting systems software to a range of future memory safety mechanisms and platforms, i.e. the first port to a memory-safe platform makes subsequent memory-safe ports much easier.

We now revisit the three research questions posed in Section 1, based on the experiences of our case study.

- **RQ1:** We confirm the CHERIoT-RTOS ecosystem offers a conducive environment for secure language runtime development. We can state this with some confidence since there are now two open-source embedded language runtimes (Microvium and MicroPython) that run on CHERIoT.
- **RQ2:** Our work shows that the enhanced security features of CHERIoT-RTOS improve the overall security of the MicroPython interpreter. For instance, we have demonstrated the mitigation of two heap buffer overflow CVEs through spatial memory safety. Additionally, CHERIoT's compartmentalization model prevents unrestricted access to the microcontroller's physical address space.
- **RQ3:** While Morello currently has a broader deployment base (approximately 1000 machines), the Sonata platform is poised for rapid adoption since it is available as open-source hardware that runs on FPGAs. Its streamlined design, sensible documentation and reasonable toolchain are highly developer-friendly. Further, the IoT domain represents a promising route to market for CHERI [6]. We anticipate that MicroPython on CHERIoT will play a significant role in this expansion, given the scripting framework's widespread use in IoT projects.

The memory safety and compartmentalization features of CHERIoT bring significant security benefits by reducing the attack surface, exemplifying a *secure-by-design* approach to virtual machine engineering. This work reinforces CHERI's potential for successful adoption in secure IoT solutions.

Acknowledgments

This work was funded by UKRI's Digital Security by Design (DSbD) programme (including EPSRC grants EP/V000349/1 and EP/X015831/1), and also through a UK Defence and Security Accelerator contract ACC6037520.

References

[1] Saar Amar, Tony Chen, David Chisnall, Felix Domke, Nathaniel Filardo, Kunyan Liu, Robert Norton-Wright, Yucong Tao, Robert NM Watson, and Hongyan Xia. 2023. *CHERIoT: Rethinking security for low-cost embedded systems.* Technical Report MSR-TR-2023-6. Microsoft.

[1] https://github.com/glasgowPLI/micropython

[2] Saar Amar, David Chisnall, Tony Chen, Nathaniel Wesley Filardo, Ben Laurie, Kunyan Liu, Robert Norton, Simon W. Moore, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. 2023. CHERIoT: Complete Memory Safety for Embedded Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23)*. 641–653. doi:10.1145/3613424.3614266

[3] AMD. 2024. AMD Artix^{TM} 7 FPGAs. https://www.amd.com/en/products/adaptive-socs-and-fpgas/fpga/artix-7.html#product-table.

[4] Jacob Bramley, Dejice Jacob, Andrei Lascu, Jeremy Singer, and Laurence Tratt. 2023. Picking a CHERI Allocator: Security and Performance Considerations. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management* (Orlando, FL, USA) *(ISMM 2023)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3591195.3595278

[5] David Chisnall. 2024. CHERIoT Programmers' Guide. https://cheriot.org/book/.

[6] Dept for Science, Innovation and Technology. 2024. CHERI adoption and diffusion research. https://www.gov.uk/government/publications/cheri-adoption-and-diffusion-research/cheri-adoption-and-diffusion-research.

[7] Damien P. George. 2024. py/objint: Fix int.to_bytes() buffer size checks. https://github.com/micropython/micropython/pull/13087.

[8] Damien P. George. 2024. Validate that the argument to indices() is an integer. https://github.com/micropython/micropython/pull/13039.

[9] Damien P. George et al. 2024. The MicroPython benchmark suite. https://github.com/micropython/micropython/tree/master/tests/perf_bench.

[10] Richard Grisenthwaite, Graeme Barnes, Robert NM Watson, Simon W Moore, Peter Sewell, and Jonathan Woodruff. 2023. The Arm Morello evaluation platform—validating CHERI-based security in a high-performance system. *IEEE Micro* 43, 3 (2023), 50–57. doi:10.1109/MM.2023.3264676

[11] Brett Gutstein. 2022. *Memory safety with CHERI capabilities: security analysis, language interpreters, and heap temporal safety.* Technical Report UCAM-CL-TR-975. University of Cambridge, Computer Laboratory. doi:10.48456/tr-975

[12] Michael Hunter. 2020. Microvium. https://github.com/coder-mike/microvium/.

[13] Shashi Jayakumar. 2020. Cyber Attacks by Terrorists and other Malevolent Actors: Prevention and Preparedness With Three Case Studies on Estonia, Singapore, and the United States. In *Handbook of Terrorism Prevention and Preparedness*, Alex P. Schmid (Ed.). 871–930. https://www.icct.nl/sites/default/files/2023-01/Chapter-29-Handbook-.pdf.

[14] John Alistair Kressel, Hugo Lefeuvre, and Pierre Olivier. 2023. Software Compartmentalization Trade-Offs with Hardware Capabilities. In *Proceedings of the 12th Workshop on Programming Languages and Operating Systems*. 49–57. doi:10.1145/3623759.3624550

[15] LowRISC. 2024. Sonata Board Reference. https://lowrisc.github.io/sonata-system/doc/architecture/board.html.

[16] LowRISC. 2024. Sunburst Project. https://www.sunburst-project.org/.

[17] Duncan Lowther, Dejice Jacob, and Jeremy Singer. 2023. CHERI Performance Enhancement for a Bytecode Interpreter. In *Proceedings of the 15th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. 1–10. doi:10.1145/3623507.3623552

[18] Duncan Lowther, Dejice Jacob, and Jeremy Singer. 2023. Morello MicroPython: A Python Interpreter for CHERI. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. 62–69. doi:10.1145/3617651.3622991

[19] Alexandra E. Michael, Anitha Gollamudi, Jay Bosamiya, Evan Johnson, Aidan Denlinger, Craig Disselkoen, Conrad Watt, Bryan Parno, Marco Patrignani, Marco Vassena, and Deian Stefan. 2023. MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code. *Proc. ACM Programming Languages* 7, POPL, Article 15 (Jan. 2023), 30 pages. doi:10.1145/3571208

[20] NIST. 2023. CVE-2023-7158. https://nvd.nist.gov/vuln/detail/CVE-2023-7158.

[21] NIST. 2024. CVE-2024-8948. https://nvd.nist.gov/vuln/detail/CVE-2024-8948.

[22] Robert Norton-Wright. 2023. CHERIoT. https://www.dcs.gla.ac.uk/~jsinger/cheritech23_slides/rnortonwright_cheritech.pdf.

[23] Ruqi. 2024. XMake: A cross-platform build utility based on Lua. https://github.com/xmake-io/xmake/.

[24] Vasily A. Sartakov, Lluís Vilanova, David Eyers, Takahiro Shinagawa, and Peter Pietzuch. 2022. CAP-VMs: Capability-Based Isolation and Sharing in the Cloud. In *16th USENIX Symposium on Operating Systems Design and Implementation*. 597–612. https://www.usenix.org/conference/osdi22/presentation/sartakov

[25] Sami Ullah and Awais Rashid. 2024. Porting to Morello: An In-depth Study on Compiler Behaviors, CERT Guideline Violations, and Security Implications. In *IEEE 9th European Symposium on Security and Privacy*. 381–397. doi:10.1109/EuroSP60621.2024.00028

[26] Kui Wang, Dmitry Kasatkin, Vincent Ahlrichs, Lukas Auer, Konrad Hohentanner, Julian Horsch, and Jan-Erik Ekberg. 2024. Cherifying Linux: A Practical View on using CHERI. In *Proceedings of the 17th European Workshop on Systems Security*. 15–21. doi:10.1145/3642974.3652282

[27] Waterman, Andrew and Asanović, Krste. 2017. The RISC-V instruction set manual Volume I:User-Level ISA. https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.

[28] Robert N. M. Watson, David Chisnall, Jessica Clarke, Brooks Davis, Nathaniel Wesley Filardo, Ben Laurie, Simon W. Moore, Peter G. Neumann, Alexander Richardson, Peter Sewell, Konrad Witaszczyk, and Jonathan Woodruff. 2024. CHERI: Hardware-Enabled C/C++ Memory Protection at Scale. *IEEE Security & Privacy* 22, 4 (2024), 50–61. doi:10.1109/MSEC.2024.3396701

[29] Robert N. M. Watson, Jessica Clarke, Peter Sewell, Jonathan Woodruff, Simon W. Moore, Graeme Barnes, Richard Grisenthwaite, Kathryn Stacer, Silviu Baranga, and Alexander Richardson. 2023. *Early performance results from the prototype Morello microarchitecture.* Technical Report UCAM-CL-TR-986. University of Cambridge, Computer Laboratory. doi:10.48456/tr-986

[30] Robert N. M. Watson, Ben Laurie, and Alex Richardson. 2021. Assessing the Viability of an Open-Source CHERI Desktop Software Ecosystem. https://www.capabilitieslimited.co.uk/_files/ugd/f4d681_e0f23245dace466297f20a0dbd22d371.pdf

[31] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Franz A. Fuchs, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. 2023. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9).* Technical Report UCAM-CL-TR-987. University of Cambridge, Computer Laboratory. doi:10.48456/tr-987

[32] White House. 2024. Back to the Building Blocks: A path toward secure and measurable software. https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf.

[33] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert M Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W Filardo, A Theodore Markettos, et al. 2019. Cheri concentrate: Practical compressed capabilities. *IEEE Trans. Comput.* 68, 10 (2019), 1455–1469. doi:10.1109/TC.2019.2914037

[34] 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. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceedings of the 41st*

Annual International Symposium on Computer Architecture. 457–468. doi:10.1145/2678373.2665740

[35] Lianying Zhao, He Shuang, Shengjie Xu, Wei Huang, Rongzhen Cui, Pushkar Bettadpur, and David Lie. 2024. A Survey of Hardware Improvements to Secure Program Execution. *Comput. Surveys* 56, 12, Article 311 (Oct. 2024), 37 pages. doi:10.1145/3672392

Received 2024-11-13; accepted 2024-12-21