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 practice, we have ported a few of the OS kernels and base systems today used 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 run-time. 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 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 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 the Morello ARM platform.

I. INTRODUCTION

This port of Linux to the CHERI (RISC-V) was developed to validate the performance and security properties of CHERI also for Linux, which is the most used OS kernel today, especially in consumer and cloud. The “CHERIification” 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 necessary changes in the program loader, acting on changes in the ELF format. The changes were needed to manage the capability-formating 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 CHERIified 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 CHERIification 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 MUSL libc and a few applications (busybox, ssh) 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, 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 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, 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, backed up by a memory tagging architecture, between capabilities stored in memory and other types of data. 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. Excellent introduction and in-depth material for learning about CHERI is available through research papers and technical reports from University of Cambridge (e.g. [2], [5], [6] and [7]) — these constitute a starting point for learning about CHERI and 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 e.g. 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. This in addition to detailed reference manuals for their ISA. 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 libc 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 memory protection in the form of allocation boundary checks with capabilities is applied to all software parts of the system.

IV. MODIFYING THE LINUX KERNEL FOR CHERI

The CHERI Clang/LLVM compiler from Cambridge University successfully compiles most of Linux kernel source code, configured to 64bits RISC-V architecture, without any changes. In our current work, only some 200 files have undergone some modification or addition 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 in perfectly acceptable in legacy 64-bit systems. The template pattern / code change to fix these problems often involves using uintptr_t, type-defined to a Cheri Clang/LLVM compiler build-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).

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 has 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 are changed under arch/riscv/include/asm/ most notably on 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 memory.c, memmove.S, memset.S, and 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 modified the instructions (that were necessary for the platforms (qemu and FPGA) that 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 a flavor of the 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 an textbook example of such an issue found in arch/riscv/include/asm/atomic.h:

```c
static inline void *mb_correct_addr_and_bit(int *bit, void *addr)
{
    if (BITS_PER_LONG == 64)
        *bit += ((unsigned long) addr & 7UL) << 3;
    else
        *bit += ((unsigned long) addr & 7UL);
return addr;
}
```

This function has a bug: the second parameter (addr) is cast to an unsigned long and its address 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:

```c
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, this kind of modification need (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 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 build-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 unsigned integer, allowing the capability to be de-referenced. The following compiler build-in functions are used for this:

```c
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

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

to encapsulate this operation, and we use this in the porting where-ever 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

```c
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;
}
```

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 (below) to internationally 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

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

In general, we reuse the function 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 allocation 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

```c
static int 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)
{
    int user *p = (int __user *)arg;
    int cnt;
    ...
```

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:

```c
#ifdef CONFIG_CPU_CHERI PURECAP
    return object;
#else
to cheri_csetbounds(object, s->size);
#endif
```

where cheri_csetbounds is a macro defined to set the upper bound of a capability, using the compiler builtin function _builtin_cheri_bounds_set((x), (y)) to provide the provenance.

In our port of the MUSL C-library, in spirit 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.

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 re-written 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

```c
static long random_ioctl(struct file *f, unsigned int cmd,
    unsigned long arg)
{
    int __user *p = (int __user *)arg;
    int cnt;
    ...
```

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

cconst 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_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

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 use 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 exacting 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 codes listed above. 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 address 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:

c1b t2., (ca0)
csb t2., (ca0)
cincoffset ca1, cal, 1
cincoffset ca0, ca0, 1
bltu al, t0, 1b

Region 2:

c1c clt2., (ca1)
csc clt2., (ca0)
cincoffset ca1, cal, CHERICAP_SIZE
cincoffset ca0, ca0, CHERICAP_SIZE
bltu al, t1, 2b

We also implemented similar handling for lib/sort.c.
In order to maintain proveniences of capabilities, we added a function void swap_words_128(void *, 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 ‘’ ending, and to optimize performance, they iterate based on the size of an unsigned long instead of char, looking for ‘’ 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 problems in lib/strings, lib/strncpy_from_user.c and lib/strncmp_user.c, below is a typical case of such an optimization:

Listing 13. Type misuse causing boundary violation

while (max >= sizeof(unsigned long)) {
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);
  ...
}
In such cases we typically made a work-around by disabling the optimization and resorting to reading and writing one byte at time.

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) by CHERI load/store instructions accessing memory via capabilities. Most of these changes are located in the arch/riscv/include/asm/ directory, arch/riscv/head.S which represents the execution flow of very early kernel startup and arch/riscv/entry.S which defines exception entry and exit.

To illustrate such changes, we use arch/riscv/head.S as an illustration. 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_reloc 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

```c
#include <asm/byteorder.h>

unsigned long
zero_bytemask(unsigned long data);

return res + find_zero(data);

/* (unsigned long | (dest+res) = c &
zero_bytemask(data);
return res + find_zero(data);

In such cases we typically made a work-around by disabling the optimization and resorting to reading and writing one byte at time.

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) by CHERI load/store instructions accessing memory via capabilities. Most of these changes are located in the arch/riscv/include/asm/ directory, arch/riscv/head.S which represents the execution flow of very early kernel startup and arch/riscv/entry.S which defines exception entry and exit.

To illustrate such changes, we use arch/riscv/head.S as an illustration. 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_reloc 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

```c
#include <asm/byteorder.h>

unsigned long
zero_bytemask(unsigned long data);

return res + find_zero(data);

/* (unsigned long | (dest+res) = c &
zero_bytemask(data);
return res + find_zero(data);

```c
#include <asm/byteorder.h>

unsigned long
zero_bytemask(unsigned long data);

return res + find_zero(data);
```

Above is the original _start_kernel function, which calls setup_vm, and subsequently calls relocation function to switch kernel from physical to virtual address operation. When executing in Cheri RISC-V target, the setup_vm call will fail, because it further calls other C functions before the __cap_reloc 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

```c
#define CONFIG_MMU
#define CONFIG_CPU_CHERI
la a0, __early_pg_dir
XIP_FIXUP_OFFSET a0
```c
#define CONFIG_MMU
#define CONFIG_CPU_CHERI
la a0, __early_pg_dir
XIP_FIXUP_OFFSET a0

Later we would call:

Listing 16. Activating capability tables

```c
cllc cra, init_cap_relocs
cjar cra
```
call relocate
```c
#define CONFIG_MMU
```c
#define CONFIG_CPU_CHERI
la a0, __early_pg_dir
XIP_FIXUP_OFFSET a0

Here, init_cap_reloc 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.

VI. INITIAL BENCHMARKING

In the context of this work, we have collaborated with an external research institute, and we hope that we can together publish a more consistent and overarching view on CHERI-RISC-V overheads for Linux, including security testing of the result.

Until then, we have performed a few tests on Linux running on the CHERI Flute core on a Xilinx Virtex Ultrascale (VCU 118) FPGA board, with and without CHERI compiled for Linux in different configurations. The CHERI compilations have been done with a compiler from Cambridge University that uses PCC linkage for calling local functions, which significantly reduces the need to consult capability tables, therefore speeding up the end result. Kernel boot-up, when the kernel is compiled in hybrid mode, i.e. supporting CHERIified userspace processes, but not using CHERI memory protection for itself is 0%, which is to be expected, since the difference between the two kernels is mainly the inclusion of CHERI support in scheduling. However, when kernel memory protection is
turned on, the boot-up time extends from 4.6s to 5.4, i.e. a 17% increase. Comparable measurements for full device boot-up for our MUSL-based run-time is around 50%, but these measurements are still unstable, so we will defer the final verdict to future testing on those.

Another aspect that explains the relatively larger overhead of CHERIfication on the application side is the amount of code size increase for different components that today are part of the open-sourced Linux CHERI. Where the kernel code size only grows by 2.5% when compiled and measured as above, applications such as busybox and MUSL libc experience code size increases of 17.4% and 24.2% respectively. We have not analyzed the difference in depth, but application code is more data-driven, i.e. our assumption is that the relatively larger use of memory buffers and memory accesses that needs protection in user space is a main cause of these differences – more CHERI commands are used, therefore executing them also takes more time.

VII. CONCLUSION

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