CheriRTOS: A Capability Model for Embedded Devices

Hongyan Xia*, Jonathan Woodruff*, Hadrien Barral*, Lawrence Esswood*, Alexandre Joannou*, Robert Kovacsics*, David Chisnall*, Michael Roe*, Brooks Davis†, Edward Napierala*, John Baldwin†, Khilan Gudka*, Peter G. Neumann†, Alex Richardson*, Simon W. Moore*, Robert N. M. Watson*

*Computer Laboratory, University of Cambridge, Cambridge, UK †SRI International, Menlo Park, CA, USA
Website: www.cl.cam.ac.uk/research/comparch Website: www.csl.sri.com

***Abstract*—Embedded systems are deployed ubiquitously among various sectors including automotive, medical, robotics and avionics. As these devices become increasingly connected, the attack surface also increases tremendously; new mechanisms must be deployed to defend against more sophisticated attacks while not violating resource constraints. In this paper we present CheriRTOS on CHERI-64, a hardware-software platform atop Capability Hardware Enhanced RISC Instructions (CHERI) for embedded systems.**

Our system provides efficient and scalable task isolation, fast and secure inter-task communication, fine-grained memory safety, and real-time guarantees, using hardware capabilities as the sole protection mechanism. We summarize state-of-the-art security and memory safety for embedded systems for comparison with our platform, illustrating the superior substrate provided by CHERI's capabilities. Finally, our evaluations show that a capability system can be implemented within the constraints of embedded systems.

I. Introduction

Embedded processors are prevalent today in consumer products (such as disk controllers, smart watches, and WiFi chips) and security-critical applications (such as self-driving vehicles, medical instruments and aviation). With larger deployments came increased connectivity, invalidating previous assumptions that embedded systems were isolated and primarily subject to physical attacks. Even though many vulnerabilities have already been disclosed by attackers and researchers [1] [2] [3] [4], it still remains a challenge to provide a comprehensive security framework within the highly constrained hardware-software budgets of embedded systems.

Unfortunately, many security mechanisms target large-scale systems and rarely scale down to low-cost, real-time and deterministic usage models, or else are too limited and too static to provide a scalable and flexible solution. Other state-of-the-art approaches aim to establish security frameworks for embedded environments, but still face many issues with the growth of embedded systems. To address these problems, we present and evaluate CheriRTOS on CHERI-64, a hardware-software platform for embedded systems that is based solely on the CHERI protection model.

The contributions of this paper include:

- CHERI-64, an implementation of 64-bit compressed capabilities for embedded systems.
- A CHERI-aware real-time operating system, CheriRTOS. In addition to the common properties of real-time embedded systems, we implement fine-grained memory protection and secure centralized heap management, dynamic task loading, low-latency and direct domain crossing, distributed trusted stacks to protect return contexts, and secure peripherals.
- Evaluation of the overhead, latency and determinism of the implementation.
- Summary of state-of-the-art memory safety approaches for embedded systems.

The CSC Cambridge Scholarship for the first author is gratefully acknowledged. This work is also part of the CTSRD project sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government. We also acknowledge the EPSRC REMS Programme Grant [EP/K008528/1], the EPSRC Impact Acceleration Account [EP/K503757/1], an ARM iCASE award and Google, Inc.

II. Background

A. State-of-the-art

The constraints of embedded devices directly forbid the use of complex, high-cost, high-latency, and non-deterministic security solutions. As a result, many security components have been implemented specifically for embedded systems.

The Memory Protection Unit (MPU) is commonly adopted in embedded processors to mark memory regions with security attributes [5] to prevent arbitrary physical memory access. Although widely implemented, MPUs have several inherent drawbacks. First, an MPU is implemented as a kernel space device, and each register takes multiple cycles to configure. As a result, they are normally configured only globally at system start-up, which makes per-task memory access control difficult; user space cannot leverage it for intra-task protection. Second, MPU entries are limited. With only around 8 MPU regions in most implementations, only security-critical memory partitions are protected, e.g., the kernel, encryption keys, code sections, etc.; thus, any fine-grained memory protection is impractical. Third, MPU lookups involve associative searches of all entries. Each cycle can potentially require up to 32 (assuming 8 MPU entries, accounting for both instruction fetch and data) comparisons. This means that MPUs are inherently inefficient in terms of power and die area. The RISC-V Physical Memory Protection (PMP) unit is a state-

of-the art MPU specification [6], which we have implemented for comparisons in this paper.

In addition to MPUs that generally provide system-wide memory protection for critical regions, TrustZone® from ARM partitions memory into secure and non-secure worlds [7] with constrained control flow. Non-secure code can jump only to valid entry points on the secure side, and secure code calls non-secure functions after clearing registers and pushing the return address on the secure stack. This constrained entry-point design guarantees a certain degree of domain isolation with Control Flow Integrity (CFI), which avoids Return-Oriented Programming (ROP) attacks. Nevertheless, TrustZone is likely to encounter scalability issues. For example, further isolation within a world is not possible, still enabling attacks in the same world. Separating the entire system into only two worlds may not be a wise design choice.

TrustLite [8] and TyTAN [9] extend the kernel and build the Execution-Aware MPU (EA-MPU), which links data and code entries to distinguish between different tasks. The aim is to control memory access on a per-task basis. However, an MPU-based approach inherits several flaws as described above, restricting the number of simultaneous tasks and inter-task memory sharing. For CFI and defense against ROP attacks, several schemes for embedded systems [10] [11] use dedicated instructions for function calls, exposing only valid entry points, hiding return addresses in protected spaces, etc. On the other hand, Sanctus [12] [13] builds tasks into Self-Protecting Modules (SPMs) to restrict access and enforce control flow. However, it sacrifices software flexibility and incurs a high hardware cost by implementing SPM loading, measurement, and runtime identification in the trusted CPU.

Research projects also tackle the memory protection problem on a programming-language level for embedded devices. nesCheck [14] modifies the language and compiler to perform stronger type safety, static analysis, and run-time checks. Others develop new compiler frameworks to address stack, array and pointer safety, and implement new heap allocation techniques [15]. However, these approaches protect only code written in specific language variants or compiled with the modified toolchain. A malicious task may directly execute or inject low-level code that circumvents any language-level invariants. Similar to many software schemes for desktop systems, this type of memory protection is most useful for debugging existing codebases, and has weaker security guarantees.

Centralized heap management. In typical embedded systems with a flat address space, all dynamic memory allocations are commonly done in a centralized heap. However, without bounded access, allocations from a user can easily overflow into other allocations or even the heap metadata to attack other users or the heap allocator itself. Many allocators including jemalloc [16] organize the metadata in separate structures and use hashing to locate the corresponding metadata; however, embedded systems still prefer to have inlined metadata into each allocated chunk and use "free-lists" like dlmalloc [17] for lower latency. Such allocators demand low-latency and fine-grained memory protection for effective isolation, which unfortunately is difficult and inherently creates the tension between flexibility and security on current hardware-software platforms.

B. Capabilities and CHERI

A capability is an unforgeable token, which when presented can be taken as incontestable proof that the presenter is authorized to have access to the object named in the token [18]. Capability systems provid an architectural protection model, along with segmented memory, MMU protection, access control lists, etc.

CHERI (Capability Hardware Enhanced RISC Instructions) [19] [20] has a capability Instruction-Set Architecture (ISA) that enforces bounded memory access via capabilities. A memory capability extends the pointer with base, top and permission bits (read, write, execute, etc.) to control memory access, with a tag bit to enforce its integrity and unforgeability. The CHERI ISA can perform manipulations on capabilities only if they do not increase rights (e.g., they cannot reduce bounds, clear permissions, or create a tag bit). For compartmentalization and fast domain crossing, capabilities can be made immutable and non-dereferenceable ("sealed capabilities") by using another capability as a key, and the sealed capability is given an Object Type (otype) from the key. Sealed capabilities can be unsealed only by the key or by secure Capability Calls (CCalls). Calling into another domain requires a pair of otype-matched sealed code and data capabilities of the callee to enter the secure domain. Upon a CCall, the sealed pair is atomically unsealed and installed, transferring control to the new domain. The immutability and non-dereferenceability also guarantee that the caller cannot tamper with callee's state using sealed capabilities.

Existing research on CHERI [19] [21] [22] has shown that a capability system can complement traditional paged-memory in modern OSes. However, we believe the nature of CHERI also integrates well with embedded systems that have no MMU, limited memory, a single flat address space, low-latency requirements and real-time guarantees. We have implemented CHERI-64 (which supports a 32-bit flat address space with 64-bit compressed capabilities) and CheriRTOS (an RTOS kernel), illustrating that capabilities can be a generic and unified security interface that offers strong and scalable task isolation, fast domain transition, as well as fine-grained memory protection. In the meantime, the system does not violate a low-cost and low-latency profile, and provides novel solutions to many problems that are inherently challenging or impossible for conventional memory safety measures in embedded systems.

III. Requirements

From the survey of state-of-the-art solutions, the shortcomings of existing protection schemes, and the properties and growth of embedded systems themselves, we identify the following requirements that are essential to a secure design.

Task isolation. In a flat physical address space, it is essential to separate different tasks into different domains. Unconstrained access means a malicious task can easily compromise other critical components in the system.

Fine-grained memory protection. Existing embedded system memory protections mediate access only to large segments of code and data. Protecting finer granularities often requires explicit checks from the compiler or programmer at a cost of run-time slowdown [15] [14], and can be error-prone or incomplete [2]. An architecture should provide a generic mechanism for low-cost and fine-grained memory protection.

Low-latency and secure domain crossing and inter-task communication. A system with multitasking often requires communication among components, either in terms of message passing and memory sharing, or in cross-domain function calls. Strong task isolation should not prohibit efficient and secure domain crossing and communication.

Secure centralized heap management. Flexible heap allocation demands fine-grained memory protection, which then enables a secure shared-heap allocator by restricting any user of allocations from reaching metadata or any other allocations.

Real-time guarantees. No security architecture should violate real-time constraints. Cached memory translation and protection, for example, directly violate the low-latency and deterministic properties of embedded systems.

Scalability. The rapid growth of the embedded market demands scalable solutions. MPU-based approaches described above suffer from scalability issues and may not meet the needs as these systems become more capable and dynamic.

A generic solution for security. Embedded chips often implement multiple components to enforce safety. ARM solutions typically provide an MPU, a Security Attribution Unit, TrustZone®, and even an Implementation Defined Attribution Unit for security. However, orchestrating many security mechanisms is non-trivial, and we have seen that vendors often revert to manual assertions and explicit checks, leaving these security layers largely unused [2]. A single, generic solution would ease the effort of orchestration and deployment.

IV. Architecture and Implementation

A. Hardware and compiler

The CHERI CPU extends the BERI processor [23] (our baseline MIPS ISA) with capability extensions and a capability coprocessor. In CHERI-64, we implement 64-bit capabilities (Figure 1) in a flat 32-bit address space. We use an encoding similar to the capability compression algorithm described in [24] [25]. The format is shown in Figure 1. The coprocessor is responsible for executing capability instructions and mediating memory accesses via capabilities. We implement 8 capability registers in our coprocessor, corresponding to typically 8 entries in MPUs. The CHERI-64 CPU is synthesized on a Stratix IV FPGA running at 100MHz. Our processor implementation (including pipeline stages, memory organization, exception and interrupt delays) is very close to a commercial ARM Cortex-R5 implementation [26].

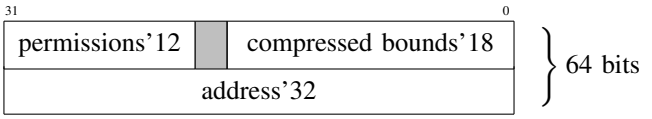

Fig. 1. Memory representation of an unsealed CHERI-64 capability.

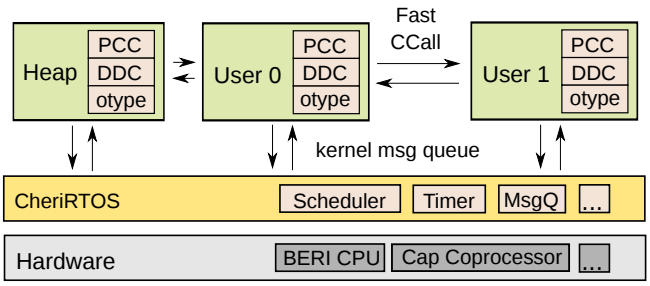

Fig. 2. Overall architecture

The compiler is the LLVM toolchain [27] with CHERI extensions. We modified the compiler to support the MIPS-n32 ABI with a 32-bit address space, using only eight 64-bit capability registers.

B. Operating System

To incorporate capabilities into the kernel itself, we implement CheriRTOS, a kernel that holds the common properties of an RTOS, including (1) multi-tasking support, (2) priority-based pre-emptive scheduling, (3) real-time guarantees, (4) message queues and direct inter-task communication, (5) cycle accurate timers, etc. Additionally, CHERI primitives are essential in implementing (6) fine-grained memory protection and secure centralized heap management, (7) dynamic task loading and unloading with non-Position Independent Code (non-PIC), (8) secure low-latency and direct domain crossing, (9) distributed trusted stacks to protect return contexts, and (10) secure peripherals.

OS structure. The structure of the CheriRTOS system is shown in Figure 2. At start-up, each user task is confined within a pair of code (Program Counter Capability, PCC) and data (Default Data Capability, DDC) capabilities, with PCC restricted to its code section and DDC restricted to data sections and the stack. The PCC and DDC pair defines the initial domain of a task. Under the CHERI architecture, the Program Counter (PC) is allowed to fetch only instructions within the bounds and permissions of PCC. By default, data loads and stores are restricted implicitly by the bounds and permissions of DDC. Any access outside the domain can be granted only by receiving additional capabilities and explicitly specifying the capability to use instead of using its own DDC (Figure 3).

The overall structure reflects the requirement of an isolated and scalable system. All tasks reside within individual domains and the architecture does not impose a limit on the number of domains created. Theoretically, CheriRTOS scales up to arbitrary numbers of tasks, and should be limited in practice only by memory and processing power.

```
lw  $t0, 8($s0)       # implicit, check against DDC
                      # load addr = DDC.base+$s0+8
clw $t0, $s0, 8($c1)  # explicit, check against $c1
                      # load addr = $c1.base+$s0+8
```

Fig. 3. Example of memory access instructions under CHERI. "$" denotes registers. Loading a word at address `$s0 + 8` (relative to the base of the capability) into `$t0`, either implicitly or via an explicit capability register.

For inter-task communication, the kernel provides buffered message queues, but direct Capability Calls (fast CCall) are supported for low-latency use cases as well. In addition, each task is given a capability as a key with a unique otype, so that it can seal and unseal capabilities with its own otype.

Dynamic task loading and unloading. The OS and the scheduler are able to dynamically load and unload tasks during run-time. Unlike many systems that require either Position Independent Code (PIC) or a binary loader to perform run-time loading at arbitrary addresses, memory accesses are offset by the base field of PCC or DDC; a task can simply be loaded or relocated by assigning a different pair of PCC and DDC without PIC or run-time relocation, significantly simplifying the binary loader.

Context switch. Capability registers have single-cycle access for most capability instructions. Context switches, exceptions, and interrupts will also store and load the capability register file (including PCC and DDC). This means a context switch also becomes a domain switch. Because capability registers can be stored or loaded like general purpose registers, we can efficiently maintain a capability context for each user task. As a result, we do not have to use the capability register file globally like MPU solutions, and we do not have an inherent limit on the number of tasks (and the number of protection regions for each task) that could be enabled simultaneously.

Fast and secure inter-task communication. CheriRTOS provides kernel message queues for regular and buffered communication, which takes around 1300 cycles ($13\mu s$) for a round trip. Meanwhile, the requirement in Section III motivates us to implement a secure and low-latency mechanism. As a result, fast CCall (introducing a `CCallFast` instruction) is used to bypass the kernel and the exception path to perform a direct cross-domain function call or message passing in user space with a round trip of around only 100 cycles. To do a fast CCall, a task needs a pair of sealed and type-matched PCC and DDC as a handle to another domain. Upon a successful `CCallFast` instruction, the callee's PCC and DDC are atomically unsealed and installed, transferring control to the new domain. Sealed capabilities can be possessed by other tasks because they are immutable and cannot be dereferenced; therefore, they do not expose memory to other tasks, and the pointer field in PCC is the only valid entry point, enforcing Control Flow Integrity (CFI) between domains.

Secure peripherals. Accesses to peripherals are commonly granted to only a few privileged tasks, mostly the kernel, to restrict and arbitrate the usage from untrusted user tasks through

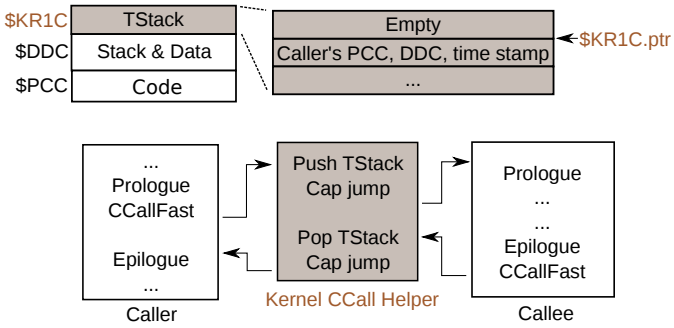

Fig. 4. Trusted stack and fast CCall round trip. Dark indicates kernel-only objects. The pointer field of $KR1C points to the top of the trusted stack.

an interface like system calls. However, with capabilities we are able to take a de-centralized approach. As peripherals are commonly memory-mapped, we simply separate their accesses from the kernel into user tasks. For example, the UART module is just a task possessing a capability to the UART memory region. Whether another user task is able to access the UART is determined by whether it has received the sealed capability pair to CCall into the UART task. There are two major advantages of this design. First, direct CCalls achieve lower latencies than kernel system calls. Second, the kernel becomes smaller, further reducing the attack interface for a minimum Trusted Computing Base (TCB).

This further illustrates capabilities being a generic security mechanism. Implementing secure peripherals is simply a specific instance of task isolation and secure inter-task communication.

Return and real-time guarantees with distributed trusted stacks and kernel CCall helper. To return from a fast CCall, the caller's PCC and DDC have to be securely restored. We implement distributed trusted stacks to protect these capabilities from tampering by the callee, as shown in Figure 4. A kernel capability register (named $KR1C, not accessible to user tasks) is reserved for this purpose, which points to a small stack for each task; a kernel CCall helper is inserted between the caller and the callee. With a trusted stack, a fast CCall first calls into a kernel helper, which pushes the caller's PCC and DDC onto the trusted stack, and then calls into the callee. A CCall return also returns into the helper, popping the caller's PCC and DDC before jumping back. These measures ensure the caller's return information cannot be tampered with or cleared; as a result, it is always possible to securely return to the caller regardless of the callee, further enforcing CFI between domains.

To enforce real-time guarantees, a caller may also specify a timeout for a CCall, which will cause a time stamp to be recorded. Time stamps are regularly checked by the timer interrupt, and force the callee to return after expiry. Only trusted tasks are given the privilege to specify timeouts, as a malicious task could combine this with the highly accurate timer to perform a series of side channel and timing attacks [28].

Secure centralized heap management. Two properties

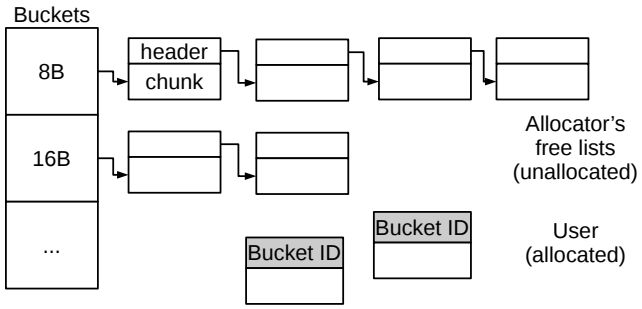

Fig. 5. Memory allocator structure (gray boxes indicate that the bucket ID is sealed inside a sealed capability)

of CHERI – fine-grained memory protection and capability unforgeability – provide the ideal infrastructure to address the challenge of a safely shared heap. Every `malloc()` call is now a fast CCall into the heap manager that returns a bounded capability instead of a 32-bit pointer. With strict bounds and permissions, we can safely use dlmalloc style allocators for low latency because it is impossible to overflow into the metadata or into other allocations.

Capabilities being unforgeable can be further leveraged to enforce heap integrity. As shown in Figure 5, the header of a freed memory chunk has a capability to the next free chunk; the header of a capability in use has a sealed capability with the pointer field being the bucket ID. Upon each `free()` call, the allocator verifies that the sealed capability is intact, changes it into the header of a freed chunk and then returns it to the free list. It is impossible for a user task to forge the sealed capability because each task is assigned a unique otype, and others cannot seal a capability with the heap allocator's otype. In this way, the allocator is able to verify that the memory chunk being freed was indeed allocated by the allocator itself, effectively preventing user tasks from freeing arbitrary memory and from freeing an already freed chunk to contaminate the free lists.

The implementation of the heap manager fits the requirement of providing a secure and fine-grained dynamic memory allocation subsystem. Unlike TrustLite, the capability register file is not privileged (except for $KR1C) and capabilities can be loaded and stored within the user's domain like traditional pointers, therefore no kernel calls are required to modify capability entries. This also means that heap allocations are scalable, as the user is no longer restricted by the number of protection entries in MPU-based solutions. For example, it is possible to construct a pointer-based data structure (e.g., a linked list) with each node being a bounded capability from the heap, whereas the limit of MPU entries often requires the programmer to request a single large contiguous memory region as a pool and manually allocate from it, losing any fine-grained protection on individual nodes.

V. Evaluation

We have chosen the MiBench benchmark suite to evaluate performance. MiBench is an open-source commercially representative benchmark suite targeting embedded system workloads [29]. We ported the benchmarks with minor modifications to several APIs so that they function under the security primitives of CheriRTOS.

On the software side, we compare CheriRTOS with its baseline version that removes all CHERI protections by using direct jumps, unguarded pointers, and offering no return or real-time guarantees. In order to compare and evaluate hardware costs, we also implement the PMP unit (the MPU component for RISC-V), as it has a state-of-the-art feature set and an open-source specification.

A. Software costs

Dynamic tasks with non-PIC. The baseline RTOS without CHERI uses Position Independent Code (PIC) for tasks to load them dynamically. Offsetting all memory accesses by the base of the capability in CHERI removes the need for PIC or run-time relocation. In our evaluation we found that avoiding PIC translates to a run-time performance gain of around 5% and a reduction of memory access of 10% on our platform due to a lower instruction count and fewer indirections for memory access. (Non-MIPS ISAs with better PIC support will see less benefit.) Therefore, to compare fairly with the baseline, we compile all benchmarks under CheriRTOS in PIC, to separate the overhead due to CHERI protections from the benefit of CHERI relocation.

Context switching. Context switches include task scheduling, system calls, exceptions, and interrupts. Due to the additional capability registers (8 user and 3 kernel capabilities), the handler needs 22 more cycles (including exception entry and exit) for each context switch. The priority-based pre-emptive scheduler in the baseline system requires 230 cycles (2.3μs) to schedule the next task, which means that the capability context adds around 10% overhead in context switches.

Fast and direct domain crossing. Message queues in CheriRTOS provide buffered inter-task communication primitives, which are created and controlled by the kernel and scheduler. With fast CCall, we are able to bypass the kernel and exception paths, and directly CCall into the callee – significantly reducing the latency of inter-task communication.

The results are presented in Figure 6. The benchmark is done by sending a short message from a user task to an encryption task, and returning immediately. The baseline system does this by directly jumping to the entry point of the callee. Under CheriRTOS, a direct jump is impossible due to task isolation, and must be done via CCall. We implement two CCall paths. One signals an exception to a kernel handler that performs software otype checking and unsealing before jumping to the new domain, similar to the exception-based domain crossing in Cortex A and R TrustZone. The other is the fast CCall path, which uses the `CCallFast` instruction to perform hardware and exception-less inter-task calls.

Compared with a direct jump, the overhead of fast CCall comes from the preparation of caller and callee capabilities and the trusted stack. For exception-based CCalls, the dominant overhead is exception entry and exit, and the attendant

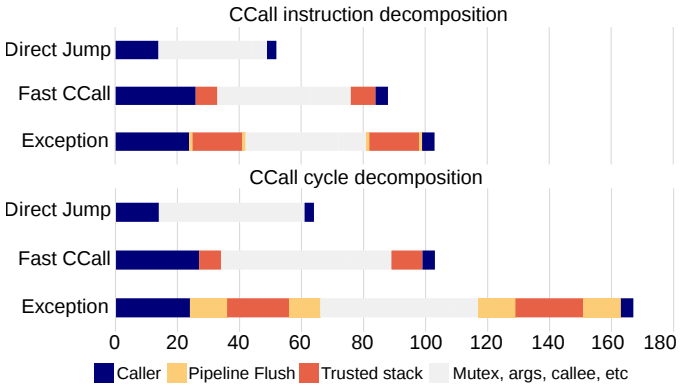

Fig. 6. Instruction and cycle counts for a round trip: direct jump vs. capability jump (fast CCall) vs. exception based CCalls

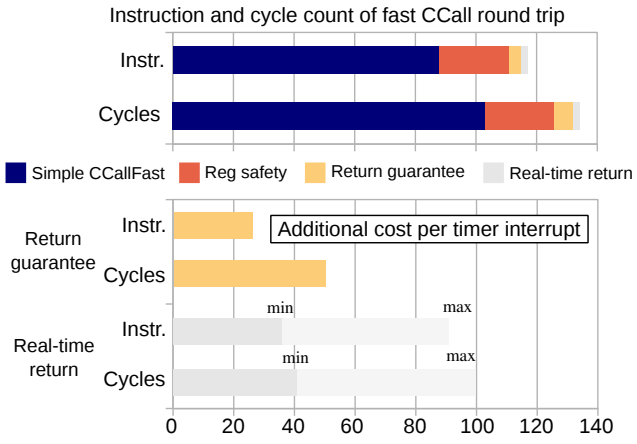

Fig. 7. Overhead of different protection levels: the bar chart at the top is additive, and the next protection level includes the costs from the previous level. The bottom shows the additional costs in the timer interrupt. Both the return and the real-time guarantees require the timer interrupt to check the time stamps on the trusted stack, while simple CCallFast and register safety do not require timer interrupt modifications. The return guarantee computes a hash and examines one random stack slot at a time, while the real-time guarantee has a cost that depends on the run-time trusted stack depth, between examining only one stack slot (min) and a full stack (max).

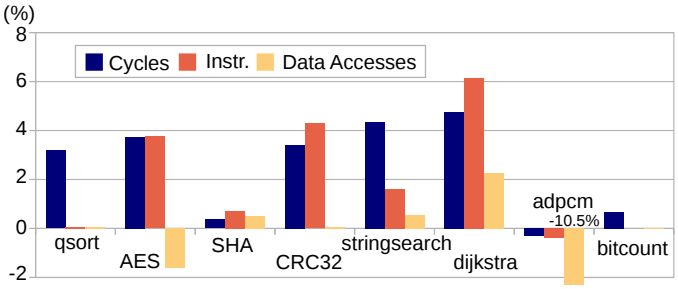

Fig. 8. Overall overhead across benchmarks

pipeline flushes. Each round trip involves 4 pipeline flushes, wasting up to 40 cycles. Bypassing the exception path with `CCallFast` avoids this overhead, drastically reducing the Cycle Per Instruction (CPI) from 1.64 down to 1.17.

Register safety, return and real-time guarantees. Domain crossing with fast CCalls achieves low-latencies while still supporting strong task isolation. To guarantee register safety as well, a caller should backup all its callee-saved registers, and clear all non-argument registers before performing a CCall. Similarly, a callee that does not trust the caller would clear all registers except for return values. These require additional instructions in the CCall prologue and epilogue. This additional latency is required only when interacting with untrusted modules; for example, a user calling `malloc()` may trust the heap allocator to maintain registers.

In addition, we attach time stamps to trusted stack entries, and use timer interrupts to guarantee that malicious callees cannot violate real-time constraints. For evaluation, we implement both randomized trusted stack checking and full trusted stack traversal. For randomized checks, the timer interrupt inspects only one random slot on the trusted stack, giving deterministic timer interrupt delays, but inaccurate expiry of CCalls. The full traversal inspects all slots on the stack and checks expiry for all time stamps. This full traversal adds non-determinism, as the timer interrupt delay now depends on the depth of the trusted stack, but may be necessary for tasks requiring precise real-time guarantees. To limit non-determinism, we restrict the trusted stack depth to only 4. This is clearly a trade-off between supporting long CCall chains and determinism; we believe that in practice a depth of one or two may be sufficient, as it is unlikely that a real-time task being called will further perform fast CCalls to deepen the stack.

Figure 7 shows the costs in both the CCall routines and the timer interrupt.

Overall system performance. In addition to latency and determinism, we evaluate the overall performance across multiple MiBench benchmarks. All datasets (e.g., arrays to be sorted in qsort, strings to be searched in stringsearch, etc.) are allocated on the heap outside the domains of the benchmarks and must now be accessed via constrained capabilities.

Of these benchmarks, Dijkstra particularly stresses centralized heap allocation. It builds graphs by frequently calling `malloc()` and `free()`, which here use capabilities with cross-domain calls into the heap manager.

We have modified the AES and SHA benchmarks of the MiBench suite in order to stress domain crossing and inter-task communication. Rather than using statically-allocated local datasets, AES and SHA have been modified to receive data through inter-task communication. Another user task, *ccalltest*, CCalls into AES and SHA to perform encryption and message digest respectively on 1-MiB of data. This test has a data buffer of 8KiB in each invocation, therefore 128 domain crossings are required in each case. In the CHERI case, these are safe domain transitions with memory safety; in the baseline case, these are simple function calls passing unprotected pointers.

We set the timer interrupt at 100Hz, the suggested rate by FreeRTOS, to also detect the expiry of real-time tasks (a resolution of 0.01s). The results are shown in Figure 8.

Overall, the cycle overhead falls below 5%, varying from

4.7% to almost no overhead. Dijkstra sees the highest overhead because each node of the graph is dynamically allocated with CCall. In addition, as a capability is double the size of a 32-bit pointer, graphs constructed with capabilities have a larger cache and memory footprint. Despite these issues, Dijkstra's cycle count is still only 4.7% above the insecure baseline.

In two benchmarks, namely AES and adpcm, a negative overhead is sometimes observed. Tracing shows that having an additional 8 registers used for capabilities relieves register pressure and reduces stack loads and stores. Although in the extreme case data accesses are reduced by 10%, they are typically on the stack with good spatial locality, which normally hit in the data cache and have less impact on cycles.

B. Hardware costs

Our baseline is the BERI processor. We synthesize two configurations, one enhancing BERI with the capability coprocessor and the other adding a Physical Memory Protection (PMP) unit for memory access control. To make the costs clear, we disable BRAM usage so that all logic is generated using combinational circuits or registers. We synthesize 5 times and take the mean and deviation.

Area. We choose to implement 8 PMP registers, to match typical commercial MPUs and also to match our capability coprocessor. The logic utilization is shown in Table I. Overall, the capability coprocessor has 39.4% more logic utilization. Note that the capability coprocessor not only supports capability registers, but also implements the CHERI ISA; the PMP is a basic implementation of the RISC-V specification. We anticipate that any extensions on PMPs or MPUs will quickly increase the logic utilization. These include increasing the register count to 16 to enable more simultaneous tasks, the Execution-Aware MPU from TrustLite and TyTAN, separating MPU entries further into subregions as in ARM embedded processors, etc. In terms of logic usage in an ASIC, we do not think that a well optimized capability coprocessor is much more expensive than a commercial MPU or PMP.

Critical path and timing constraints. The pipeline of the capability coprocessor is simple and operates in parallel with the main pipeline, not disturbing the critical path. However, fitting the PMP into our pipeline proves to be very difficult. For 8 PMP entries, we have to perform a full associative match, introducing 32×32-bit comparators (each PMP entry has a base and length and the associative match has to be done for both data and instruction fetch) within a single clock cycle. This difficulty is confirmed by timing analysis, which shows that the maximum clock frequency achieved is around 20MHz lower than with the capability coprocessor (Table I). Unsurprisingly, further analysis reveals that the PMP lies in the critical path while the capability coprocessor does not.

TABLE I
FPGA RESOURCE UTILIZATION AND TIMING

Category	PMP Unit	Capability Coprocessor
Total ALUTs	5174 ± 26	7212 ± 27
Combinational w/o register	4241 ± 26	3879 ± 25
Combinational w/ register	759 ± 20	2278 ± 18
Register only	174 ± 20	1055 ± 20
Core clock freq. (MHz)	86.71 ± 6.37	105.83 ± 3.13

Power. Although FPGA synthesis is not indicative of ASIC power, we can still qualitatively estimate the power consumption. The major power draw from PMPs (as well as MPUs) is the large number of comparisons within each cycle. A capability coprocessor has drastically reduced power consumption due to the absence of associative searches. CHERI always specifies the region of each access explicitly, so that only one bounds check is required for instruction fetch, and one for data access: instruction fetch is checked against PCC, and the data access is checked against either the DDC or an explicit capability. On the other hand, the CHERI coprocessor does require additional power to decompress the compressed capability bounds, but this is still dramatically less than the power required by an active MPU. We might note that the MPU model also has room for optimization. For example, the EA-MPU avoids full associative searches by linking code and data regions. Once the code entry is matched, only linked data regions will be searched.

VI. Future work and directions

Many future directions are worth pursuing to improve the efficiency of our system.

Having a separate capability coprocessor requires separate logic and extra context switch latency. It is possible to extend existing registers to hold capabilities and to merge the capability coprocessor operations into the main core, reducing the context-switch latency and overall hardware logic usage.

Currently, pushes and pops on the trusted stack are done in software by the kernel CCall helper, as the traditional MIPS pipeline is unable to handle complicated loads and stores in a single instruction. However, in other ISAs we imagine that the kernel CCall helper can be completely implemented in hardware to remove the software helper in the middle of a CCall, further reducing the latency of inter-task calls.

VII. Conclusion

In this paper, we present the CHERI-64 processor and CheriRTOS, a hardware-software architecture for embedded systems. We conclude that our capability approach addresses many existing memory safety problems for embedded systems, and also enables novel, efficient, and scalable solutions to task isolation with control-flow integrity. Both the software and hardware evaluations confirm that these benefits can be achieved without violating the performance and determinism constraints of embedded systems.

Our review of state-of-the-art security architectures for embedded systems demonstrates the difficulty in finding a comprehensive security framework that is efficient, scalable and generic. Nevertheless, our implementation relies on the fundamental protection mechanisms of CHERI, which are

utilized by our platform to enforce fine-grained memory protection, task separation with fast and secure inter-task domain crossing, secure peripherals, secure centralized heap allocation and returns, and real-time guarantees.

We envisage a future where a fine-grained and unified security interface eases the safe design and deployment of embedded software systems. For example, removing position-independent code or run-time relocation results in a much simpler binary loader; separating peripherals from the kernel reduces its complexity and the attack surface; fine-grained memory protection removes many manual (and likely incomplete) bound checks and assertions made by programmers, resulting in higher assurance and performance. New software stacks targeting CHERI-64 and CheriRTOS specifically could easily achieve high robustness and efficiency, with less effort than state-of-the-art approaches.

REFERENCES

[1] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A Large-Scale Analysis of the Security of Embedded Firmwares," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 95–110.

[2] G. Beniamini, "Over The Air: Exploiting Broadcoms Wi-Fi Stack," 2017. [Online]. Available: https://googleprojectzero.blogspot.co.uk/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html

[3] ——, "QSEE privilege escalation vulnerability and exploit," 2016. [Online]. Available: https://bits-please.blogspot.co.uk/2016/05/qsee-privilege-escalation-vulnerability.html

[4] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, "Comprehensive Experimental Analyses of Automotive Attack Surfaces," in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 6–6.

[5] *ARMv8-M Memory Protection Unit*, 0200th ed., ARM Ltd., 2 2017.

[6] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanovi, "The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10," EECS Department, University of California, Berkeley, Tech. Rep., May 2017.

[7] *TrustZone technology for ARMv8-M Architecture*, 0101st ed., ARM Ltd., 8 2016.

[8] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A Security Architecture for Tiny Embedded Devices," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: ACM, 2014, pp. 10:1–10:14.

[9] F. Brasser, B. E. Mahjoub, A. R. Sadeghi, C. Wachsmann, and P. Koeberl, "TyTAN: Tiny trust anchor for tiny devices," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, pp. 1–6.

[10] L. Davi, M. Hanreich, D. Paul, A. R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "HAFIX: Hardware-Assisted Flow Integrity eXtension," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, pp. 1–6.

[11] L. Davi, P. Koeberl, and A. R. Sadeghi, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2014, pp. 1–6.

[12] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013, pp. 479–498.

[13] R. Strackx, F. Piessens, and B. Preneel, "Efficient Isolation of Trusted Subsystems in Embedded Systems," in *Security and Privacy in Communication Networks*, S. Jajodia and J. Zhou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 344–361.

[14] D. Midi, M. Payer, and E. Bertino, "Memory Safety for Embedded Devices with nesCheck," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17. New York, NY, USA: ACM, 2017, pp. 127–139.

[15] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner, "Memory Safety Without Garbage Collection for Embedded Applications," *ACM Trans. Embed. Comput. Syst.*, vol. 4, no. 1, pp. 73–111, Feb. 2005.

[16] J. Evans, "A scalable concurrent malloc(3) implementation for FreeBSD," in *BSDCan*, 2006.

[17] D. Lea, "A Memory Allocator," http://g.oswego.edu/dl/html/malloc.html, April 2000.

[18] J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," *Communications of the ACM*, vol. 17, no. 7, Jul. 1974.

[19] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, June 2014, pp. 457–468.

[20] R. N. M. Watson, D. Chisnall, B. Davis, W. Koszek, S. W. Moore, S. J. Murdoch, P. G. Neumann, and J. Woodruff, "Capability Hardware Enhanced RISC Instructions: CHERI Programmer's Guide," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-877, Sep. 2015.

[21] R. N. M. Watson, R. M. Norton, J. Woodruff, S. W. Moore, P. G. Neumann, J. Anderson, D. Chisnall, B. Davis, B. Laurie, M. Roe, N. H. Dave, K. Gudka, A. Joannou, A. T. Markettos, E. Maste, S. J. Murdoch, C. Rothwell, S. D. Son, and M. Vadera, "Fast Protection-Domain Crossing in the CHERI Capability-System Architecture," *IEEE Micro*, vol. 36, no. 5, pp. 38–49, Sept 2016.

[22] D. Chisnall, B. Davis, K. Gudka, D. Brazdil, A. Joannou, J. Woodruff, A. T. Markettos, J. E. Maste, R. Norton, S. Son, M. Roe, S. W. Moore, P. G. Neumann, B. Laurie, and R. N. Watson, "CHERI JNI: Sinking the Java Security Model into the C," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 569–583.

[23] R. N. M. Watson, J. Woodruff, D. Chisnall, B. Davis, W. Koszek, A. T. Markettos, S. W. Moore, S. J. Murdoch, P. G. Neumann, R. Norton, and M. Roe, "Bluespec Extensible RISC Implementation: BERI Hardware reference," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-868, Apr. 2015.

[24] R. N. Watson, P. G. Neumann, J. Woodruff, M. Roe, J. Anderson, D. Chisnall, B. Davis, A. Joannou, B. Laurie, S. W. Moore *et al.*, "Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 6)," University of Cambridge, Computer Laboratory, Tech. Rep., 2017.

[25] A. J. P. Joannou, "High-performance memory safety - Optimizing the CHERI capability machine," Ph.D. dissertation, University of Cambridge, May 2018.

[26] *RM57L843 Hercules™ Microcontroller Based on the ARM Cortex-R Core*, June 2016 ed., Texas Instruments, 2016. [Online]. Available: http://www.ti.com/lit/ds/symlink/rm57l843.pdf

[27] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–.

[28] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, pp. 1–27.

[29] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, Dec 2001, pp. 3–14.