On CHERIfying Linux 5.15

Kui Wang*, Dmitry Kasatkin*, Vincent Ahlrichs†, Lukas Auer†, Konrad Hohentanner†, Julian Horsch†, and Jan-Erik Ekberg*
**Huawei Technologies*, Helsinki, Finland
†*Fraunhofer AISEC*, Garching near Munich, Germany

Abstract—**For a few years now, our team in Huawei Technologies, Helsinki System Security laboratory have been following the open-source CHERI work from Cambridge University with great interest, as this re-birth of a hardware capability system has the promise to revolutionize how the mobile industry deals with memory protection in consumer devices, beyond what can be achieved with technologies like ARM Pointer Authentication or Memory Tagging — features that already have appeared in contemporary processors.**

Of special interest to our team has been the consideration of achieved security benefit vs. performance degradation in the software stack. To analyze this in practice, we, together with a team from Fraunhofer AISEC, have ported a few of the OS kernels and base systems used today in products to the CHERI platform, more specifically the RISC-V one for now, with the intent to collect first-hand evidence of what level of compiler or software optimization still needs to take place to help make CHERI technology mainstream, and at the same time set up a reference platform to eventually demonstrate this opportunity.

This white-paper accompanies the open-sourcing of our current Linux 5.15 CHERI port for RISC-V, along with a minimal runtime. Considering the significant on-going research effort on the Morello ARM platform, we hope this work is of benefit for other Linux community ports taking place in the CHERI context, both by pin-pointing where in the code the implementation effort for a functioning CHERI adaptation needs to happen in Linux, and to provide indication of some of the design patterns applied by us to complete the necessary code adaptation needed when migrating from a non-CHERI ISA to a full-capability setup. Today, our Linux image runs in full-capability mode on CHERI RISC-V, but we do not yet claim it is full-featured or even properly analyzed for optimal capability application. It is however ready as a starting point for future research.

Index Terms—**Hardware Capabilities, Linux kernel, Mobile Security, RISC-V**

I. Introduction

This port of Linux to the CHERI (RISC-V) was developed to validate the performance and security properties of CHERI for Linux, which is the most used OS kernel today, especially in consumer and cloud. The "CHERIfication" of Linux, primarily involved two main endeavors: The first was to support user-space programs and daemons compiled with CHERI. In order to achieve this, programs needed to be loaded with the awareness that they were compiled with CHERI support — requiring changes in the program loader, acting on changes in the ELF format. The changes were needed to manage the capability-formatting of environment variables for the program. Also, the scheduler / interrupt handler in the kernel needed to be made CHERI-aware, i.e. to know whether a user-space process is CHERIfied or not, since register saves and restores have to account for whether capability registers are in use during scheduling. The second endeavor was to compile the kernel proper with CHERI memory protection, i.e. to let the bounds in CHERI capabilities guard the memory references within the kernel. The current state of this part of the CHERIfication covers only the main kernel, its memory management code, its bootstrap for RISC-V and selected drivers (filesystem, network) that have been used for validation in QEMU and on FPGAs. This part of the work mostly included fixes for pointer (capability) provenance, i.e. to modify casts from integers to pointers which in most architectures can be done, but in CHERI, the address must at least be accompanied with the range of the reference turning the pointer into a capability. A few instances where kernel code modified in this way actually turned out to reference memory addresses beyond the allocation (mostly different optimizations) where also corrected.

The accompanying open-source github repository [1] contains our CHERI-modifications to a number of different existing projects centered around the Linux kernel and a very minimal run-time for it. The project is complete enough to run the Linux kernel with a runtime consisting of either the MUSL or glibc C standard library and a few applications (busybox, ssh, dbus, systemd) on top of the QEMU RISC-V CHERI emulator and necessary scripting (buildroot) is included to replicate this setup. We have also run the same code on a Xilinx FPGA with the CHERI Flute core for some first benchmarking experiments. We return to this in Section VI.

We hope the academic research, CHERI and Linux communities can leverage this work for further evolving the CHERI software stack towards the fully functional, deployed and secure computing architecture it deserves to become.

II. The CHERI Architecture

The project on Capability Hardware Enhanced RISC Instructions (CHERI) is a hardware-software co-design project to enable hardware capabilities in a contemporary ISA on modern processors — to "enable fine-grained memory protection and highly scalable software compartmentalization" [2]. This research direction has been on-going for more than a decade at the University of Cambridge Computer Laboratory [3], and even though recent research investment like the Digital Security by Design Initiative [4] and, e.g., the Morello Project that prototypes CHERI capabilities on ARM hardware have expanded the research and evaluation work on CHERI much beyond Cambridge, the university still takes center stage in its evolution.

A one-line introduction to CHERI from a memory protection and software perspective is that processors with this technology provide a mechanism for memory references (pointers) — now called capabilities — to architecturally include metadata such as bounds for the memory area the capability is allowed to dereference. CHERI also implements strict typing between capabilities stored in memory and other types of data, backed by a memory tagging architecture. These two architectural features provide the fundament of the capability system. The CHERI mechanisms has, to date, been implemented on at least MIPS, RISC-V and ARM architectures. An excellent introduction and in-depth material for learning about CHERI is available through research papers and technical reports from the University of Cambridge (e.g. [2], [5], [6] and [7]) — these constitute a starting point for learning about CHERI, their content is not repeated or digested here. Much of the rest of the material in this white-paper assumes a rudimentary understanding of CHERI and its ISA.

To be noted is that CHERI exists in the open source in many forms that all came together in making our project possible. HDL-based processor (Bluespec) images are available for a set of cores that implement CHERI, our work has been done on the Flute RISC-V CHERI core. The university maintains CLANG/LLVM compiler images supporting CHERI extensions for the platforms mentioned above, we have made use of the CHERI RISC-V compiler toolchain to complete this work. In addition, detailed reference manuals for their ISA are available. QEMU support is also available for CHERI for initial testing and debugging. University researchers were also kind enough to take compiler bug reports and explain ISA intricacies in chat groups along our journey, and all of these were crucial to our eventual success to get Linux running with CHERI.

III. The modified code

As stated above, the code modifications we provide cover a few open-source projects. Table I collects these in one view. The main effort has been on the Linux kernel, but also getting the MUSL and glibc C libraries CHERIfied was a significant endeavor. To note is still that the lines of code needing modification to compile and run CHERI (for kernel and libc) is insignificant compared to the total code-bases of the respective projects — the compiler does most of the heavy-lifting. This seems to hold even more true for system and user applications, if our very limited sample can be considered any indication. Of course any single modification constitutes a separate analysis and fixing activity, either in the form of a compilation error or, more often than not, a run-time crash, so even few code changes may represent a fair bit of debugging effort. In Section IV we provide some insight into the most common fixes and modification patterns that we ended up applying in the kernel context.

Figure 1 shows the overview of the system that is provided as part of this project. The kernel can be compiled either in CHERI hybrid mode, where the kernel supports applications with capability protection, but is not itself engineered for memory protection, or in CHERI full-capability mode where

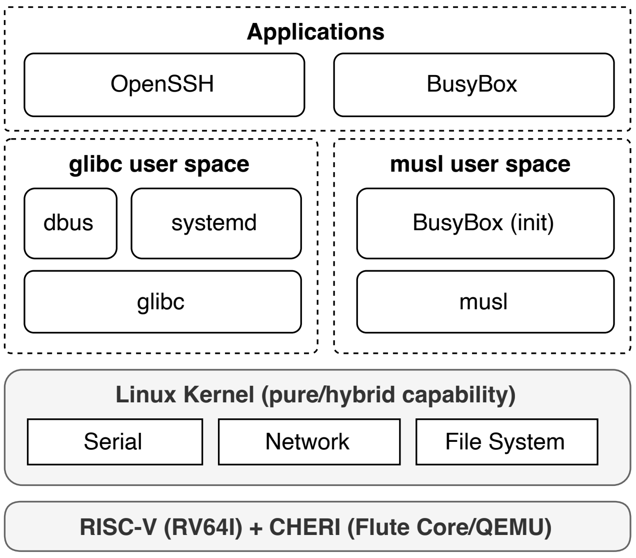

Figure 1. Architecture overview of our RISC-V CHERI Linux system

memory protection in the form of allocation boundary checks with capabilities is applied to all software parts of the system. We provide two user space variants, one based on our musl port and the other on our glibc port. The glibc variant uses our cherified systemd version as init system, the musl variant uses the cherified BusyBox. Both variants use our BusyBox and OpenSSH CHERI-ports to realize a basic shell environment.

IV. Modifying the Linux Kernel for CHERI

The CHERI Clang/LLVM compiler from Cambridge University successfully compiles most of the Linux kernel source code, configured for the 64 bits RISC-V architecture, without any changes. In our current work, only some 200 files were modified because of CHERI. The majority of those files have only been marginally modified, caused by us fixing compiler errors / warnings.

Some code causes run-time exceptions due to CHERI tag or bounds violations. A typical example is that, in the original Linux code, a `pointer` is cast to `unsigned long`, then manipulated by bit-wise OR/AND/SHIFT, and finally cast back to a `pointer`. This is obviously violating the pointer provenance of CHERI, not least because a 128 bits capability cannot be represented by a 64-bit long. This is not an error or mistake by default — to note is that this approach is perfectly acceptable in legacy 64-bit systems. The template pattern / code change to fix these problems often involves using `uintptr_t`, type-defined to the CHERI Clang/LLVM compiler built-in type `__uintcap_t`. When the `unsigned long` is replaced with `uintptr_t` the capability provenance is maintained and the tag is not dropped during the cast as the `uintptr_t` is guaranteed to fit a pointer / capability. The semantics of the original code can thus be maintained, and the fix is backwards compatible (say `uintptr_t` can be type-defined to `unsigned long` for conventional 64 bits RISC-V architectures).

Table I
OVERVIEW OF SOFTWARE PROJECTS MODIFIED AS PART OF THE CHERIFICATION PROCESS

Project	URL	Description	LoC
buildroot	https://buildroot.org	Tool to generate setup	N/A
Linux Kernel 5.15	https://www.kernel.org	Linux kernel	5942
MUSL libc	https://musl.libc.org	Lightweight libc for Linux syscalls	2030
glibc	https://www.gnu.org/software/libc/	GNU C Library	2268
Busybox	https://busybox.net	Unix utilities + shell in single executable	21
OpenSSH	https://www.openssh.com	OpenBSD secure shell	6
OpenSSL	https://www.openssl.org	Secure communication library	24
systemd	https://www.freedesktop.org/wiki/Software/systemd/	System and service manager	52
dbus	https://www.freedesktop.org/wiki/Software/dbus/	Message bus system	29

One thing to clarify is that the pointer provenance upgrades were done based on run-time errors, not by static analysis. To consistently modify all problematic cases with this method would require thorough code coverage support. This has not yet been performed, but we have modified all encountered cases from device power-up until the user space command line prompt appears. Additionally, unit tests, performance tests and some ad-hoc usage of applications like SSH have uncovered a few additional kernel provenance cases that have been fixed. But there is no guarantee that the port is complete beyond the practical testing effort.

In practice, assembly code is changed in `arch/riscv/include/asm/` most notably in `atomic.h`, `bitops.h`, `cherireg.h`, `cheri.h`, `cmpxchg.h`, `ptrace.h`, `syscall.h`, `uaccess.h`, and for architecture code in `arch/riscv/lib/` changes are mainly done in `memcpy.S`, `memmove.S`, `memset.S`, as well as in `uaccess.S`. In-kernel modifications in `arch/riscv/kernel` are concentrated around `head.S`, `entry.S` and `process.c`.

The CHERIfication of drivers is not complete by virtue of this work. Instead, we have only provided modification to drivers that were necessary for the platforms (`qemu` and `FPGA`) we have tested on. The drivers that were changed in the context of this work represent this minimal port: `drivers/block`, `drivers/tty`, `drivers/char`, and to support them, also changes to file system under `fs/`, memory management under `mm/`, network code under `net/` as well as headers under `include/linux/` were modified.

V. EXAMPLES AND CONTEXT

In this section we provide a few examples of CHERI porting that we encountered during the project. These are not intended to be a complete list of issues solved nor a comprehensive porting guide — instead we hope these examples will provide an idea of the type of work involved when porting a full-featured OS kernel to the CHERI platform.

A. Pointer provenance

Above, we discussed at length how code that casts pointers to unsigned long explicitly causes the pointer to lose capability provenance. Here is a textbook example of such an issue found in `fs/ext4/mballoc.c`:

Listing 1. Pointer provenance
```
static inline void *mb_correct_addr_and_bit(int *bit, void
     *addr)
{
#if BITS_PER_LONG == 64
  *bit += ((unsigned long) addr & 7UL) << 3;
  addr = (void *) ((unsigned long) addr & ~7UL);
#elif BITS_PER_LONG == 32
  *bit += ((unsigned long) addr & 3UL) << 3;
  addr = (void *) ((unsigned long) addr & ~3UL);
#else
#error "how many bits you are?!"
#endif
  return addr;
```

In this situation, `addr` as a pointer is first cast to `unsigned long`, bit manipulated, then cast back to a pointer. After compiling for the CHERI RISC-V ISA and executing it on CHERI RISC-V, `addr` is represented by a capability and its memory storage is tagged to mark the capability provenance. As the tag is cleared if `addr` is cast to `unsigned long`, any later dereferencing of `addr` will cause an exception to be raised. The appropriate fix for this issue is listed below:

Listing 2. Pointer provenance fixed
```
*bit += ((uintptr_t) addr & 7UL) << 3;
addr = (void *) ((uintptr_t) addr & ~7UL);
```

The example above is benevolent, as the fix is concentrated to one location in the code. Unfortunately, the kind of modification needed (capability exception caused by lost provenance) can often be more complex and require modification in multiple header and source files when the cast value is passed around using macros, or as function arguments. But in any case, this is the most common problem when CHERIfying legacy code.

B. Generating capabilities

To accommodate the existing code structure in the Linux kernel and to limit the amount of modifications to it, there are numerous cases where a pointer / capability needs to be explicitly constructed from an integer. Particularly in the part of the Linux kernel that handles memory management, the practice of using unsigned integers to temporarily store memory addresses is not uncommon, whereby the recreation of the pointer / capability is needed in the case where a complete re-write of the memory management code is not undertaken. The CHERI Clang/LLVM compiler provides built-in functions to gain provenance for such pointers from a default global data capability, i.e. capability metadata (with e.g. initially very unconstrained boundaries) is used to make up the missing metadata for the capability cast from the unsigned integer, allowing the capability to be de-referenced. The following compiler built-in functions are used for this:

Listing 3. Compiler built-ins for capability (re)creation

```
__builtin_cheri_global_data_get()
__builtin_cheri_address_set(x, y)
```

The first function returns a capability equivalent to the global data capability, whereas the second one modifies the address of a capability while maintaining its provenance (i.e. sets the address). Thus, to intentionally convert an integer to a pointer, a capability equivalent to global data capability is used as a base, and its address is replaced with the one stored in the unsigned integer. We created a helper C function

Listing 4. Helper function

```
uintcap_t cheri_long_data(unsigned long addr);
```

to encapsulate this operation, and we use this in the porting wherever provenance of a capability cannot be easily established (and tighter capability bounds assigned). One example of such a case is the following:

Listing 5. Missing provenance

```
static inline void setup_vmalloc_vm_locked(struct vm_struct
      *vm, struct vmap_area *va, unsigned long flags, const
     void *caller)
{
  vm->flags = flags;
  vm->addr = (void *)va->va_start;
  vm->size = va->va_end - va->va_start;
  vm->caller = caller;
  va->vm = vm;
}

struct vmap_area {
  unsigned long va_start;
  unsigned long va_end;
  ..
};
```

In this function `vm->addr` is a pointer, assigned to the value of `va->va_start`, however the provenance of the capability that represents `vm->addr` cannot be established, because `struct vmap_area` is defined as above and `va_start` is defined as type `unsigned long`. Of course, if `vm->addr` is later de-referenced, an exception will be raised, but the current code does not give / store information about the intended pointer provenance — that would require deeper code modification. Therefore, we modified this code (shown below) to intentionally give provenance to the pointer, so it can be later de-referenced, and leave the assignment of tighter provenance as future work:

Listing 6. Assigning global provenance

```
vm->addr = (void *)cheri_long_data(va->va_start);
```

In general, we constrain the use of `cheri_long_data` to give provenance to pointers related to Linux memory management in files such as `arch/riscv/include/asm/page.h`, `arch/riscv/mm/init.c`, `mm/vmalloc.c` and `mm/ioremap.c`.

C. Allocators

For most buffers and references to buffers that can be statically resolved by the CHERI compiler, such as a memory allocation on the stack (a local array with defined length) the provenance of the capability for this buffer will be set by the compiler with proper bounds set to match the array. However, for heap memory allocators, the range can only be resolved at run-time, since the size of allocations is passed as an argument to the allocation function. Also the alignment and memory location of the allocation is determined dynamically. In such situations, we have modified the memory allocation function itself to set the range for the capability pointer. For example in the Linux kernel `mm/slub.c` allocator, in the allocation function

Listing 7. Memory allocator

```
static __always_inline void *slab_alloc_node(struct
     kmem_cache *s, struct list_lru *lru, gfp_t gfpflags,
     int node, unsigned long addr, size_t orig_size)
{ .. }
```

at the end of the function, before returning the pointer / capability to the caller, we have replaced the original code to set the proper boundaries for the allocated object as follows:

```
#ifndef CONFIG_CPU_CHERI_PURECAP
  return object;
#else
  return cheri_csetbounds(object, s->size);
#endif
```

where `cheri_csetbounds` is a macro defined to set the upper bound of a capability, using the compiler built-in function `__builtin_cheri_bounds_set((x), (y))` to provide the provenance.

In our port of the MUSL and glibc C-libraries, a very similar addition was made for the malloc function. These additions result in most of the memory allocations in the capability-enabled kernel (and in the run-time) to be properly bounded and enforced by CHERI — only the exceptions mentioned above have too lax provenance for the time being.

In addition, we found multiple user space programs such as dbus to include their own allocators. These have to be adapted to ensure an alignment of all memory allocations to 128 bit, to allow capabilities to be stored in them. In addition the allocators must be adapted to set the appropriate bounds on capabilities to the memory allocations.

D. When local modification is not enough

Like said above, the CHERIfication of Linux was performed with a conservative strategy to emphasize localized, contained modifications, and to not embark on a journey where Linux is rewritten as a capability OS. The main goal was to get to a point where Linux runs successfully, with in-kernel capability support as well as having the "hybrid" ability to run cherified workloads on a CHERI RISC-V target. Still, not all modifications could be completed in only local scope, and the modification of the `ioctl` service was one of those, as outlined below. In general, where either widely used data structures or function signatures needed to be augmented for capability support, this caused changes to spread in the code. As an example of this, we examine modifications done to the random_ioctl function signature:

Listing 8. Modifying the function signature
```
static long random_ioctl(struct file *f, unsigned int cmd,
     unsigned long arg)
{
  int __user *p = (int __user *)arg;
  int ent_count;
  ..
```

The random_ioctl function has an argument arg defined as unsigned long, and immediately on function entry this argument is cast to a pointer p. This is the typical case of a cast leading to a memory reference that will fail at dereferencing. At the same time, the semantics of this function expects its caller to pass a pointer to it so that the kernel can locate data to read or write from / to user space memory. We modified the function signature to contain a memory reference / pointer type:

Listing 9. Modified function signature
```
static long random_ioctl(struct file *f, unsigned int cmd,
     uintptr_t arg)
```

which allows a capability to be used through the interface. However, this started a chain reaction of modifications to be carried out. As the random_ioctl is present as the unlocked_ioctl member variable in struct file_operation

Listing 10. Struct file operation
```
const struct file_operations random_fops = {
  .read_iter = random_read_iter,
  .write_iter = random_write_iter,
  .poll = random_poll,
  .unlocked_ioctl = random_ioctl,
  .compat_ioctl = compat_ptr_ioctl,
  .fasync = random_fasync,
  .llseek = noop_llseek,
  .splice_read = generic_file_splice_read,
  .splice_write = iter_file_splice_write,
};
```

its definition in turn needed to be upgraded to the following:

Listing 11. Type update
```
struct file_operations {
  ..
  long (*unlocked_ioctl) (struct file *, unsigned int,
     uintptr_t);
  ..
};
```

The modification on struct file_operations turned out to have a wide impact across the Linux kernel, since all code that uses an instance of struct file_operations and its unlocked_ioctl member variable now have to provide a matching function pointer to it. This update alone caused changes in around 40 source code files, and considering that only few drivers have been ported at this time, the total cost of this update in the upstream kernel would be significantly higher. The example shows that although memory protection of individual CHERIfied memory references in the kernel is relatively easy and at large a compiler-assisted endeavor, wherever the interfaces and data structures need to be updated in a kernel context, the impact can be quite severe and hard to contain in a localized manner.

E. *Capabilities and memory access*

As discussed before, the provenance of a capability is lost when extracting part of it. This was discussed earlier on situations where the code casts a pointer to an integer and manipulates its address. A related issue is that some functions do not exchange pointers and integers explicitly, but they copy or move data between memory locations, and these memory location may contain capabilities. If a capability is not copied or moved in memory as a whole, its provenance is also lost. Functions that expose such operations include arch/riscv/lib/memcpy.S, arch/riscv/memset.S, arch/riscv/lib/uaccess.S, as well as lib/sort.c.

A memory location that contains a capability is aligned with the size of capability, and in CHERI RISC-V the size is 128 bits and the alignment 16 bytes. For a memory address that is 16 bytes aligned, we must assume that it can contain a capability, and when copying or moving its contents, we must do it at 16 bytes granularity. These modifications have been implemented e.g. in the files listed above, but also in the memcpy and memmove implementations in the MUSL and glibc C libraries. Take for example the memcpy.S implementation: We now split the source memory location to be copied into three regions: the first region is from the start address to the lowest 16-bytes aligned address in the source buffer, the middle region spans all 16-bytes aligned addresses up to the highest 16-bytes aligned address in source, and then follows a remainder region to the end address. For the first and the last regions, copying is (and can) be made at byte granularity, since we know it will not fit a capability:

Listing 12. Capability-aware memory copying
```
Region 1:
clbu t2, (ca1)
csb t2, (ca0)
cincoffset ca1, ca1, 1
cincoffset ca0, ca0, 1
bltu a1, t0, 1b

Region 2:
clc ct2, (ca1)
csc ct2, (ca0)
cincoffset ca1, ca1, CHERICAP_SIZE
cincoffset ca0, ca0, CHERICAP_SIZE
bltu a1, t1, 2b

..
```

We also implemented similar handling for lib/sort.c. In order to maintain provenances of capabilities, we added a function void swap_words_128(void *a, void *b, size_t n) to swap two memory region in 16-bytes granularity, and adjusted the rest of the C-code in the spirit explained above.

Functions may also intentionally read or write beyond boundaries of capabilities. For example, some string manipulation functions determine the end of a string by checking its final '\0' ending, and to optimize performance, they iterate based on the size of an unsigned long instead of char, looking for '\0' from the read unsigned long to determine the end of a string. Such operation can potentially cause the reading of memory beyond the boundaries of capabilities set for the char pointers. We encountered this type of problem in lib/strings, lib/strncpy_from_user.c and lib/strnlen_user.c, but also in the MUSL and glibc C libraries. In such cases we typically made a workaround by disabling the optimization and resorting to reading and writing one byte at time. Below is a typical case of such an optimization:

Listing 13. Type misuse causing boundary violation

```
while (max >= sizeof(unsigned long)) {
  unsigned long c, data;
  c = read_word_at_a_time(src+res);
  if (has_zero(c, &data, &constants)) {
    data = prep_zero_mask(c, data, &constants);
    data = create_zero_mask(data);
    *(unsigned long *)(dest+res) = c & zero_bytemask(data);
    return res + find_zero(data);
  }
  *(unsigned long *)(dest+res) = c;
  res += sizeof(unsigned long);
  count -= sizeof(unsigned long);
  max -= sizeof(unsigned long);
}
```

F. Kernel assembler parts

In CHERI RISC-V, some assembler instructions, particularly load/store instructions, have changed semantics compared to the standard RISC-V instruction set. Also, some pseudo-instructions have been replaced with new ones in capability mode. The compiler will of course use the new instructions when compiling for CHERI, but assembler code (both inlined in C code and included as individual files) have to be changed by hand. In the kernel code, we have replaced all RISC-V load/store instructions (using integer addresses) with CHERI load/store instructions accessing memory via capabilities. Most of these changes are located in the arch/riscv/include/asm/ directory, arch/riscv/kernel/head.S which represents the execution flow of very early kernel startup and arch/riscv/kernel/entry.S which defines exception entry and exit.

To illustrate such changes, we use arch/riscv/kernel/head.S as an example. It defines the _start_kernel assembly function that picks one core to run the main boot sequence, to set up virtual memory, to relocate the kernel to use virtual memory, and then it continues by calling other kernel initialization functions. When running the Linux kernel on a CHERI RISC-V target, _start_kernel also needs to initialize the CHERI __cap_relocs table, whose entries relocate to functions' calling addresses. After __cap_relocs is successfully setup, C function calls can be made. The CHERI Clang/LLVM compiler provides a function cheri_init_globals_3 for this purpose and we need to call it as part of _start_kernel:

Listing 14. Snippet from start_kernel

```
/* Save hart ID and DTB physical address */
  mv s0, a0
  mv s1, a1

  la a2, boot_cpu_hartid
  XIP_FIXUP_OFFSET a2
  REG_S a0, (a2)
  /* Initialize page tables and relocate */
  /* to virtual addresses */
  la sp, init_thread_union + THREAD_SIZE
  XIP_FIXUP_OFFSET sp
#ifdef CONFIG_BUILTIN_DTB
  la a0, __dtb_start
#else
  mv a0, s1
#endif /* CONFIG_BUILTIN_DTB */
  call setup_vm
#ifdef CONFIG_MMU
  la a0, early_pg_dir
  XIP_FIXUP_OFFSET a0
  call relocate
#endif /* CONFIG_MMU */
```

Above is the original _start_kernel function, which calls setup_vm, and subsequently calls relocation function to switch the kernel from physical to virtual address operation. When executing on a CHERI RISC-V target, the setup_vm call will fail, because it further calls other C functions before the _cap_relocs is set up. In code modified by us, we delay setup_vm to be called after __cap_reloc, and for the kernel to be able to switch to virtual address operation, we also implement a temporary memory mapping for this in create_page_tables:

Listing 15. Modified _start_kernel

```
#ifdef CONFIG_MMU
#ifndef CONFIG_CPU_CHERI
  la a0, early_pg_dir
  XIP_FIXUP_OFFSET a0
#else
  call create_page_tables
  la s2, early_pg_dir
#endif
  call relocate
#endif /* CONFIG_MMU */
```

Later we call:

Listing 16. Activating capability tables

```
cllc cra, init_cap_relocs
cjalr cra
..
mv a0, s1
cllc cra, setup_vm
cjar cra
```

Here, init_cap_relocs is called first, which calls into the compiler provided function cheri_init_globals_3. After that setup_vm is called. At this stage, the processor is switched to capability mode, and the instruction pattern to call functions is changed to the cllc and cjar pair of instructions.

G. Pointer size assumptions

One common issue we encountered during the CHERIfication of supporting libraries and user space applications is that some software makes assumptions on the pointer size. These break once the software is compiled for CHERI. Two software projects, where we encountered this were dbus and systemd. These problems are fixed by removing the static assumed pointer size.

The libdbus library uses a specific structure (namely DBusMessageRealIter), which should not be directly accessed by library users and is therefore opaque. In order to still permit stack allocations, libdbus offers a dummy structure which is casted to the actual structure where necessary in the library. Both structures must have the same size and alignment, which is ensured by assertions. With CHERI's 128-bit pointers, the size and alignment of both structures changes and breaks the assertions. Consequentially, the dummy structure had to be adapted during CHERIfication.

Systemd employs an optimized hashmap implementation that uses packed structures nested into each other. CHERI's 128-bit capabilities change the layout of those structures and breaks the assumptions in the code, manifesting in failing assertions. During CHERIfication, these structures had to be adapted.

VI. Evaluation

We evaluated both the security and performance of the final system. For the security evaluation we used a QEMU-based CHERI system and for the performance evaluation an FPGA-based CHERI system. In both cases, a Linux setup based on the glibc C libary was used.

A. Security Evaluation

With capabilities, the CHERI extensions offer a flexible concept for building software protection features. The main goal of CHERI is to provide fine-grained memory protection. The primary goal of our security evaluation is to ensure that the expected memory safety guarantees are achieved in the developed system. Additionally, the security offered by those features should be put into a greater context, pointing out areas in which the CHERI protections have no positive influence on.

Memory safety is a property of a program, language, or runtime and describes the absence of memory corruptions. Memory safety is typically divided into *temporal* and *spatial* safety. Both parts of memory safety refer to different bug types.

Spatial memory safety concerns the access bounds of pointers and objects. A pointer to an object should only be able to access the storage space of the object and must not read or write beyond the object's bounds. Typical spatial memory corruptions that should be tested are *buffer over-* and *underflows*. Both bug types can occur in different data regions of the program (stack, heap, and globals) and via read and write accesses, resulting in several different test cases. Over- and underflows can also occur *intra-object*, i.e., inside of complex objects such as a `struct` containing a buffer.

Temporal memory safety concerns the lifetime of dynamically allocated objects. The lifetime of those objects is typically explicitly managed by the programmer, which can lead to several issues, whose detection by the CHERI-based system should be tested. First, a *use-after-free* bug is present if a pointer to an already freed object (i.e., a "dangling" pointer) is used, which can lead to memory corruptions and leaks. Second, a *double-free* bug is present, if a pointer to a freed object is used to free the object again, which can lead to memory corruption by the allocator.

To get an impression of the security improvements the developed system and CHERI in general can offer in comparison to a non-CHERI system, we use tests from the Juliet Test Suite for C/C++ [8], [9]. The Juliet Test Suite is a set of tests developed by the NSA Center for Assured Software. In its current version, 1.3, it includes a large collection of test cases categorized under 118 different Common Weakness Enumerations (CWEs).

The Juliet test suite was developed with the evaluation of static analysis tools in mind. This goal is supported by the design of the single test cases. Each test case is a small, artificial piece of software that contains exactly one type of intended flaw. The software includes one function that exhibits the flaw ('bad') and one or more functions that implement a non-flawed ('good') version of the same functionality. With this ground-truth the ability of static analysis tools to classify the functions in a test case can be evaluated.

1) Test Setup: For our test setup, we made the following changes to the Juliet test suite to adapt it to our system and simplify the evaluation of the results.

a) Run script: We added a run script, which runs all selected 'good' and 'bad' test cases and collects their exit codes. We differentiate between 'normal exits', 'timeouts', 'segfaults', 'CHERI violations', 'allocation failures', 'explicit error exits', 'aborts' and 'unknown exit codes'.

b) Incompatible test cases: We filtered out Windows-specific test cases and C++ source files, because these depend on a cherified C++ standard library which is not available on our system.

c) Test cases expecting input: For automated testing, the test script already provides a timeout mechanism to handle hanging test cases. These test cases usually are waiting for user input or a socket connection. In a simple test setup as ours, these inputs, and also socket connections, are not provided. We filter out these test cases based on keywords in the functional component of the test case name, such as "fgets" or "socket".

d) Test cases with randomness: For some test cases randomized values for variables are used so that the intended flaw is not triggered every time. When evaluating static analysis tools, these test cases make sense. However, as we aim to evaluate the CHERI extension during run-time, the included randomness leads to unreliable results. For this reason we filter out test cases using random values, which are identified via the presence of the keyword "rand" in their name.

e) Flow variants: Multiple control and data flow variants exist for each functional flaw. For the evaluation we only included the base flow variant.

2) Results: The adapted test suite is run twice, once with CHERI-support enabled and once disabled. It contains 546 test cases distributed over 78 CWEs.

Table II shows the summary output of the run script for the plain RISC-V and for the cherified RISC-V system considering only the 'bad' code executables, i.e., test cases in which a normal exit means that a flaw remained undetected. In summary, CHERI reduces the number of normal exits by 108, the number of segfaults by 52, and the number of aborts by 8. In other words, the CHERI-based system detects and prevents 108 more flawed test cases. The number of timed out test cases and explicit test case exits remain the same. In total 168

Table II
SUMMARY OF THE AGGREGATED EXIT CODES OF THE NON-CHERI AND CHERI VERSION OF THE JULIET TEST SUITE.

	Plain RISC-V	RISC-V CHERI
Normal Exits	419	311
Timeouts	37	37
Explicit Error Exits	1	1
Allocation Error Exits	3	3
Segfaults	63	11
Aborts	23	15
CHERI Violations	0	168
Test Cases	546	546

Table III
SUMMARY OF THE AGGREGATED EXIT CODES OF THE NON-CHERI AND CHERI VERSION OF THE JULIET TEST SUITE FOR SPATIAL MEMORY SAFETY (CWES 121, 122, 124, 126, 127).

	Plain RISC-V	RISC-V CHERI
Normal Exits	111	6
Segfaults	41	0
CHERI Violations	0	146
Test Cases	152	152

CHERI exceptions (tag and length violations) are triggered. Of note, the cherified system also reduces the number of possibly exploitable segfaults. Instead of the segfaults, fine grained CHERI violations are triggered.

3) Discussion: It was shown that the CHERI extension successfully provides complete protection against spatial memory corruptions (cf. Table III). Of the 152 test cases, which can be classified as spatial memory flaws, 146 test cases lead to memory corruptions during run-time and trigger a CHERI exception during run-time. The remaining 6 test cases do not violate memory safety during run-time. Furthermore, CHERI also provides partial or transitive protection against several additional weaknesses. These weaknesses typically lead to spatial memory corruption or information leaks that are subsequently detected by CHERI.

However, CHERI does not yield any improved protection against many other weaknesses included in the Juliet test suite. On the one hand, many of theses weaknesses cannot be detected at run-time and, hence, are unrelated to CHERI. On the other hand, the CHERI extension and the cherified software components do not protect against temporal memory issues (cf. Table IV). However, there is research into a revocation-based garbage collection system for CHERI capabilities to enforce temporal memory safety for the heap [10], [11].

Table IV
SUMMARY OF THE AGGREGATED EXIT CODES OF THE NON-CHERI AND CHERI VERSION OF THE JULIET TEST SUITE FOR TEMPORAL MEMORY SAFETY (CWES 401, 415, 416, 562, 590).

	Plain RISC-V	RISC-V CHERI
Normal Exits	29	29
Segfaults	1	9
Aborts	19	11
CHERI Violations	0	0
Test Cases	49	49

B. Performance Evaluation

Apart from the effectiveness of the offered protection features, the performance of pure-capability software components and CHERI-based systems in general is crucial for CHERI's success. We used different benchmarks to get a general impression of the impact that CHERI has on the performance. In the following, the test setup is detailed before showing and discussing the gained results.

1) Test Setup: The Flute CPU, a 5-stage in-order RISC-V core, was used for the performance evaluation. It has been extended with CHERI support by the University of Cambridge. The core at commit version `3c31d0072e`[1] was used to run the benchmarks. It was compiled using the bluespec compiler at commit version `d05342e358`[2], together with `bsc-contrib` at commit version `2bd5f09170`[3]. It was synthesized to run at 94 MHz on a Xilinx Virtex UltraScale+ FPGA (Xilinx VCU118 evaluation kit) using Xilinx Vivado 2020.2.

The benchmarks are compiled with the CHERI clang toolchain and are run on a buildroot-based, hybrid capability system configured to use the busybox-init system and the glibc C library. The root filesystem of the system is stored in RAM and an SD-card attached to the FPGA board is used to transfer the benchmark results off of the board. Additional detailed performance metrics are collected from each benchmark run with the `minimal_count_stats`[4] tool. The corresponding performance counters are set up in `riscv-pk`[5] and are briefly described in the documentation of the Flute core[6].

At the start of the data collection, each benchmark goes through a warm-up phase. In this phase, the benchmark is run twice without data being collected from the run. This ensures that the internal state of the processor is the same for the following benchmark runs. Afterwards, the benchmark is run ten times.

The results of the benchmarks are compared with a baseline with no CHERI software components, the plain platform. It uses the same setup as described above (same software and compiler version for all components), but does not activate the CHERI software support. The glibc version used in both platforms differs in one important aspect. Several glibc optimization are disabled on the CHERI platform, but are present on the plain platform. The exception is `memcpy`. We disabled its optimizations on the plain platform as well, because the

[1]https://github.com/CTSRD-CHERI/Flute/commit/3c31d0072e
[2]https://github.com/B-Lang-org/bsc/commit/d05342e358
[3]https://github.com/B-Lang-org/bsc-contrib/commit/2bd5f09170
[4]https://github.com/CTSRD-CHERI/cheribsd/tree/master/usr.bin/minimal_count_stats
[5]https://github.com/CTSRD-CHERI/riscv-pk/blob/cheri_purecap/machine/minit.c#L128
[6]https://github.com/CTSRD-CHERI/Flute/blob/CHERI/Doc/Performance_Monitor/Performance_Monitoring.md

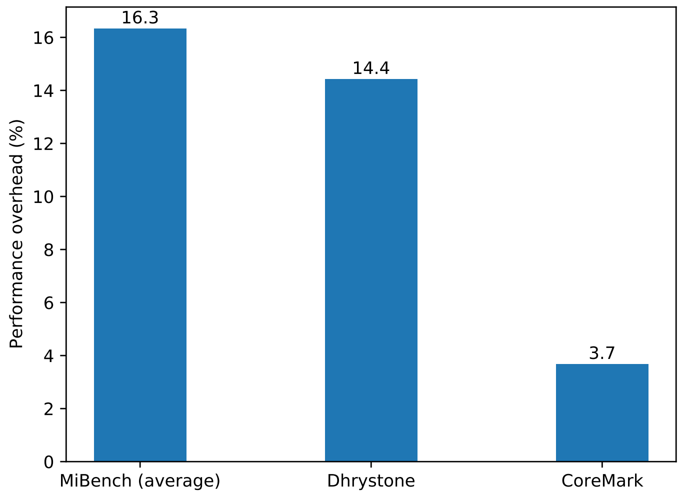

Figure 2. Overhead of the three evaluated performance benchmarks

benchmarks make frequent and heavy use of it, which would skew the results against CHERI.

For a comprehensive impression of the CHERI performance, different benchmarks have been selected:

- Dhrystone [12] is a synthetic benchmark designed to test a system's integer performance. It was constructed by extracting common constructs from a broad range of software. The final score is presented as the number of Dhrystone runs per second. In the performance evaluating setup described here, Dhrystone is configured to calculate its final score over 10 million runs.
- CoreMark [13] was developed as a replacement to the Dhrystone benchmark. Its goal is to provide a portable benchmark suite for CPUs and MCUs used in embedded systems. The final score is based on the time required for completing a set of algorithms (list processing, matrix manipulation, state machines, and CRC calculation).
- MiBench [14] is a set of benchmarks designed to represent commercially representative embedded tasks. These include JPEG decode / encode, AES encrypt / decrypt, and SHA1.

2) Results: The main result from the performance evaluation is the overhead created by the CHERI versions of the benchmarks in comparison to the baseline, non-CHERI versions. In the following, this overhead is illustrated and discussed for the different benchmarks.

Figure 2 shows the average overhead of the CHERI system compared to a non-CHERI system in all three benchmarks. The MiBench and Dhrystone benchmarks show a 16.3% and 14.4% overhead respectively. For MiBench, the value was calculated by taking the average overhead of its 20 sub-benchmarks. During measurements, the cherified CoreMark system showed only 3.7% overhead.

In Figure 3 we can see more detailed results of the individual MiBench test cases. Here, the overheads range from 49.1% for the complex `network-patricia` benchmark, to only 1.7% for the much simpler `telecomm-FFT` algorithm.

Based on the analysis of the additional performance metrics data collected from the benchmarks using the `minimal_count_stats` tool, the following observations were made. Note that they are limited due to several reasons.

- Detailed micro-benchmarks are required to be able to quantify the overhead created by single events tracked by the performance counters. The analysis below makes assumptions regarding which events are more expensive.
- The CPU architecture, for example, the instruction and data cache configurations, was not considered in the analysis.
- The compiler output, specifically the differences between CHERI and non-CHERI binaries, was not analyzed. Changes to the binaries can have a large impact on the resulting performance, for example by changing how efficiently the caches are used.

It was observed that the performance overhead is closely linked to the amount of additional instructions executed by the CHERI variants. In many cases, the instruction overhead directly correlates with the performance overhead. From a preliminary analysis, the cause of the instruction overhead appears to be unoptimized library functions of glibc. Several optimizations present in the glibc version of the plain platform were incompatible with CHERI and had to be removed as part of the cherification process. Improved performance is likely possible by developing alternative, CHERI-compatible, optimizations. However, it is anticipated that this will not be possible for all optimizations. A performance overhead is therefore always expected to be incurred due to less performant library functions.

The instruction and data caches were identified as another factor affecting the performance overheads. Due to changes in memory use with 128 bit capabilities (data cache) and of the binary itself (instruction cache), the cache access patterns are different and the memory sub-system may experience increased pressure with CHERI binaries. This can have both positive and negative impacts on the cache efficiency. However, a negative impact was observed more frequently. Particularly the impact of 128 bit capabilities on the data cache efficiency should be investigated further to determine if alternative cache designs are more advantageous.

3) Discussion: The benchmarks used in the performance evaluation had an average performance overhead of between 3.7% and 16.3%. Individual benchmarks showed a performance overhead of between 1.7% and 49.1%. Overall, these values are manageable or negligible in production systems. In some target applications, however, the performance overhead of the current system may be too high. Since the current system is a prototype, the benchmark results must be considered as upper bound values. With more work towards optimizing the CHERI system, it is expected that the performance overhead can be reduced.

VII. Conclusion

We have presented our initial CHERI port of Linux 5.15 for RISC-V, which supports both a hybrid and full-capability

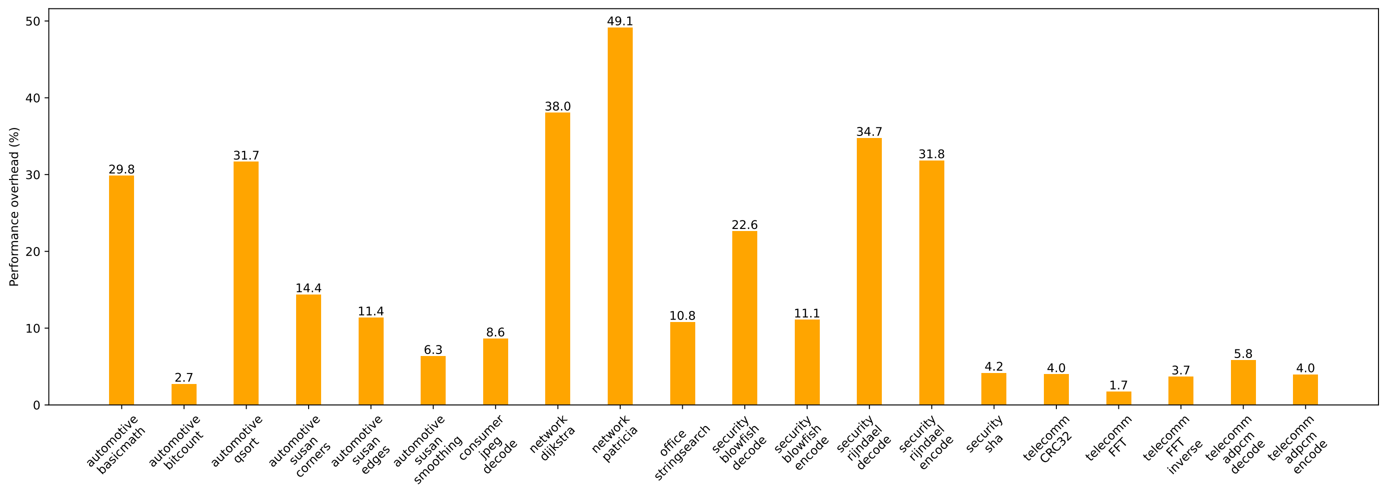

Figure 3. Overhead created by the individual MiBench benchmarks

mode. We described the common problems we encountered during the CHERIfication of software projects and how we solved them in our port. A security and performance evaluation of the complete system was performed.

For the security analysis, we used the Juliet Test Suite, containing a large variety of security tests organized by CWEs. The results show that the developed CHERI system successfully achieves spatial memory safety and, furthermore, protects against some other flaws that eventually also result in spatial memory corruptions. Nonetheless, the analysis also confirmed that there are still some language runtime-related issues that are not handled by the developed system or CHERI. First and foremost, temporal memory corruptions remain unsolved and while there already exist approaches to realize temporal memory safety with CHERI, CHERI's design does not seem optimal for it.

For our performance analysis, we used the benchmarks Dhrystone, CoreMark, and MiBench. The results show an average overhead of between 3.7% to 16.3%. The microarchitectural metrics suggest that the overhead is likely due to additionally executed instructions, often caused by missing libc optimizations, and less efficient cache usage. With future optimizations, this source of overhead can be reduced.

In companion work to this paper [1], we open-source what is to the best of our knowledge the first freely available CHERI port of the Linux kernel, initially for RISC-V. Linux is by far the most used operating system kernel in the world, at least in consumer equipment, and we hope this work will provide a baseline or at least inspiration for further evolution and research in the area, until the day CHERI hardware is widely available and CHERI Linux is mature enough to be considered for up-streaming.

References

[1] Huawei, "Linux CHERI RISCV project," https://github.com/cheri-linux, 2022, accessed: 2022-09-01.

[2] R. N. Watson, S. W. Moore, P. Sewell, and P. G. Neumann, "An introduction to cheri," University of Cambridge, Computer Laboratory, Tech. Rep., 2019.

[3] U. of Cambridge, "Capability Hardware Enhanced RISC Instructions (CHERI)," https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/, 2022, accessed: 2022-09-01.

[4] U. I. S. C. F. (ISCF), "The Digital Security by Design (DSbD) Initiative," https://www.ukri.org/what-we-offer/our-main-funds/industrial-strategy-challenge-fund/artificial-intelligence-and-data-economy/digital-security-by-design-challenge/, 2022, accessed: 2022-09-01.

[5] J. Woodruff, R. N. 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)*. IEEE, 2014, pp. 457–468.

[6] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie *et al.*, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 20–37.

[7] R. N. Watson, J. Woodruff, M. Roe, S. W. Moore, and P. G. Neumann, "Capability hardware enhanced risc instructions (cheri): Notes on the meltdown and spectre attacks," University of Cambridge, Computer Laboratory, Tech. Rep., 2018.

[8] NSA Center for Assured Software, "Juliet Test Suite v1.2 for C/C++ User Guide." [Online]. Available: https://samate.nist.gov/SARD/downloads/documents/Juliet_Test_Suite_v1.2_for_C_Cpp_-_User_Guide.pdf

[9] P. E. Black, "Juliet 1.3 test suite: Changes from 1.2." [Online]. Available: http://nvlpubs.nist.gov/nistpubs/TechnicalNotes/NIST.TN.1995.pdf

[10] H. Xia, J. Woodruff, S. Ainsworth, N. W. Filardo, M. Roe, A. Richardson, P. Rugg, P. G. Neumann, S. W. Moore, R. N. M. Watson, and T. M. Jones, "CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, pp. 545–557.

[11] N. Wesley Filardo, B. F. Gutstein, J. Woodruff, S. Ainsworth, L. Paul-Trifu, B. Davis, H. Xia, E. Tomasz Napierala, A. Richardson, J. Baldwin, D. Chisnall, J. Clarke, K. Gudka, A. Joannou, A. Theodore Markettos, A. Mazzinghi, R. M. Norton, M. Roe, P. Sewell, S. Son, T. M. Jones, S. W. Moore, P. G. Neumann, and R. N. M. Watson, "Cornucopia: Temporal Safety for CHERI Heaps," in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 608–625.

[12] R. P. Weicker, "Dhrystone benchmark: Rationale for version 2 and measurement rules," vol. 23, no. 8, pp. 49–62.

[13] S. Gal-On and M. Levy, "Exploring CoreMark™ — A Benchmark Maximizing Simplicity and Efficacy." [Online]. Available: https://www.eembc.org/techlit/articles/coremark-whitepaper.pdf

[14] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. 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)*. IEEE, pp. 3–14.