Assessing the Viability of an Open-Source CHERI Desktop Software Ecosystem

PUBLIC FINAL REPORT

Version 1

Robert N. M. Watson, Ben Laurie, and Alex Richardson

Capabilities Limited

17 September 2021

InnovateUK PROJECT NO: 107145

Supported by UKRI’s Digital Security by Design Research Programme.

This page intentionally blank

Abstract 5
1. Introduction 7
2. Team 8
3. Background 8
CHERI C and C++ 9
CheriBSD and CheriABI 9
Adapting software to CHERI C and C++ 9
Challenge: Idiomatic C code confusing pointers and integers 10
Challenge: Language runtimes 10
4. Approach 10
5. Prototype CHERI-enabled desktop stack 12
Build and emulation environments 12
Baseline operating-system stack 13
Display server 13
Display and UI-facing software libraries 14
Desktop 14
Applications 15
Excluded software packages 15
6. CHERI C/C++ compilation 17
Initial GUI prototype: Qt applications via X11 SSH forwarding 17
Porting complexity 17
Summary 18
Full-screen GUI using remote-desktop solutions (XVNC) 19
XVNC 19
IceWM minimal desktop on top of XVNC 20
Summary 21
Full KDE Plasma desktop on top of XVNC 22
Qt frameworks 23

KDE frameworks 24

Plasma desktop and applications 25

Summary 26

Complete changes summary 27

7. Compiler improvements 28

-Wshorten-cap-to-int 28

CHERI UndefinedBehaviorSanitizer 29

8. Compartmentalization whiteboarding 29

Compartmentalization example 1: QImageReader 31

Compartmentalization example 2: Okular's document renderer 31

Compartmentalization example 3: KFileMetadata 32

9. Security evaluation 33

Information sources 33

Advisory and vulnerability descriptions 34

Severities 34

Threat model 34

Privilege escalation due to arbitrary code execution or file modification 35

Private data disclosure 35

Undesired data modification 35

Denial of service 35

Mitigation 36

Memory safety 36

Compartmentalization 36

Depth of analysis 37

X.org 37

Qt 40

KDE 41

Other libraries 45

10. Desktop demonstration narrative 48

11. Constructive plan 49

Memory safety 49

Compartmentalization 50

12. Limitations of this study 51

Software stack 51

Software adaptations 51

Compartmentalization sketches 52

Vulnerability information 52

Vulnerability analysis 53

13. Reproducing our results 53

14. Recommendations for future research 54

15. Related work 55

16. Acknowledgments 56

17. Conclusion 57

18. References 57

Appendix A: full list of build targets for the desktop 60

Abstract

This report describes a three-month pilot study to assess the feasibility and value of deploying CHERI memory protection and CHERI software compartmentalization in an open-source desktop software stack. The context for this work is Arm's forthcoming Morello processor, SoC, and board, which is the first industrial grade implementation of CHERI, and has the hardware facilities and performance to be suitable for desktop use.

We experimentally compile a significant open-source stack using memory-safe CHERI C/C++, validating the work in emulation, and perform a series of whiteboarding exercises to explore potential applications of software compartmentalization. We evaluate the results with respect to source-code disruption and a retrospective study of the vulnerability of selected components in that stack. We measure a 0.026% Lines-of-Code (LoC) change rate in approximately 6 million lines of C and C++ code to introduce CHERI memory safety. In our review of past vulnerabilities, we see likely mitigation rates of 91% for X11, 82% for Qt, 43% for KDE, and 100% for other supporting libraries (typically image processing).

Despite a number of limitations to this study, for example as relates to limited dynamic testing, we conclude that deploying CHERI protection in a contemporary, open-source desktop stack is feasible and offers significant value in terms of practical security. We recommend a number of avenues for future research and development.

1. Introduction

In this three-month pilot research study, part of UKRI's Digital Security by Design (DSbD) research programme, we investigated and assessed the applicability of CHERI [1, 2] memory protection and compartmentalization to an open-source desktop software stack. We adapted portions of X11, Qt, and KDE for memory-safe CHERI C/C++ compilation [3], reviewed a larger body of open-source code and its vulnerability history, and sketched a number of trial compartmentalizations of key libraries and applications. We conducted this exploration using the open-source CHERI-RISC-V and DSbD prototype Arm Morello architectures implemented on the QEMU ISA-level emulator. The forthcoming Arm Morello [4, 5] board will have both the hardware facilities and performance to enable security-critical desktop use. To date, desktop software has not been a focus of CHERI-related research, however. We used CheriBSD's CheriABI [6] pure-capability (memory-safe) process environment, with the further assumption of heap temporal memory safety via a technique such as Cornucopia [7], but the results of our work should have broad applicability as other operating systems (e.g., Linux) gain improved CHERI support. We have sought a high level of reproducibility, and include instructions in this report to build and run the resulting memory-safe software stack.

We demonstrated clear applicability of CHERI protection in hardening this open-source desktop environment, and we describe a potential implementation strategy for a larger-scale future project. We also identified minor improvements to the compiler toolchain making software adaptation easier, and gaps in understanding around software compartmentalization that may motivate future research. Our CHERI-specific changes required modifying 0.026% of 6 million C and C++ Lines-of-Code (LoC). In our review of past vulnerabilities in selected portions of the software stack, we saw likely mitigation rates of 91% for X11, 82% for Qt, 43% for KDE, and 100% for other supporting libraries (typically image processing). We describe a potential narrative for combined deployment of memory protection and compartmentalization across a broader desktop software corpus, especially with respect to larger desktop applications that are frequently built using large library stacks.

Overall, our conclusion is that CHERI and Morello have the potential to support a significantly more secure desktop ecosystem, with relatively low developer burden in adapting most existing software. This is especially true for CHERI memory protection, which appears to address a large proportion of past vulnerabilities, while requiring relatively little code change. Compartmentalization also contributes substantially to resolving vulnerabilities key to vendor threat models, especially with respect to potential denial of service and higher-level logical bugs in applications. However, there remain important research challenges, including improving systems software (and especially operating systems) to provide easy-to-use and well-documented facilities for compartmentalization, and in terms of tooling support for compartmentalization to reduce the potential developer burden in performing that work. It is also clear that this work, given additional time, could have been taken substantially further to provide greater certainty of results; for example, there is the opportunity to take a more concrete adversarial approach in further analyzing vulnerabilities and their potential mitigations. We attempt to document limitations of the study in some detail.

2. Team

Capabilities Limited is a UK-based consultancy providing technical expertise and software development services relating to open-source software and security:

Dr Robert Watson is a Director of Capabilities Limited, and also Reader in Systems, Security, and Architecture and the Principal Investigator (PI) leading development of CHERI at the University of Cambridge and SRI International. He is a strong advocate of, and contributor to, open-source software, including being a FreeBSD developer, and board member (and past President) of the FreeBSD Foundation.

Ben Laurie is a Director of Capabilities Limited, and also technical lead in security in Google Research. He has played a significant role in creating a number of key open-source projects including the Apache Software Foundation and OpenSSL, and has been involved in the creation of the CHERI ISA.

Dr Alex Richardson is a technical staff member at Capabilities Limited, and also a Senior Research Software Engineer (SRSE) at the University of Cambridge. His PhD dissertation developed key ideas and prototypes for the CHERI C/C++ programming language and linkage models. He is also a contributor to the FreeBSD, LLVM, and KDE projects.

The Capabilities Limited project team collaborated over a three-month period in mid-2021, at approximately one full-time equivalent (FTE) level of effort collectively (totaling three staff months), to investigate and analyse an open-source graphics and application stack to assess the viability of applying CHERI protections to it.

3. Background

Developed by SRI International and the University of Cambridge, **CHERI** (Capability Hardware Enhanced RISC Instructions) is a computer processor architecture protection technology supporting the implementation of fine-grained referential, spatial, and temporal memory protection, as well as enabling scalable software compartmentalization [1]. The CHERI protection model has been applied to multiple Instruction-Set Architectures (ISAs) including 64-bit MIPS, 32- and 64-bit RISC-V, and 64-bit ARMv8-A, known respectively as CHERI-MIPS, CHERI-RISC-V, and Arm Morello.

The **Arm Morello board**, processor, and System-on-Chip (SoC), a Digital Security by Design technology, ships in early 2022. The Morello SoC is an experimental high-performance, multi-core, multi-GHz design that includes a GPU, and will be the first platform suitable to use as a CHERI-extended desktop system. Although the Morello board was not yet available at the time this work was done, the architecture specification, emulator, and software stack are usable and they offer reasonable confidence that results from our work will apply to production boards. Lessons learned from the Morello project are expected to influence subsequent editions of the ARM architecture.

CHERI C and C++

The **CHERI C** and **CHERI C++** programming languages utilize CHERI's **architectural capabilities** to implement and protect language-level pointers and the data to which they refer, as well as sub-language data structures such as the stack, GOTs (Global Offset Tables, used to access global variables), and other portions of the language runtime [3]. This is referred to as **pure-capability code**, as all pointers, explicit and implied, are represented as capabilities rather than integers. CHERI C and C++ are implemented by the CHERI-extended **CHERI Clang/LLVM** compiler suite [8]. This provides strong referential safety (protecting pointers), spatial safety, and, with the addition of **Cornucopia** [7] or related techniques, heap temporal safety, which we take for granted in our retrospective vulnerability analysis. Stack temporal safety is not provided by current CHERI hardware and software prototypes, and is not assumed in this work.

The degree to which off-the-shelf C and C++ software can simply be recompiled to achieve higher levels of memory safety is an ongoing topic of research, as CHERI C and C++ have evolved substantially since earlier studies were completed, and compatibility is sensitive to idiomatic programming styles. For example, code that makes correct use of `uintptr_t` or primarily relies on C++ programming idioms will tend to adapt to CHERI without substantial (or any) modification. On the other hand, language runtimes using pointer compression techniques, or older source bases using the `long` data type to hold pointers, may require more invasive modifications.

CheriBSD and CheriABI

CheriBSD [9], a CHERI-extended version of the open-source FreeBSD operating system, implements a pure-capability **CheriABI** process environment able to execute CHERI C/C++ code [6]. Under CheriABI, the kernel, run-time linker, and system libraries cooperate to support the compiler in implementing strong and fine-grained memory protection. To date, the primary software adaptation focus has been low-level systems software such as the FreeBSD userspace, OpenSSH daemon, PostgreSQL database, nginx web server, and so on. CheriBSD contains experimental support for CHERI temporal memory safety via Cornucopia, and also for software compartmentalization, although that remains an active area of research [10]. It is also possible to compile the CheriBSD kernel itself as CHERI C, providing strong spatial memory safety for the kernel.

Adapting software to CHERI C and C++

In general, adaptation of contemporary C and C++ source code to CHERI C and C++ is straightforward, requiring occasional minor improvements in C type use (e.g., to deconflate integer and pointer values) detected by the compiler, or sometimes dynamic issues such as insufficiently strong alignment in custom memory allocators. Adaptation will often turn up minor memory-safety issues, such as buffer overflows missed by other debugging tools for various reasons. However, some types of software may require substantial work to adapt.

Challenge: Idiomatic C code confusing pointers and integers

Low-level C code often embeds assumptions about the interchangeability of pointers and integers. Such use often dates from the late 1990s or earlier, prior to standardization of the `intptr_t` type -- an integer type able to hold a pointer value (which is implemented as a capability in CHERI C/C++). This is a generally well understood area in CHERI research, and examples include some low-level systems software as well as some language runtimes. Improvements to use the more appropriate type can be some work to develop and test, but can often be readily upstreamed to open-source projects even without the specific motivation of CHERI. We selected a broad corpus of C and C++ code to evaluate against, including low-level (and generally older) C-language libraries such as giflib and libpng, as well as higher-level (and generally more recent) C++ library and application code such as Qt and KDE.

Challenge: Language runtimes

Language runtimes can present a more serious adaptation to CHERI C/C++ if they:

- Are highly aware of, and specifically target, machine code on current architectures, such as Just-In-Time (JIT) compilers, which will require a new architecture target to be developed;
- Employ pointer compression techniques to minimize existing pointer-size overheads (common with highly optimized runtimes), which interact poorly with CHERI pointers and also may limit the use of CHERI pointers due to their size; or
- Are fundamentally structured around enabling arbitrary code execution and flexible use of C types and memory, and hence may require some adaptation to run with CHERI, and further work to harden it using CHERI features.

Due to the complexity of extending language runtimes for CHERI support, we have limited our work in this project to a single instance: the QML language runtime used by Qt for user interface description and component interconnection. There has been other work on adapting language runtimes to CHERI, including Apple’s WebKit; however, the size and scope of this project did not permit exploring CHERI adaptation to, for example, the QtWebEngine JavaScript interpreter. We consider the adaptation of additional language runtimes to CHERI an essential area for future research, given their security criticality and frequent vulnerabilities.

4. Approach

Our objective was to assess the viability of extending an existing open-source desktop software environment to make extensive use of CHERI C/C++ memory protection and CHERI software compartmentalization for the purposes of vulnerability mitigation. We explored the potential impact of CHERI via two approaches:

1. We pursued selected prototyping case studies in CHERI C/C++ adaptation of software components at each layer in the stack, including the X11 window server and Qt/KDE desktop environment.

2. We manually inspected selected software for potential compartmentalization opportunities, exploring (on a whiteboard) the impact on software structure and security.

In both cases, the aim was to gather evidence required to plan a larger project to more completely implement a CHERI-enabled desktop environment suitable for the Morello board. We therefore considered several evaluation criteria to help us understand the potential costs of adaptation, and also the potential benefits:

1. **Ease of software adaptation** -- identifying less easily adapted idiomatic C use (e.g., in older software or in language runtimes), architectural awareness (e.g., in JITs or other highly optimized software packages requiring non-trivial change), and the extent of natural encapsulation opportunities for compartmentalization. A key concern was the extent to which changes to C and C++ were CHERI-specific or not -- and if not, whether they could be upstreamed to the underlying open-source project as acceptable improvements in source-code quality.

2. **Security impact** -- as judged with respect to potential mitigation of past software vulnerabilities for the target software stacks, relating to both memory safety and compartmentalization. This requires characterizing software threat models, which are often not documented explicitly; we instead look to past vulnerability announcements to understand the *de facto* threat model for each piece of software. Here, whiteboarding exercises (based primarily on vendor security analyses) are used, due to the limit on available time.

3. **Potential for performance overhead** -- QEMU and the Arm Morello FVP are not suitable for performance studies. However, memory protection can have measurable overhead, especially in pointer-dense workloads (e.g., language runtimes). Further, potential placement of compartmentalization boundaries is a performance-sensitive activity. We must therefore estimate the acceptability of introducing compartmentalization boundaries.

One technical challenge lay in scaling our work to a very large open-source software corpus. Anywhere manual inspection or prototyping is required, we were necessarily resource constrained. Modest enhancements to compiler warnings and dynamic checking, described later in this report, assisted substantially in our work, and have now been upstreamed to the CHERI LLVM project, or submitted for review for future inclusion.

Another technical challenge lies in the difference between the software techniques used in low-level systems software, where CHERI experience is strong, versus in higher-level desktop applications, where there is limited experience. Extensive use of C++, embedding of language runtimes, and IPC-linked components limit the scope of our experiments given constrained time. It is likely that there will be the opportunities for significant wins from CHERI deployment in that software (e.g., CHERI IPC performance improvements), and also new challenges (e.g., understanding how CHERI can protect language runtimes for which a key design goal is controlled arbitrary code execution), which we are not able to fully explore in this project.

Where we made changes to open-source software stacks that were not CHERI-specific -- e.g., making better use of C integer and pointer types, or to correct memory-safety violations

-- we attempted to upstream those changes. This offers us some measure of the acceptability of CHERI-motivated (but not CHERI-specific) changes in these open-source communities. We did not attempt to upstream any CHERI-specific changes (e.g., setting capability bounds in custom memory allocators, etc.), as CHERI remains a research technology. Prior CHERI adaptation efforts, especially those involving low-level C/C++ runtime software, have sometimes benefited from a middle category of changes -- e.g., restructurings to ease adoption of CHERI while neither being CHERI-specific nor correcting adherence to language specifications. However, the comparatively high-level software components studied here did not generally require changes along these lines. Where we have successfully upstreamed patches, we have indicated them in the tables and/or footnotes below; this was the vast majority of patches other than those avoiding dependencies not yet available on CheriBSD.

5. Prototype CHERI-enabled desktop stack

The open-source community has developed a number of complete GUI software stacks consisting of window servers, class libraries, and application stacks. For the purposes of this research, we have selected a specific vertical stack including the X.org window system and Qt/KDE-based desktop environment, along with dependent libraries. We selected this stack for multiple reasons, including its accessibility as a stack, spread of functionality, and use of C and C++ across various layers in the stack. In this section, we elaborate our choice of baseline stack, as well as the supporting build, emulation, and OS environments we used in our experimentation. Appendix A contains a complete list of adapted software packages.

Build and emulation environments

All software and demonstrations were built using the cheribuild build framework,[1] and compiled with CHERI Clang/LLVM for CHERI-RISC-V[2] and Morello.[3] We developed our software adaptations on CHERI-RISC-V, and also tested them with Arm Morello. For emulation of both architectures, we used the QEMU-CHERI emulator developed by the University of Cambridge [11]. We used versions of these tools from their respective development trunks as of July 2021.

The host environment for our experiments was macOS, used with the XQuartz X11 display server 2.8.1 (for X11 SSH-forwarding of individual applications) and TigerVNC VNC client 1.11.0 (for full-screen desktop display). However, none of this project depends on a macOS host environment, and we also tested cross-compilation from Linux systems.

Software module(s)	Description
CHERI and Morello Clang/LLVM toolchain	Existing macOS-/Linux-/FreeBSD-hosted cross-compiler for CHERI C/C++
cheribuild	Existing tool to simplify and automate building and

[1] https://github.com/CTSRD-CHERI/cheribuild
[2] https://github.com/CTSRD-CHERI/llvm-project
[3] https://git.morello-project.org/morello/llvm-project

	testing (cross-compiled) projects. We have extended it as part of this project to support building and testing of the CHERI desktop stack.
QEMU-Morello	Existing macOS-hosted emulator
QEMU-CHERI-RISC-V	Existing macOS-hosted emulator

Baseline operating-system stack

All work was targeted at the CheriBSD operating system (OS) developed by SRI International and the University of Cambridge. We used the CheriABI pure-capability process environment on CheriBSD [6], and also utilized a pure-capability CheriBSD kernel configuration, with the aim of achieving full-stack C/C++ memory safety. CheriBSD includes a full suite of low-level system libraries and utilities compiled as pure-capability CHERI C/C++ code for the CheriABI process environment; these did not require extension for our work. The CheriBSD development trunk as of July 2021 was used for this project.

Software module(s)	Description
CheriBSD libraries	Existing pure-capability libraries (C/C++)
CheriBSD kernel	Existing pure-capability OS kernel (C)

CheriBSD currently offers the most mature execution environment for pure-capability CHERI C/C++ code, having a cross-development toolkit, unified build system, tightly integrated CHERI support including the CheriABI process environment, and also a large suite of CHERI adapted libraries. However, it is our expectation that the vast majority (if not all) of the adaptation we performed on higher-level application components, from the display server through to KDE itself, is applicable to a future mature CHERI adaptation of the Linux operating system (e.g., with CheriABI support).

Display server

Display servers are programs that allow applications to render output to the user display, and accept input via devices such as keyboards and mice. To test desktop applications, we made use of the XVNC display server, which creates a virtual X Windows (X11) display for the virtual machine hosted within QEMU. The output of the display is forwarded over a socket to a VNC client running on the host system (in this case the macOS version of TigerVNC 1.11.0). We adapted the XVNC display server for CHERI C/C++ compilation; we used the XVNC development trunk as of July 2021 and compiled it against the X11 XServer 1.20 stable branch as of July 2021.

Software module(s)	Description
XVNC (TigerVNC server)	Newly adapted pure-capability remote display server (mostly C, some C++)

XServer (common code used by XVNC)	Newly adapted pure-capability remote display server (C)
TigerVNC client	Unmodified VNC client to display the QEMU output under macOS

Display and UI-facing software libraries

Contemporary desktop applications rely on an extensive library stack to communicate with the display server, render output, accept input, compute, interact with other applications via shared services such as copy-and-paste, and use I/O services such as networking and storage. We adapted the K Desktop Environment (KDE) and its library dependencies, including Qt and X11, for CHERI C/C++ compilation, as well as investigating them for compartmentalization opportunities. For the Qt libraries, we compiled the 5.15 LTS stable branch and for all other libraries and programs we used the latest git snapshot as of 30th July 2021.

Software module(s)	Description
KDE frameworks libraries	Newly adapted pure-capability libraries (C++)
Qt class libraries	Newly adapted pure-capability libraries (C++)
X11 libraries	Newly adapted pure-capability libraries (C)
Other supporting libraries (e.g. libjpeg-turbo, poppler, fontconfig, freetype2).	Newly adapted pure-capability libraries (C/C++)

Desktop

A typical open-source UNIX desktop environment consists of a window manager/compositor providing features such as window decorations, as well as a desktop shell that provides interactive elements such as the start menu, desktop background, and system tray. For our demonstrator system we initially adapted the minimal IceWM desktop due to the low number of dependencies. We then expanded our scope to include the full KDE software stack including the feature-rich Plasma desktop shell. As with the KDE frameworks, we use the latest git snapshots from July 2021.

Software module(s)	Description
KWin window manager	Newly adapted pure-capability application (C++)
IceWM minimal desktop	Newly adapted pure-capability application (C++)
Plasma desktop shell	Newly adapted pure-capability application (C++)

Applications

We adapted a number of Qt and KDE applications for CHERI C/C++ compilation, as well as investigating them for compartmentalization opportunities. As the scope of this project does not allow for porting and testing the entire KDE application stack (over 200 applications) [12], we selected a few applications that should be representative of commonly used desktop software (e.g., a file manager and viewers/editors for various file formats). Many of these applications are also interesting from a security point of view, because they often interact with content received from untrustworthy (and potentially malicious) sources. We used the latest git snapshots from July 2021.

Software module(s)	Description
Dolphin	Newly adapted pure-capability file manager (C++)
Gwenview	Newly adapted pure-capability image viewer and minimal editor (C++)
Plasma system settings	Newly adapted pure-capability application (C++)
Okular	Newly adapted pure-capability document viewer with support for many file formats such as PDF or EPub (C++)

Excluded software packages

Due to limited time and scope for this effort, we did not include a number of key software modules in this work, which would be required to implement a fully elaborated desktop environment. In the X11/Qt/KDE ecosystem, we excluded:

Software module(s)	Description
D-Bus	IPC middleware used in KDE and Gnome (C) (omitted due to project timeline)
Kate	Advanced text editor with syntax highlighting and code completion for many programming languages (C++) (omitted due to project timeline)
KMail and KOrganizer	Email client and Calendar applications (C++) (omitted due to project timeline)
Calligra and Krita	Office suite and advanced painting program (C++) (omitted due to project timeline; LibreOffice is the more commonly-used open-source office suite, but was beyond the scope of this project.)

Also, we were not able to include a number of other display and desktop-focused open-source software packages of interest beyond the X11/Qt/KDE desktop ecosystem:

Software module(s)	Description
OpenGL	Graphics rendering and acceleration framework (required for GPU acceleration) (C and C++)
X11 Panfrost userspace device driver	Userspace device driver required for Arm Mali GPUs (C)
KMS, DRM, and Panfrost kernel device drivers	Kernel device drivers required for Arm Mali GPUs (C)
Wayland[4]	Contemporary userspace display server intended to replace X11 (C)
Gnome Desktop Environment	Alternative open-source desktop stack to the KDE ecosystem (C,C++, and Vala)
LibreOffice	Open-source office package including word processor, spreadsheet, and other tools (C, C++, Java, and other languages, and also comes with a number of challenges, including using Java and its own Foreign Function Interface (FFI) mechanism that might require adaptation.)
VLC media player	Open-source media player with support for many different audio and video formats (C, C++)
Gimp	Open-source graphics package (C)
Thunderbird	Open-source mail reader (C, C++)
Chromium, Firefox	Open-source web browsers (C, C++). The CHERI project has previously created an experimental CHERI (and Morello) WebKit adaptation, although there was not yet a writeup analyzing the results of that work at the time this report was published.[5]

It is reasonable to assume that several of these uninvestigated software packages would present substantial engineering challenges due to their inclusion of language runtimes, including Just-in-Time (JIT) compilers.

Finally, we were unable to analyse or experimentally adapt proprietary desktop software packages available for FreeBSD and/or Linux, including the following indicative examples:

Software module(s)	Description
Zoom	Desktop video conferencing software
Skype	Desktop video conferencing server

[4] As an initial scoping exercise, we ported the base Wayland libraries to CHERI C without any problems; however, all Wayland display servers required significant dependencies that we could not have realistically ported within the project timeline.
[5] https://github.com/CTSRD-CHERI/webkit/

Morello-enabled operating systems such as CheriBSD and Morello Linux are able to run unmodified Arm64 applications distributed as binaries -- but are not able to offer them strong internal memory safety, since that requires, at minimum, recompilation.

6. CHERI C/C++ compilation

For all projects we report the total number of source lines[6] (counted using cloc version 1.90 [13]), as well as the number of changes made in order to support CHERI C/C++ compilation. We also show the changes that we made that were unrelated to CHERI. However, most of those are related to cross-compilation and will therefore no longer be required once Morello boards are available. The scripts we used to perform the analysis of change percentages have been made available as open-source and are available at [14].

Initial GUI prototype: Qt applications via X11 SSH forwarding

For the initial exploration of graphical applications on top of a CHERI-based operating system we decided to run X11 QtBase example applications via SSH X11 forwarding. We chose this as a first step since X11 forwarding over SSH avoids dependencies on graphics drivers and we had a pre-existing CHERI port of Qt 5.10 dating back to 2017, so the only missing pieces for this initial proof-of-concept were the X11 client libraries.

Porting complexity

Most of the X11 libraries worked out-of-the box when compiled for CHERI C, but we did have to make two changes to tell the libX11 and libXt libraries that pure-capability CHERI-RISC-V has a 64-bit `long` (due to a hardcoded list of architectures),[7] one change to allow for cross-compilation from macOS[8] and finally two kinds of changes that affect only CHERI C.

The first type of change to support CHERI involved replacing various uses of `long` with `uintptr_t` in libXt code that predates the C99 standard[9]. Most of these changes were straightforward, as the compiler flagged them with warnings. However, two of them only showed up at run time. The first was caused by the assumption that two structures have identical layouts if one of them uses pointer members and the other one `unsigned long`. This assumption does not hold for CHERI; as part of the upstreaming effort we also added compile-time assertions to catch this problem. The second issue was a SIGBUS caused by an under-aligned structure access, due to a memory buffer being aligned to only 64 bits instead of the size of a pointer (128 bits for Morello).

---

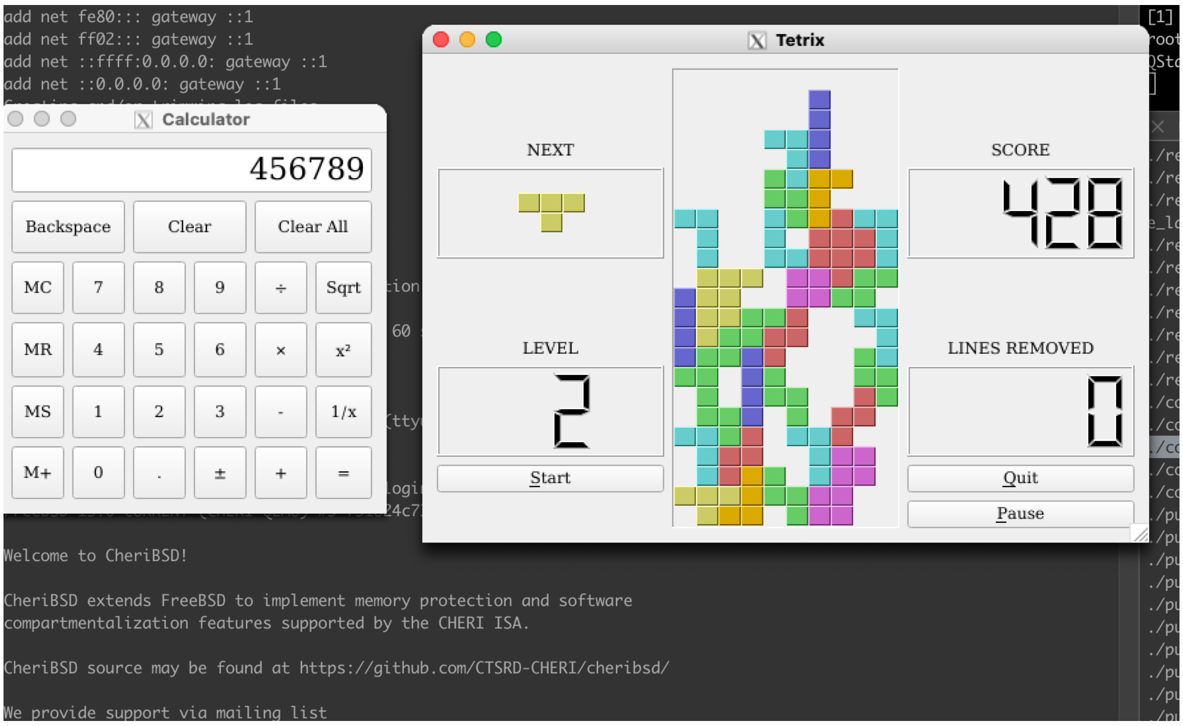
[6] We report "source" lines of code, i.e., lines of code ignoring comments and whitespace since counting whitespace and comments would represent the percentage of change as being unrealistically small.

[7] libXt: Define `LONG64` if `SIZEOF_LONG` indicates 64-bit `long` and xorgproto: Define `LONG64` if `SIZEOF_LONG` indicates 64-bit `long`

[8] libX11: Fix macOS cross-compilation

[9] libXt: Support architectures where pointers are bigger than `long`

Additionally, we discovered incorrect uses of `realloc()` that happen to work on most architectures, but create invalid or out-of-bounds pointers in CHERI C. libX11 includes at least two dynamically allocated data structures that contain pointers to the same memory allocation. These data structures can be resized using the `realloc()` function, which can result in the underlying memory being copied to another region. The existing libX11 attempts to correct the pointer values after reallocation by adding the offset between the old and new allocation to each of the contained pointers. However, this is undefined behaviour according to the C standard and with CHERI C results in a pointer that has the address of the new allocation but the bounds of the old one[10], and therefore will trigger a CHERI exception when dereferenced. The solution for this is to update pointers by deriving from the new allocation, instead of adding an offset to the old pointer. We have submitted this change upstream and it has since been accepted upstream.[11]


```
add net fe80::: gateway ::1
add net ff02::: gateway ::1
add net ::ffff:0.0.0.0: gateway ::1
add net ::0.0.0.0: gateway ::1

Welcome to CheriBSD!

CheriBSD extends FreeBSD to implement memory protection and software
compartmentalization features supported by the CHERI ISA.

CheriBSD source may be found at https://github.com/CTSRD-CHERI/cheribsd/

We provide support via mailing list
```

Qt calculator and Tetris demo running via X11 SSH forwarding on CheriBSD

Summary

Overall, it took about 8 hours to port the X11 client libraries to CHERI C/C++ and run example applications via X11 forwarding. Most of the required changes were straightforward (and partially suggested by the compiler), and therefore should not require much prior CHERI experience. However, the incompatibilities caused by `realloc()` are more subtle and would have taken developers without prior CHERI experience significantly longer to debug. A more detailed description of the debugging process to resolve these issues can be found at [16]. In later stages of this project we discovered that misuse of `realloc()` and similar problems that cause pointers to go out-of-bounds are relatively common in legacy C

[10] In general, these pointers will be so far out-of-bounds that they are no longer representable with the CHERI capability encoding scheme [15], so they will become untagged after this pointer arithmetic.
[11] libX11: Fix undefined behaviour after realloc()

codebases, so we added CHERI Clang/LLVM compiler instrumentation to diagnose them earlier (when creating an unrepresentable capability rather than on dereference). See the next section for more details.

Component	Lines of code	Language	Required adaptations
xorgproto	29 K	C	5 LoC changed (0.017%) to make the library aware that `long` is 64 bits.
libX11	114K	C	4 LoC changed (0.004%) to fix undefined behaviour after realloc and a macOS cross-compilation fix.
LibXt	34 K	C	47 LoC changed (0.138%) to make the library aware that `long` is 64 bits long and various changes replacing `long` with `intptr_t`.
Additional X11 libraries[12]	43K + 102K generated	C	Completely unmodified existing C code, the majority of it being generated code in libxcb (created from XML protocol descriptions).
QtBase (version 5.10)	1,504 K + 505K tests + 74K examples	C++	530 LoC changed in the library (0.04%) and 42 LoC changed in the test (0.008%). No changes to the examples. See [17] for the nature of changes.

Full-screen GUI using remote-desktop solutions (XVNC)

After successfully running graphical applications using X11 SSH forwarding, we moved on to running a full desktop environment on top of an XServer. As we do not yet have a usable graphics driver, we chose XVNC, an XServer implementation that sends the framebuffer contents over the network to a VNC viewer on the host. Many open-source minimal desktop solutions exist, but after an initial investigation we chose to port IceWM due to the low number of dependencies (a working XServer and X11 libraries) and the ability to run it without OpenGL and D-Bus. Running a window manager and XVNC XServer also required porting additional X11 libraries and further dependencies such as a JPEG and PNG decoder/encoder libraries and font-rendering libraries.

XVNC

When porting the newly required libraries, most incompatibilities were flagged at compile time. Firstly, we made changes to cast via `uintptr_t` instead of `long` in fontconfig,[13] freetype,[14] and libjpeg-turbo[15] to retain capability metadata. Additionally, we updated freetype2 to use C11 atomics instead of GCC's `__sync_*` builtins that do not work with

[12]This includes the following ten libraries: libXau, libxcb, libXTrans, libXext, libXFixes, libXi, libXRender, libICE, libSM, libXmu

[13] https://gitlab.freedesktop.org/fontconfig/fontconfig/-/merge_requests/190 (merged)

[14] https://gitlab.freedesktop.org/freetype/freetype/-/merge_requests/52 (merged)

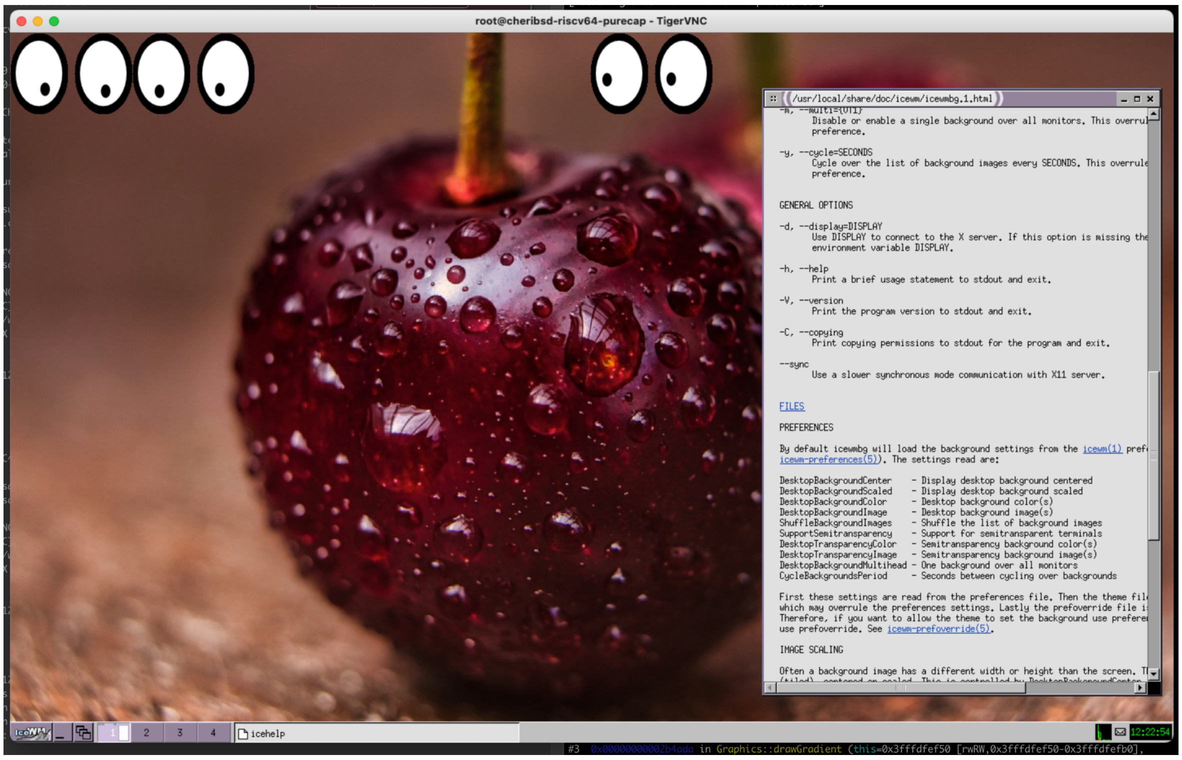
[15] https://github.com/libjpeg-turbo/libjpeg-turbo/pull/538 (merged)

CHERI capabilities in all cases.[16] Finally, we also submitted fixes to address compiler warnings that are errors by default in our toolchain[17] and support cross-compilation for FreeBSD[18]. However, some issues only showed up when we tried to start the XVNC server. The fontconfig library contained incorrect uses of `realloc()` (as described in the previous subsection) and, separately, created pointers by adding offsets to a different allocation.[19] Both of these are C undefined behaviour and will create out-of-bounds capabilities with CHERI that result in run-time crashes. Fixing these issues would have been significantly harder without our newly added CHERI Clang/LLVM checks (see the next section for details). These changes were sufficient to run X11 applications over XVNC.

IceWM minimal desktop on top of XVNC

Running IceWM on top of XVNC required only some minor build-system fixes to support cross-compilation, which have now been merged upstream.[20] As can be seen on the figure below, this minimal desktop provides a start menu, desktop background, application list (pager), and a window manager that is responsible for the window decorations. It is also possible to change the appearance of the desktop using a theming system. While testing the functionality of the desktop, we noticed that selecting one of the default themes (NanoBlue) resulted in IceWM crashing due to a CHERI capability bounds violation. As of writing this report, we have not attempted to fix this issue -- as the IceWM desktop was only an intermediate step on the way to running the full KDE desktop stack.

[16] Due to compiler implementation details it is not possible to use these builtins with a capability argument, so we have to use the newer `__atomic_*` builtins or C11 atomics instead. Fix submitted as https://gitlab.freedesktop.org/fontconfig/fontconfig/-/merge_requests/192 (merged).
[17] https://gitlab.freedesktop.org/pixman/pixman/-/merge_requests/48 (under review)
[18] https://gitlab.freedesktop.org/freetype/freetype/-/merge_requests/48 (merged)
[19] realloc() fix submitted as https://gitlab.freedesktop.org/fontconfig/fontconfig/-/merge_requests/193 (under review), but the fix for the second issue has not yet been sent upstream at time of writing.
[20] https://github.com/bbidulock/icewm/pull/601 (merged) and https://github.com/bbidulock/icewm/pull/603 (merged)


root@cheribsd-riscv64-purecap - TigerVNC
/usr/local/share/doc/icewm/icewmbg.1.html
IceWM
1
2
3
4
icehelp
12:22:54

```
-m, --multi={0|1}
       Disable or enable a single background over all monitors. This overrul
       preference.

-y, --cycle=SECONDS
       Cycle over the list of background images every SECONDS. This overrule
       preference.


GENERAL OPTIONS

-d, --display=DISPLAY
       Use DISPLAY to connect to the X server. If this option is missing the
       environment variable DISPLAY.

-h, --help
       Print a brief usage statement to stdout and exit.

-V, --version
       Print the program version to stdout and exit.

-C, --copying
       Print copying permissions to stdout for the program and exit.

--sync
       Use a slower synchronous mode communication with X11 server.


FILES

PREFERENCES

By default icewmbg will load the background settings from the icewm(1) pref
icewm-preferences(5)). The settings read are:

DesktopBackgroundCenter    - Display desktop background centered
DesktopBackgroundScaled    - Display desktop background scaled
DesktopBackgroundColor     - Desktop background color(s)
DesktopBackgroundImage     - Desktop background image(s)
ShuffleBackgroundImages    - Shuffle the list of background images
SupportSemitransparency    - Support for semitransparent terminals
DesktopTransparencyColor   - Semitransparency background color(s)
DesktopTransparencyImage   - Semitransparency background image(s)
DesktopBackgroundMultihead - One background over all monitors
CycleBackgroundsPeriod     - Seconds between cycling over backgrounds

First these settings are read from the preferences file. Then the theme fil
which may overrule the preferences settings. Lastly the prefoverride file i
Therefore, if you want to allow the theme to set the background use prefere
use prefoverride. See icewm-prefoverride(5).

IMAGE SCALING

Often a background image has a different width or height than the screen. T
```

```
#3  0x00000000002b4ada in Graphics::drawGradient (this=0x3fffdfef50 [rwRW,0x3fffdfef50-0x3fffdfefb0],
```

Minimal IceWM desktop running on QEMU and displayed on the host over VNC

Summary

Overall, porting XVNC and a minimal desktop was mostly straightforward, with the notable exception being fontconfig. Fontconfig's serialization code heavily relied on being able to create pointers from arbitrary pointer arithmetic, and this is not compatible with CHERI. Attempting to find the sources of invalid pointer arithmetic by setting breakpoints and looking at crashes due to invalid capabilities was a very challenging undertaking, so after fixing the first case, we investigated compiler instrumentation to find similar issues earlier (see the next section). Otherwise, the overall amount of changes to software was always well below 0.1% of lines changed and many projects required no changes (or only cross-compilation fixes). A summary of the changes can be seen in the table below.

Component	Lines of code	Language	Required adaptations
IceWM	68 K	C++	No changes required for CHERI, only build system fixes for cross-compilation.
XServer	200 K + 174K unused[21]	C	6 LoC changed (0.003%) to fix an incorrect use of `realloc()`[22] and to fix the -Werror build.[23]
libXFont	21K	C	4 LoC changed (0.02%) to fix a read one byte

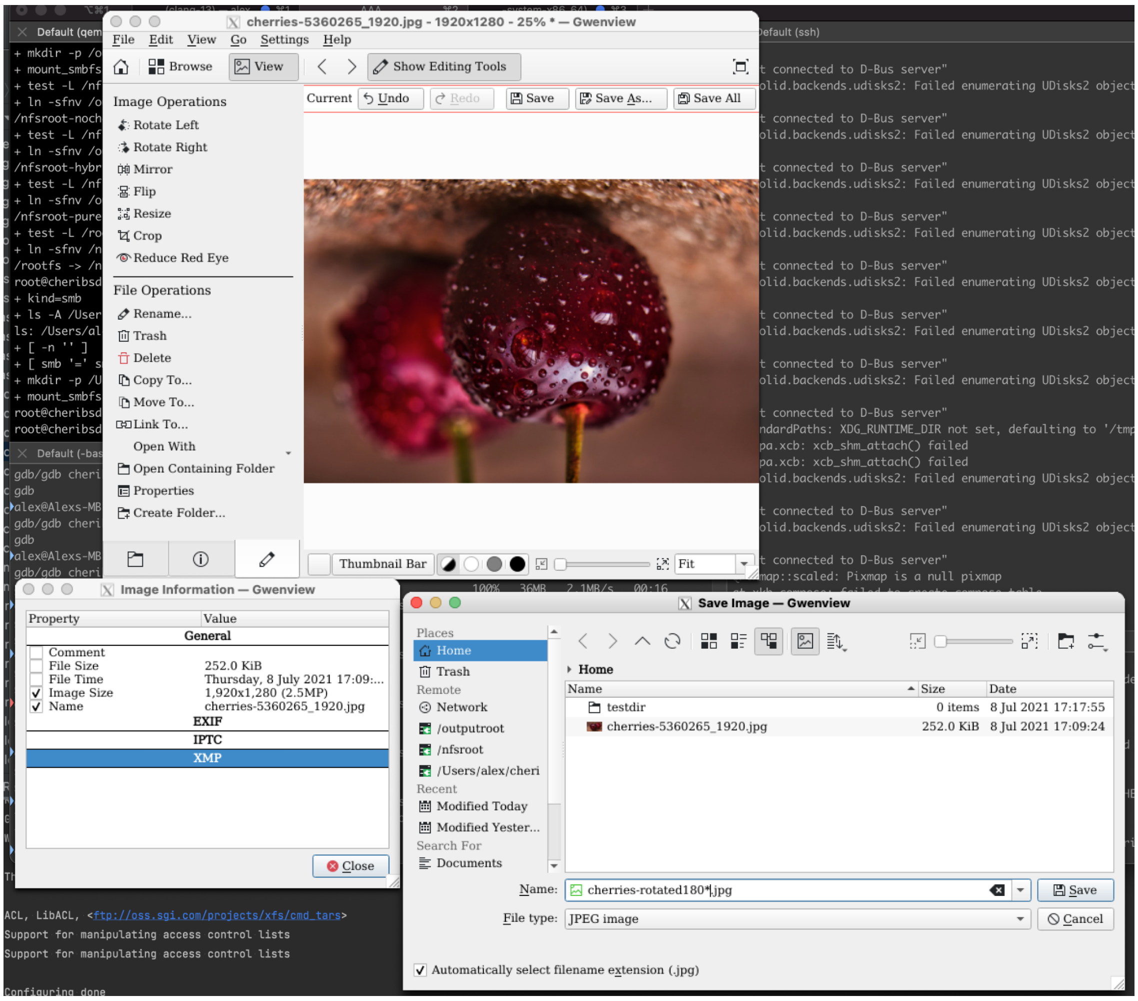
[21] This includes code for various XServer implementations such as XQuartz that were not compiled.
[22] https://gitlab.freedesktop.org/xorg/xserver/-/merge_requests/721 (under review)
[23] https://gitlab.freedesktop.org/xorg/xserver/-/merge_requests/720 (merged)

			before the start of a string.[24]
Additional X11 libraries[25]	75K	C	Unmodified
Additional X11 programs[26]	36K	C	Unmodified.
Freetype	125K	C	5 LoC changed to use `uintptr_t` instead of `unsigned long`
Fontconfig	28K	C	118 LoC changed (0.424%) to add support for C11 atomics and fix provenance issues after `realloc()` and when reading serialized data.
libjpeg-turbo	88K	C	10 LoC changed (0.011%) to cast via `uintptr_t` when adjusting alignment
libpng	57K	C	3 LoC (0.005%) changed to use `intptr_t` instead of `long` when casting.
TigerVNC	55 K	85% C++ 15% C	No changes required for CHERI, only build system fixes for cross-compilation.[27]

Full KDE Plasma desktop on top of XVNC

Compared to the initial minimal IceWM, the KDE Plasma desktop has many more features and, to provide those features, depends on a large number of libraries and additional applications. The underlying foundations of the Plasma desktop are the Qt frameworks [18] (including declarative user-interface descriptions requiring the QtQML language runtime), the majority of the KDE frameworks, and further libraries providing access to, e.g., hardware sensors or plug-and-play device notifications. After porting Qt and the KDE frameworks, we ran individual applications (e.g. Gwenview, see figure below) over SSH to verify that they worked as expected. After this was confirmed working, we moved on to porting the window manager (KWin) and the full desktop running on XVnc.

[24] https://gitlab.freedesktop.org/xorg/lib/libxfont/-/merge_requests/10 (merged)
[25] libfontenc, libxcb-cursor, libxcb-image, libxcb-keysyms, libxcb-render-util, libxcb-util, libxcb-wm, libxcomposite, libxcursor, libxdamage, libxft, libxkbcommon, libxkbfile, libxpm, libxrandr, libxtst, xbitmaps, xcbproto, xkeyboard-config, xorg-font-util, xorg-macros, xorg-pthread-stubs
[26] twm, xauth, xev, xeyes, xkbcomp, xprop, xsetroot
[27] https://github.com/TigerVNC/tigervnc/pull/1290, https://github.com/TigerVNC/tigervnc/pull/1289, https://github.com/TigerVNC/tigervnc/pull/1291 (all merged)



```
+ mkdir -p /o
+ mount_smbfs
+ test -L /nf
+ ln -sfnv /o
/nfsroot-noch
+ test -L /nf
+ ln -sfnv /o
/nfsroot-hybr
+ test -L /nf
+ ln -sfnv /o
/nfsroot-pure
+ test -L /ro
+ ln -sfnv /n
/rootfs -> /n
root@cheribsd
+ kind=smb
+ ls -A /User
ls: /Users/al
+ [ -n '' ]
+ [ smb '=' s
+ mkdir -p /U
+ mount_smbfs
root@cheribsd
root@cheribsd
```

```
gdb/gdb cheri
gdb
alex@Alexs-MB
gdb/gdb cheri
gdb
alex@Alexs-MB
gdb/gdb cheri
```

```
t connected to D-Bus server"
olid.backends.udisks2: Failed enumerating UDisks2 object
t connected to D-Bus server"
olid.backends.udisks2: Failed enumerating UDisks2 object
t connected to D-Bus server"
olid.backends.udisks2: Failed enumerating UDisks2 object
t connected to D-Bus server"
olid.backends.udisks2: Failed enumerating UDisks2 object
t connected to D-Bus server"
olid.backends.udisks2: Failed enumerating UDisks2 object
t connected to D-Bus server"
olid.backends.udisks2: Failed enumerating UDisks2 object
t connected to D-Bus server"
olid.backends.udisks2: Failed enumerating UDisks2 object
t connected to D-Bus server"
ndardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp
pa.xcb: xcb_shm_attach() failed
pa.xcb: xcb_shm_attach() failed
olid.backends.udisks2: Failed enumerating UDisks2 object
t connected to D-Bus server"
olid.backends.udisks2: Failed enumerating UDisks2 object
t connected to D-Bus server"
map::scaled: Pixmap is a null pixmap
```

Property	Value
General	
☐ Comment	
☐ File Size	252.0 KiB
☐ File Time	Thursday, 8 July 2021 17:09:...
✓ Image Size	1,920x1,280 (2.5MP)
✓ Name	cherries-5360265_1920.jpg
EXIF	
IPTC	
XMP	

```
ACL, LibACL, <ftp://oss.sgi.com/projects/xfs/cmd_tars>
Support for manipulating access control lists
Support for manipulating access control lists

Configuring done
```

Home

Name	Size	Date
testdir	0 items	8 Jul 2021 17:17:55
cherries-5360265_1920.jpg	252.0 KiB	8 Jul 2021 17:09:24

Name: cherries-rotated180*.jpg

File type: JPEG image

✓ Automatically select filename extension (.jpg)

Rotating an image in Gwenview (running on CheriBSD displayed via X11 SSH forwarding)

Qt frameworks

As part of this project we also updated the Qt 5.10 port to the latest LTS release at the time (5.15), as the KDE frameworks depend on this version,[28] and ported additional modules such as QtDeclarative and QtSvg. For QtDeclarative we also had a pre-existing port of version 5.10. However, much of the QML language runtime was changed so significantly for version 5.15 that updating to 5.15 was almost equivalent to starting from scratch.

Inside QtBase, the majority of changes were related to `QByteArray::fromRawData()`. This function creates a new `QByteArray` that references the passed argument instead of making a deep copy.If the size of the buffer is not passed explicitly this function uses `strlen()` to compute the size of the array. For CHERI this means that the trailing zero byte will not be included in the capability bounds when converting this `QByteArray` to a zero-terminated C

[28] The initial code update was undertaken at the University of Cambridge just prior to the start of this project, however, as part of this work we had to fix various issues in code paths that had not been exercised in previous experiments. Furthermore, this previous work was limited to the QtBase library and did not include components such as the QML language runtime.

string[29], so this can result in a bounds error at runtime. To fix this issue, we introduced a new function, `QByteArray::fromNulTerminatedRawData()`, that sets the size of the array to the length of the string but ensures that the trailing zero byte is included in the capability bounds.

The other library that required non-trivial changes was QtDeclarative, as this contains a language runtime for the QML language that is used e.g., for most of the Plasma desktop user interface. The majority of the language runtime is essentially a JavaScript engine with a few additional features. Moreover, it is quite similar to, and (as far as we can tell) partially based on, the WebKit JavaScript engine. As we have prior experience with adapting WebKit for CHERI, we were able to leverage this knowledge to port QtDeclarative to CHERI C++ within about one week rather than the much longer timeframe originally required for WebKit. During this porting effort, we encountered multiple instances of capability metadata being lost due to implicit casts to `uint64_t`. Debugging such an issue at run time can be rather difficult since the problem usually occurred a long time before the crash happens. Therefore, we developed a new compiler diagnostic, `-Wshorten-cap-to-int,` to diagnose these issues at compile time rather than through a run-time crash (see Compiler improvements).

Unlike our approach for other projects, we did not attempt to upstream our Qt changes while undertaking this project, as the Qt frameworks have moved to a new major version (6.2 at the time of writing), and many of our changes no longer apply due to major refactorings (e.g., the QString data storage implementation was almost completely rewritten) and investigating which patches can still be upstreamed was not possible due to the project timeline.

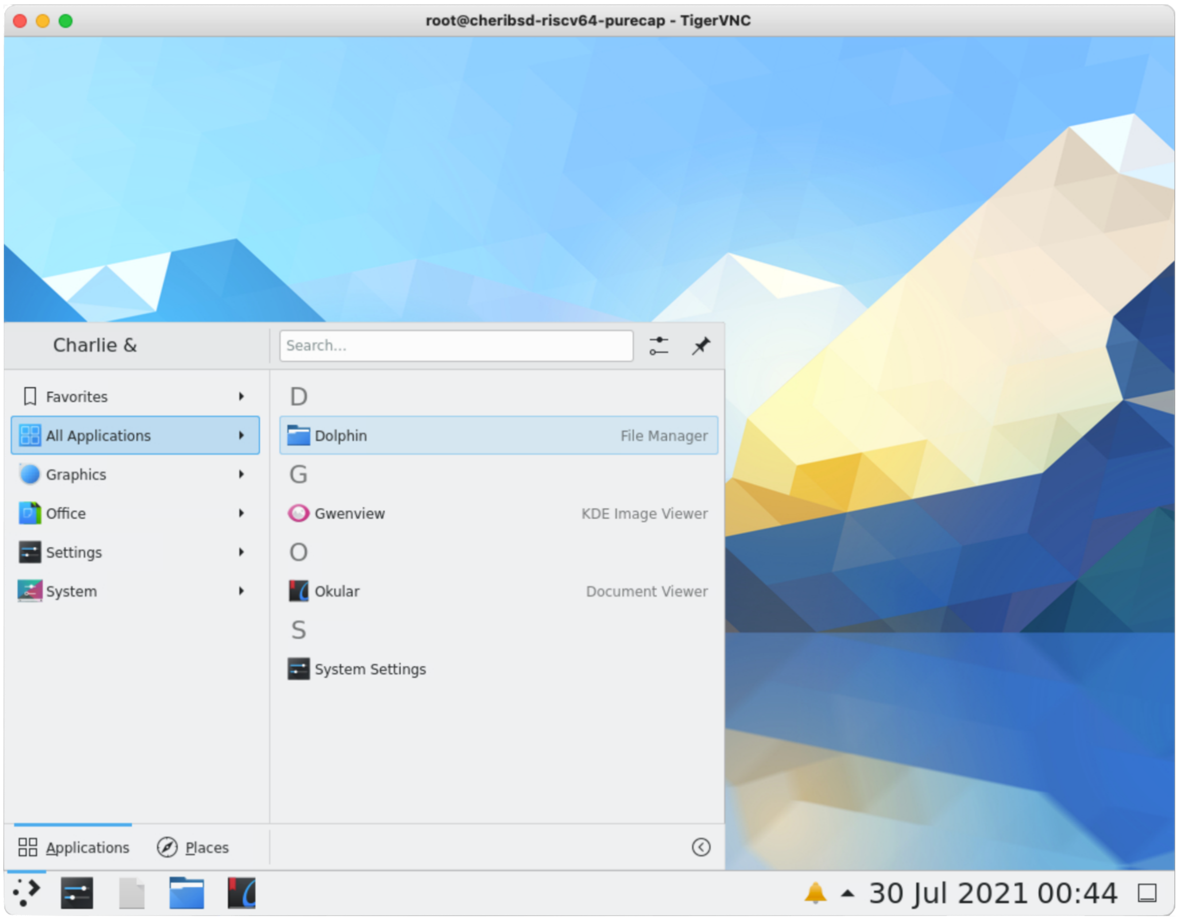
KDE frameworks

Out of the 83 KDE frameworks [19], we compiled 55 for CHERI C/C++. The remaining frameworks were not used as a dependency of either Plasma desktop or of the applications that we chose as demonstrators, so we omitted them due to the project timeline.

In terms of changes required due to CHERI, we had to make only a single one-line change to avoid a one-byte out-of-bounds read.[30] Other than this single-line change, the other changes that we made were mostly related to fixing cross-compilation issues as most KDE users only compile the software for their respective machines (the exception being cross-compilation for Android, which has slightly different restrictions compared to compiling on macOS for FreeBSD).[31] We also submitted a few changes that make certain dependencies optional (e.g., the KDocTools for documentation or OpenGL for hardware-accelerated

[29] This is only a problem when the raw data is accessed directly, since any modification of the QByteArray will result in a deep copy which always appends a zero byte.
[30] The code in question was using `strlen()` to read up to a terminating zero byte, but the `QByteArray` that was being passed had bounds that did not include the zero byte. We fixed this by using the `.size()` member instead of `strlen()` (https://invent.kde.org/frameworks/kio/-/merge_requests/493).
[31] https://invent.kde.org/frameworks/solid/-/merge_requests/44 (merged),
https://invent.kde.org/frameworks/kcoreaddons/-/merge_requests/109 (merged),
https://invent.kde.org/frameworks/kcoreaddons/-/merge_requests/110 (merged),
https://invent.kde.org/frameworks/kcoreaddons/-/merge_requests/117 (under review),
https://invent.kde.org/frameworks/syntax-highlighting/-/merge_requests/220 (merged),
https://invent.kde.org/frameworks/kpty/-/merge_requests/12 (under review)

rendering) as we do not require them for our demonstrator.[32] Similarly, we made changes to avoid a dependency on Wayland, but did not submit those changes upstream -- as a longer-term upstream goal is to default to Wayland rather than X11. Finally, we also submitted an optimization that significantly reduced application startup times on slow systems.[33]



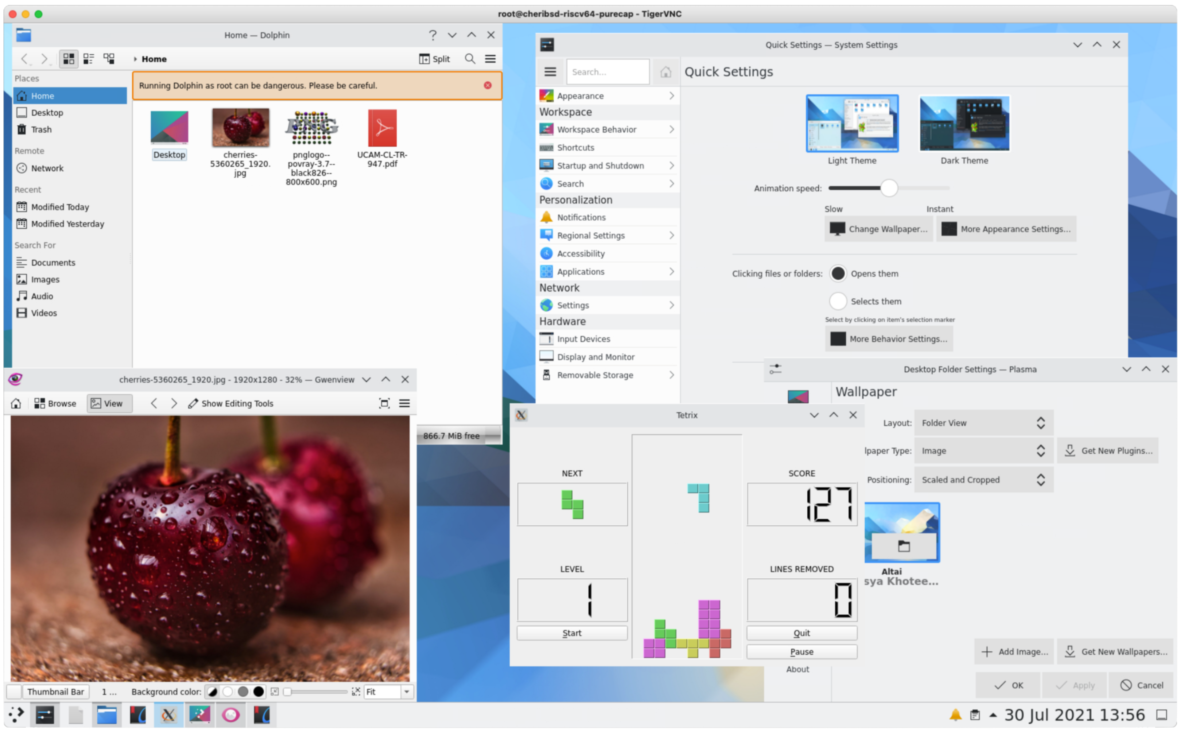
Plasma desktop shell (including the QML-based start menu) running on CHERI-RISC-V

Plasma desktop and applications

For the KDE applications, we had to make CHERI-related changes to only one application: the KWin window manager, which contained an out-of-bounds read. As this memory access could potentially have a security impact, we have reported our finding to the KDE security team, and have not yet published the fix to any open repositories.[34] Other than this, we had to make only straightforward (albeit tedious) changes to make certain dependencies optional, as we do not have them on our desktop prototype. Many applications unconditionally depended on OpenGL, DBus, Wayland, or documentation tools.[35]

[32] OpenGL: https://invent.kde.org/frameworks/kdeclarative/-/merge_requests/64 (merged),
Documentation: https://invent.kde.org/network/kio-extras/-/merge_requests/110 (merged),
QML debugger: https://invent.kde.org/frameworks/kdeclarative/-/merge_requests/65 (merged),
DBus: https://invent.kde.org/frameworks/kdbusaddons/-/merge_requests/10 (merged),
[33] https://invent.kde.org/frameworks/kcoreaddons/-/merge_requests/116 (merged)
[34] The KDE security team reviewed this change after the initial version of this report was completed and concluded that it does not warrant a security advisory. The patch has now been published and can be seen at https://invent.kde.org/plasma/kwin/-/merge_requests/1400.
[35] Optional OpenGL: https://invent.kde.org/graphics/gwenview/-/merge_requests/95

Furthermore, we encountered a long-standing bug in CMake[36] that prevented cross-compilation of the poppler PDF rendering library that is used by the Okular document viewer. We submitted a merge request to poppler to work around this issue.[37] Finally, we also upstreamed minor changes to support cross-compilation (mostly build-system related).[38]


root@cheribsd-riscv64-purecap - TigerVNC
Home — Dolphin
Running Dolphin as root can be dangerous. Please be careful.
Quick Settings — System Settings
Quick Settings
Light Theme
Dark Theme
Desktop Folder Settings — Plasma
Wallpaper
Tetrix
NEXT
SCORE
LEVEL
LINES REMOVED
cherries-5360265_1920.jpg - 1920x1280 - 32% — Gwenview
30 Jul 2021 13:56

Various desktop applications running on CHERI-RISC-V

Summary

We have submitted the majority of our changes upstream. As of writing this report, almost all of these changes have now been merged into the upstream repositories. The table below gives an overview of these changes.

Component	Lines of code	Language	Required adaptations
KIO	115 K	C++	1 line changed (0.0009% of total) to avoid an out-of-bounds read.

Optional DBus: https://invent.kde.org/graphics/gwenview/-/merge_requests/94, https://invent.kde.org/system/dolphin/-/merge_requests/231, https://invent.kde.org/plasma/kwin/-/merge_requests/1206, https://invent.kde.org/graphics/okular/-/merge_requests/460
Optional Wayland: https://invent.kde.org/plasma/kscreenlocker/-/merge_requests/41 (rejected)
Optional documentation: https://invent.kde.org/system/dolphin/-/merge_requests/230 (merged)
https://invent.kde.org/graphics/gwenview/-/merge_requests/92 (merged)
https://invent.kde.org/plasma/systemsettings/-/merge_requests/74 (merged)

[36] https://gitlab.kitware.com/cmake/cmake/-/issues/22414

[37] https://gitlab.freedesktop.org/poppler/poppler/-/merge_requests/887 (under review)

[38] https://invent.kde.org/plasma/plasma-desktop/-/merge_requests/532 (merged), https://invent.kde.org/graphics/okular/-/merge_requests/455 (merged), https://invent.kde.org/graphics/okular/-/merge_requests/456 (under review), https://invent.kde.org/plasma/kscreenlocker/-/merge_requests/42 (merged)

60 unmodified KDE libraries	620 K	C++	Completely unmodified C++, only some minor cross-compilation build system fixes.
KWin	117 K	C++	38 LoC changed (0.021%) to avoid two out-of-bounds reads. 711 LoC changed (0.403%) to make OpenGL and Wayland support optional.
Plasma-framework Plasma-workspace Plasma-desktop	28K 112K 45K	C++	No changes related to CHERI, 14/383/39 LoC changed in the respective repositories (0.049/0.343/0.087%) to make OpenGL, DBus and Wayland optional.
Dolphin	42K	C++	No CHERI changes, 13 LoC changed (0.031%) to make DBus optional.
Gwenview	45K	C++	No CHERI changes, 23 LoC changed (0.051%) to make DBus and OpenGL optional.
poppler	193K	85% C++ 15% C	No CHERI changes, 2 LoC changed (0.001%) to silence a warning. Minor CMake build system fix for cross-compilation.
Okular	87K	C++	No CHERI changes, 4 LoC changed (0.005%) to make dependencies optional and fix warnings.
Systemsettings	4K	C++	No CHERI changes, 1 LoC (0.025%) changed to make DBus optional.
QtBase 5.15	1,661 K + 570K tests + 77K examples	C++	817 LoC changed in the library (0.049%) and 42 LoC changed in the tests (0.022%). No changes to the examples.
QtSvg 5.15	15 K	C++	12 LoC added (0.078%) to avoid an out-of-bounds read with empty strings.
QtDeclarative 5.15	496 K	C++	391 LoC changed (0.079%)
QtGraphicalEffects 5.15	3K	C++	No CHERI changes, 8 LoC (0.275%) changed to make OpenGL optional.
Additional libraries and programs[39]	560 K	68% C 30% C++ 2% ASM	Unmodified other than minor cross-compilation changes and non-CHERI warning fixes.

Complete changes summary

In total we compiled more than 6 million source lines of code for CHERI C/C++ successfully with only a very few changes: 2071 LoC or 0.034%. Of these 2071 changed lines, many were unrelated to CHERI; if we exclude changes that are required to make certain dependencies optional, we have a total of 1584 changed lines or 0.026% of the total.

[39] epoll-shim, exiv2, lcms2, libevdev, libexpat, libinput, libintl-lite, libudev-devd, mtdev, openjpeg, pixman, QtTools, QtQuickControls, QtQuickControls2, QtX11Extras, shared-mime-info

Compared to previous analyses of porting software to CHERI [6, 17], we find that this rate of 0.026% is noticeably lower than previously reported values of 0.1% (or 1.4% for the FreeBSD kernel[40]). This lower rate can be attributed to multiple factors: first, improvements to compiler analysis and CHERI C/C++ semantics[41] over the past few years have removed the need for many changes that were previously required. Second, our previous analysis showed that higher-level code generally requires fewer adaptations for CHERI C/C++, and the software we have ported in this work is mostly higher-level application code rather than low-level system libraries or language runtimes (with the exception of QtQml). Finally, a large proportion of the ported code is C++ rather than C, and writing code in C++ means certain patterns that can cause incompatibilities with CHERI are not used -- since the language or standard library provides abstractions for them (e.g., incorrect use of `realloc` rarely happens, since programmers can use classes such as `std::vector`). We also found that conversions between integers and pointers were less common in C++ compared to C, since C++ can use templates for generic data structures whereas C must resort to `void*` or `intptr_t`.

7. Compiler improvements

As part of our porting efforts we encountered multiple recurring CHERI-incompatible patterns that can at times be awkward to debug. These patterns are often undefined behaviour according to the C standard (creation of out-of-bounds pointers), but happen to "work" as expected on conventional architectures (at least, until a sufficiently sophisticated optimizing compiler attempts to leverage this undefined behaviour for optimization purposes).

`-Wshorten-cap-to-int`

This newly added compiler diagnostic warns whenever a capability is *implicitly* truncated to an integer. This diagnostic catches cases where a pointer that has been stored in a `uintptr_t` is implicitly converted to a smaller integer type before being converted back to `uintptr_t`. If this happens, the result of casting back to a pointer will be missing the capability metadata and cannot be dereferenced anymore. It is very similar to the existing clang warning `-Wshorten-64-to-32` that was added to find the equivalent issue when porting from 32-bit to 64-bit (truncating `intptr_t` via `int`). This warning was extremely useful while porting QtDeclarative since the code uses `uint64_t` and `uintptr_t` interchangeably for JavaScript object data. The QtQml port would have taken significantly longer without this warning, since we would have encountered the problems only at run time, rather than through compile-time diagnostics.

[40] The change rate of 1.4% is higher than normal in this case since we also support compiling the kernel as a hybrid program where every capability needs an explicit `__capability` annotation.
[41] For example, we changed the CHERI C semantics for casting between capabilities and integers to always use the address instead of the capability offset, and we modified CHERI Clang to support inferring the provenance source for arithmetic expressions involving multiple capabilities.

CHERI UndefinedBehaviorSanitizer

We also extended the UndefinedBehaviorSanitizer (UBSan) [20] compiler instrumentation to leverage CHERI architectural features (bounds and tagged memory) in order to detect creation of significantly out-of-bounds pointers. Creating out-of-bounds pointers is undefined behaviour in C,[42] and therefore the compiler may assume that this does not happen and optimize accordingly. However, in our porting efforts we discovered that multiple C libraries (e.g., libX11, fontconfig or the X Server) created such pointers due to incorrectly relocating pointers after calling `realloc()`. As conventional CPU architectures do not keep track of bounds at run time, this incorrect code appears to work but triggers traps when run as CHERI C/C++. The creation of the invalid out-of-bounds pointer and the actual use (i.e., dereference) usually occur with a rather large time gap, so a debugger backtrace rarely allows programmers to infer where the incorrect code is. After spending significant time debugging two such issues, we decided to add compiler instrumentation instead.

The current implementation of the CHERI UBSan[43] relies on the tag-clearing nature of out-of-bounds pointer arithmetic: a CHERI capability that is significantly out-of-bounds will become untagged [15] and thereby non-dereferenceable. This allows us to detect significantly-out-of-bounds pointers (guaranteed further than one past the end) by comparing the capability tag before pointer arithmetic with the resulting one. If the value changed, we can issue a diagnostic message or terminate the program to allow debugging with GDB. This new instrumentation can be enabled using a new `-fsanitize=cheri-unrepresentable` command line flag, and it remains possible to mix libraries compiled with and without instrumentation.

In the future, we plan to extend the CHERI UBSan instrumentation to not only detect capabilities that have become unrepresentable, but also look at the capability bounds to diagnose the creation of capabilities that are more than one element out-of-bounds. This is a rather simple change to the instrumentation, but it does add additional run-time overhead compared to only checking the capability tag.[44] As this more thorough instrumentation was not required to debug the libraries and programs used by our chosen prototype desktop stack, we have not yet implemented it.

8. Compartmentalization whiteboarding

Implementing software compartmentalization can be a substantial software engineering activity, and compartmentalization frameworks for CHERI and Morello remain ongoing research. As such, we did not attempt to develop end-to-end demonstrations of compartmentalization, but instead explored potential applications of compartmentalization in the Qt/KDE desktop environment through whiteboard exercises.

To the extent possible, we have applied knowledge gained in our early software compartmentalization work including the Capsicum OS sandboxing framework [21] and also in developing software compartmentalization techniques for CHERI [10]. We have taken into

[42] With the exception of one-past-the-end pointers which are in fact legal.
[43] https://github.com/CTSRD-CHERI/llvm-project/pull/553
[44] Currently, we would need to compare the computed address to both the lower and upper bounds of the capability. This overhead could be reduced by adding an CInBounds instruction to the ISA.

account the potential structural and performance improvements introduced by CHERI, such as single-address-space operation for multiple processes, and IPC roughly 1.5 orders-of-magnitude faster than MMU-enabled UNIX IPC, as predicted by on-FPGA CHERI research.

While CHERI supports a variety of compartmentalization models, we have chosen to whiteboard (and evaluate) with respect to a performance-enhanced *colocated process* (*co-process*) model, in which UNIX processes are able to perform substantially accelerated context switching and message passing. The practical import for user-level application software is that compartments are represented as processes, except that it is then possible to assume much faster communications between them making some previously untenable compartmentalizations viable. For example, with co-processes, we are able to assume that processing all image files within sandboxes is performance-viable, whereas we would not normally do that for MMU-based hardware designs. Isolation of software components is therefore done by a combination of CHERI protection within user-level and the kernel process abstraction when in system calls or traps.

We are therefore able to assume that FreeBSD's current Capsicum security model is applicable in sketching compartmentalized software policies. Capsicum is a software capability-based model in which sandboxed processes are denied access to global namespaces, such as the filesystem or network services, unless specifically granted them via UNIX file descriptors (in effect, making file descriptors into another kind of capability). We can therefore assume, for example, that a sensible compartmentalization of a software component would prevent undesired network or filesystem access beyond that specifically configured for the use case (privilege minimization).

Key focuses in our current work were identifying natural 'fracture lines' within KDE software, with focuses on:

- **Security benefit**: Compartmentalization should encapsulate and contain software elements with known high risk (e.g., image processing), exposure to untrustworthy input (e.g., network connectivity), and/or high-value information (e.g., passwords or private keys).
- **Natural encapsulation boundaries**: Sandboxed components should align with existing public or internal KDE or Qt APIs, providing for clean interfaces and benefiting from relatively careful API design with respect to internal data.
- **Performance plausibility**: Boundaries are selected that will offer affordable overhead either using classical MMU-based IPC or CHERI-enabled co-process IPC.

It is important to recognize that introducing software compartmentalization can be disruptive to the software, if it has not been designed with compartmentalization in mind from inception. This is an area deserving of substantial further research, but existing work [22–24] suggests that there is reason to hope that automated tooling could reduce the level of engineering and improve longer-term maintainability. For the purposes of this work, we assume that introducing compartment boundaries along existing encapsulation boundaries and APIs is feasible and will be accepted by the affected software communities.

Compartmentalization example 1: QImageReader

We analyzed the list of prior vulnerabilities (see 8. Security evaluation), and clearly saw that many of them are related to file format parsing (especially for image formats). We therefore believe that isolating the Qt image-parsing code would be one of the most impactful applications of compartmentalization in the current desktop stack. When loading/saving an image using the QImage/QPixmap/QIcon classes in Qt, the conversion between raw pixels and an image format is handled by QImageWriter and QImageReader. These classes provide a generic interface that dispatches to file format parsers inside Qt or a large number of third-party libraries such as libpng. These classes also include a plugin-based mechanism that allows registering handlers for other image formats. In the current version of QtBase, reading and writing image data is performed in-process without any form of sandboxing. Additionally, these classes expose a small API surface: the only communication is passing raw/compressed image data between the application and the compartment. This makes these classes much easier to compartmentalize than an API that uses (e.g.) function pointer callbacks and/or complex data structures. Finally, the main API is a single read()/write() function; therefore we should be able to avoid regular context switches between the main application and the compartmentalized image reader/writer.

We believe that compartmentalizing the Qt image format handlers would have a significant security impact, since it would mitigate vulnerabilities in any application that uses Qt to render and/or save images. Compartmentalizing the Qt image format handlers would not only prevent exploitation of applications that directly load images, but would have also mitigated configuration errors such as CVE-2019-7443, where a privileged daemon (running as root) that is not normally expected to handle images could be tricked into decoding arbitrary user-provided data as an image. This flaw can grant an attacker full control over the system using a vulnerability in any of the supported image format libraries. Moreover, compartmentalizing the image parsing code would allow returning an invalid QIcon or an "image not found" icon if the compartment crashes. This would convert all denial-of-service flaws (e.g., NULL-pointer dereferences) in image-rendering libraries such as libpng from fatal application crashes into recoverable errors.

Compartmentalization example 2: Okular's document renderer

Looking at the application stack ported as part of this work, another clear candidate for compartmentalization is the KDE document reader, Okular. It will often be used to read PDF files downloaded from potentially untrustworthy sources; the underlying PDF rendering library, poppler, has seen numerous exploitable Common Vulnerabilities and Exposures (CVEs) over the past years.[45]

Okular uses a plugin architecture to support many different rendering backends in addition to PDF files, so the code already uses a higher-level API that does not involve any function pointers. For document formats, Okular uses a Generator class that loads the appropriate plugin for the current file format and has APIs e.g. to return the raw image data for a given area of a page as well as the textual contents. In a previous research project at the University of Cambridge in 2015 [25], it was shown that Okular's renderer design can be

[45] https://www.cvedetails.com/product/24992/Freedesktop-Poppler.html?vendor_id=7971

adapted with relative ease to use process-based sandboxing techniques (in this case Capsicum [21] was used). In this work it was also shown that compartmentalization resulted in low performance overheads (less than 30ms to transfer the image data from the helper process) despite being an unoptimized proof-of-concept implementation. We believe that a compartmentalization approach based on CHERI would further reduce this overhead as it would allow running code within the same address space, and thereby reduce copying overheads.

Unlike the Qt image reader example, this compartmentalization will mitigate flaws only for a single application, but it is to be noted that it is an application with a rather large attack surface. One advantage of compartmentalizing applications over libraries is that it can be easier to adapt them as the internal APIs can be changed to be more friendly to compartmentalization.

Compartmentalization example 3: KFileMetadata

KFileMetadata is a KDE framework that provides a plugin-based system to read and write descriptive metadata (e.g., image dimensions, document authors, licenses, etc.) embedded in various file formats. It is commonly used for file indexing (e.g., the Baloo framework used by KDE) or to update metadata when writing files. Reading and writing metadata is performed in-process and usually performed by other lower-level libraries (e.g., libexiv2 for image formats, ffmpeg for videos, or poppler for PDFs). Many of these libraries have an extensive history of exploitable CVEs,[46] so we believe compartmentalization would provide a significant reduction in attack surface.

The API provided by KFileMetadata is very high-level, with all of the data processing in the internal implementation, and therefore should be easily adaptable to a compartmentalized software architecture. To read metadata from a file, users of KFileMetadata must find the appropriate extractor plugin for a given file format by calling `ExtractorCollection::fetchExtractors()`. This returns a subclass of `ExtractorPlugin` that implements two functions: `extract()` to return the metadata and `mimeTypes()` which returns a list of supported MIME types. This architecture is quite similar to the one used in Okular, so we believe that the same approach of wrapping the plugin's APIs with a compartmentalized proxy is feasible.

In terms of vulnerability mitigation, this will most likely have a lower impact than sandboxing the Qt image rendering code. However, on KDE Plasma desktops with file indexing enabled, untrusted files downloaded from the Internet will be scanned for metadata using Baloo (and thus KFileMetadata), which is a very high-risk library -- and should therefore be a prime candidate for compartmentalization.

[46] https://www.cvedetails.com/vendor/7561/Exiv2.html, https://www.cvedetails.com/vendor/3611/Ffmpeg.html, https://www.cvedetails.com/product/24992/Freedesktop-Poppler.html

9. Security evaluation

In this section, we review security vulnerabilities from X11, Qt, KDE, and a selection of supporting libraries over a five-year period (August 2016 - July 2021)[47] to assess whether our proposed or actual CHERI adaptations would have impacted the severity of the vulnerabilities. This is done as a whiteboarding exercise, due to time and scope constraints, but reflects reasonable best estimates of the impacts of CHERI memory protection (including temporal heap memory safety) and compartmentalization (as described). A more rigorous study would perform more in-depth studies of the specific code paths, ideally with an adversarial element to evaluate practical exploitability.

Information sources

For each software category, we review the complete set of Common Vulnerabilities and Exposures (CVEs) or other announced past vulnerabilities. Where possible, we rely on vulnerability lists documented on the web pages of the corresponding open-source project websites (e.g., X.org and KDE). However, in some cases, where a project doesn't maintain such a list, we turn to externally maintained lists (e.g., CVE Details[48]). For software components in the former category, researching vulnerabilities was typically straightforward, as security advisories provide detailed technical information, references to software changes and patches, and important contextual information. Those in the latter category required substantially more work to research vulnerabilities across multiple independent websites, issue trackers, source repositories, and so on.

Open-source projects inevitably handle discovered reported vulnerabilities differently -- for example, whether all potential vulnerabilities are announced by the project, whether and when denial of service is considered a vulnerability, and how vulnerabilities are assigned severities. We do not address these potential concerns, given the scope of this project, and instead simply review each CVE to offer our analysis of how it might have been impacted by CHERI deployment.

We drew on a blend of information sources, including individual vendor websites as well as their issue trackers and source-code repositories, the CVE Details web site, and the National Vulnerability Database (NVD).[49] In some cases we also turned to analyses presented in OS vendor advisories and issue trackers such as those from Ubuntu[50] and RedHat[51]; this was especially helpful when analyzing supporting library vulnerabilities, which were rarely documented by the vendors themselves. In our analysis, we indicate the primary source(s) of vulnerability information used for each project.

[47] As noted in the Qt evaluation section, Qt did not have announced vulnerabilities in this period, so we reviewed vulnerabilities back through 2011.
[48] https://www.cvedetails.com
[49] https://nvd.nist.gov
[50] https://ubuntu.com/security/notices
[51] https://access.redhat.com/security

Advisory and vulnerability descriptions

Open-source projects with vulnerability disclosure processes request CVEs for specific software vulnerabilities. However, their security advisories may address more than one vulnerability -- for example, when a set of related vulnerabilities is reported as a result of deploying a new static analysis tool, or when auditing for further cases of a newly reported vulnerability class. In our analysis, we consider vulnerabilities at the granularity provided by the open-source project: one entry per advisory (and potentially multiple vulnerabilities) if reported in that way by the project, and otherwise one entry per vulnerability if advisories are not issued by the project.

For each table entry, we report the following:

- **CVE(s)**: The unique vulnerability identifier(s) reported by the software vendor.
- **Date**: The date the vendor released an advisory or patch for the vulnerability.
- **Severity**: The indication of vulnerability severity, by the vendor, or our own, if not.
- **Description**: A very brief description of the vulnerability or vulnerabilities.
- **Assessment**: Our brief assessment of the potential impact of CHERI memory protection and compartmentalization on the vulnerabilities. If we have reduced confidence in our analysis for a particular vulnerability, we also note that here.

Severities

Where open-source projects issue advisories with severities, we report those severities. If the projects do not issue advisories, or issue advisories but do not assign severities to vulnerabilities, we assign severities as follows:

- **Critical vulnerabilities** are those likely yielding or contributing directly to arbitrary code execution across a trust boundary (e.g., as the local user following processing of an untrustworthy data file, or as the root user when interacting with an unprivileged desktop user).
- **Moderate vulnerabilities** are those leading to denial of service, or that may unnecessarily expose more vulnerable attack surfaces without necessarily constituting a vulnerability. For example, an information disclosure might provide useful information required to build a successful exploit chain, while not itself enabling direct privilege escalation.

Threat model

Most of the open-source GUI and desktop code we reviewed did not document a well-defined threat model. However, as many of the projects have structured vulnerability disclosure and review processes, and assign criticalities to disclosed vulnerabilities, we were able to reason about their de facto threat models. In general, for the purposes of vulnerability analysis and disclosure, the projects were concerned with **privilege escalation**, **private data disclosure**, and **denial of service**.

Privilege escalation due to arbitrary code execution or file modification

This appeared to be considered the most important concern spanning almost all vulnerability disclosures we reviewed, and tended to fall into one of two categories:

- **Remote to user privilege**. With these vulnerabilities, the desktop user is interacting with network services (e.g., viewing websites, reading instant messages, downloading and handling files, etc.) that are able to trigger local code execution as the desktop user. In most cases this is due to memory-safety vulnerabilities, but in some it may derive from bugs in (for example) archiving programs that permit arbitrary local file replacement as the user. This is trivially escalated to arbitrary code execution.
- **User privilege to system privilege**. With these vulnerabilities, the desktop user interacts with privileged system services (e.g., the window server, screen locker, etc.) and is able to trigger local code execution as the privileged user. As above, in most cases this is due to memory-safety vulnerabilities, but likewise arbitrary file overwrite is trivially escalated to arbitrary code execution.

The specifics of arbitrary code execution vulnerabilities varied substantially. X11, for example, has experienced numerous C-language buffer overflows. KDE also suffered from arbitrary code execution vulnerabilities, but these were more typically with respect to logical errors, such as by permitting file overwrites due to incorrect enforcement of target directory constraints in archiving tools.

Private data disclosure

The Private Data Disclosure category relates to library or application logical bugs in which private data is improperly disclosed. For example, KDE's KMail client contained a logical bug causing email intended to be submitted encrypted to instead be sent in plain text. This class appears to be largely secondary compared to arbitrary code execution, featuring in few vulnerability advisories. Nevertheless, private data disclosure is a key attacker end objective often achieved via arbitrary code execution.

Undesired data modification

Undesired data modification vulnerabilities are most frequently considered to be a means to the end of arbitrary code execution; for example, focus is placed on buffer overflows onto stack or heap metadata. The broader category of concerns here -- e.g., overflow into application data that does not lead to user data corruption -- seems largely unconsidered.

Denial of service

Although often assigned a lower severity than vulnerabilities leading to arbitrary code execution, denial-of-service vulnerabilities still featured prominently. This was particularly true for image-processing libraries, where the implications of a crash could be significant for the larger application;they also featured prominently in the KDE vulnerability corpus. CHERI memory protection coerces potential arbitrary code execution vulnerabilities into deterministic crashes, which may reduce a critical vulnerability to one of low or moderate severity -- but does not completely eliminate it.

Mitigation

We consider a vulnerability mitigated if a bug would no longer be considered a vulnerability under the vendor's threat model. However, as vendors rarely publish threat models, and we must work with de facto ones, this presents some challenge to analysis.

Memory safety

CHERI memory protection in CHERI C and C++ directly mitigates both vulnerabilities (e.g., buffer overruns and use-after-reallocation) and exploit techniques (e.g., control-flow pointer injection, integer-pointer confusion, and violations of spatial safety in the implementation), coercing attempted attacks from arbitrary code execution into software crashes. Cornucopia and related techniques can reliably and deterministically implement heap safety when using CHERI, and our analysis assumes heap temporal safety is present. These techniques collectively can mitigate a substantial proportion of C and C++ vulnerabilities, in which simple programming errors yield a high-severity vulnerability, or at least essential steps in a larger exploit chain (such as pointer value leakage, or arbitrary write primitives).

We choose to consider a memory-safety vulnerability mitigated only if CHERI C/C++ directly addresses the vulnerability itself (e.g., spatial safety preventing a buffer overflow from running into another allocation or global variable, or heap temporal safety preventing use-after-reallocation). While CHERI may indeed limit exploit techniques -- for example, by deterministically preventing pointer reinjection and hence making malicious control-flow manipulation more difficult -- it is not currently clear how to best reason about the strength of that protection.

The potential impact of denial of service arising from memory safety triggering a crash is extremely application- and use-case dependent. For example, converting a buffer overflow into software termination prevents arbitrary code execution, but might still interrupt service delivery to the user if it crashes their web browser or window server. On the other hand, a command-line tool crashing when processing an untrustworthy image may have little impact on user experience.

Compartmentalization

CHERI compartmentalization will sometimes be able to mitigate denial of service by limiting portions of applications affected by software termination, including when crashes originate with memory-safety violations. For example, if image processing is compartmentalized, a failed image processing sandbox may lead to a 'bad image file' icon in a web or file browser, rather than crash of the tab or full application. Reviewing desktop relevant vulnerabilities, it was clear that many vulnerabilities arise from the processing of images and other media files, and compartmentalizing that processing would have a significant impact on reliability in the presence of attempted attacks.

The extent to which a vulnerability is 'fully mitigated' can therefore be hard to determine. We take the view that a substantial reduction in effect of a vulnerability (e.g. arbitrary code execution to software crash) constitutes a significant mitigation -- but will always note where a crash might have a broader impact that itself would need mitigating, and whether software compartmentalization would be likely to contribute to further mitigation.

When software compartmentalization detects a memory-protection failure (or other fail-stop condition), that impact on software can also vary substantially. In some cases, it may be possible to entirely mask the failure by restarting a compartment and proceeding to the next item of data, such as a packet or message. In other cases, there are clear and user-relevant failure modes that can be exposed without interrupting operation. For example, if a memory-protection exception is caught when rendering an image, a “broken image” icon could be displayed, as is the case today when attempting to render a corrupted image file that does not contain an exploit.

However, there may be some cases where compartmentalization is unable to usefully mask a failure essential to the operation of affected software. For example, a memory-protection error in the X Windows server could lead to termination of the entire desktop session. As we review vulnerabilities for potential compartmentalization opportunities, it can sometimes be difficult to understand the scope for mitigation without full application context, especially for libraries in isolation from a specific application that uses them.

Depth of analysis

Due to the limited timeline and scope of this project, we have generally relied heavily on the vulnerability analyses provided by the software vendors (e.g., in the vendor’s own revision-control history or vulnerability announcement), or by a downstream software distribution (e.g., Ubuntu or Redhat analysis). In some cases, we performed direct source-code analysis where an advisory was unclear or the implications were not fully elaborated. We have marked certain vulnerabilities as containing insufficient information, or as low confidence analyses, where the nature of the vulnerability was unspecified or unclear, where the threat model or vulnerability argument was unclear, or where our confidence in mitigation is lower. Given further time, it would be desirable to take our analysis further through closer code inspection or experimentation.

X.org

The X.org X Window System is a display server along with a collection of device drivers, client libraries, and command-line tools supporting desktop graphics for UNIX (and other) systems. X.org releases regular security advisories and software updates for reported vulnerabilities, from which we have gathered this data, but does not publish an explicit threat model or assign severities to vulnerabilities.[52]

The threat model around X server vulnerabilities has changed over time. Historically, the X server has been run as root due to using user-level device drivers for display hardware. A key threat, then, is X client applications exploiting vulnerabilities over their connections to the server, potentially granting additional privilege to an otherwise less privileged client program. Running the X server as root has become less common, but criticality assessments for remote code execution continue to take this potential configuration into account. Our presentation of X server vulnerability data also adopts this perspective.

[52] https://www.x.org/wiki/Development/Security/

The threat model around X client library vulnerabilities often relates to the potential for arbitrary code execution in client programs running with elevated privilege, and connected to a potentially malicious X server (or program behaving as an X server). Historically, the concern involved setuid root clients linked against X client libraries, which the attacker would run connected to a malicious (but unprivileged) X server instance that would be able to gain root privilege by sending specially crafted messages, leading to high criticality for such vulnerabilities. A significant effort has been invested in deprivileging X client programs for this reason, but a number still remain. Our presentation of client-library vulnerability data again adopts the X.org perspective.

In most cases, we assess that while CHERI memory protection will mitigate a vulnerability, it will lead to an X server or X application crash that is not easily mitigated by software compartmentalization. However, we consider a potential arbitrary code-execution vulnerability to be mitigated if it will then deterministically crash rather than allow code execution. We have taken the view, for the purposes of this work, that crashes in the X server or X client libraries do not lend themselves to mitigation by software compartmentalization due to the essential role X11 plays in applications. It could be that further analyses show this assumption to be incorrect.

Vulnerability	Date	Severity	Description	Assessment
CVE-2021-3472	13 April 2021	Critical	An integer overflow allowed out-of-bounds memory accesses in the X server, which could lead to arbitrary code execution.	Mitigated by memory safety (but will cause the X server to crash).
CVE-2020-14360, CVE-2020-25712	1 December 2020	Critical	Two independent failures of input validation in the XKB extension allow out-of-bounds memory accesses in the X server, which could lead to arbitrary code execution.	Mitigated by memory safety (but will cause the X server to crash).
CVE-2020-14345, CVE-2020-14346, CVE-2020-14361, CVE-2020-14362	25 August 2020	Critical	Insufficient input validation in multiple X server extensions allow out-of-bounds memory accesses, which could lead to arbitrary code execution.	Mitigated by memory safety (but will cause the X server to crash).
CVE-2020-14363	25 August 2020	Critical	Integer overflow and double free in libx11 locale handling could lead to arbitrary code execution in X client applications.	Mitigated by memory safety (but will cause an application crash).
CVE-2020-14344	31 July 2020	Critical	Integer overflows and signed/unsigned comparisons in libX11 input methods could lead to arbitrary code execution in X client applications.	Mitigated by memory safety (but will cause an application crash).
CVE not assigned	25 October	Critical	Incorrect command-line validation in the X server can	Unmitigated (software design error).

	2018		lead to arbitrary code execution, or arbitrary file overwrite.	
CVE not assigned	22 August 2018	Moderate	libXcursor could write one byte out of bounds when processing Xcursor theme files held in a malloc'd buffer on the heap, which could be a step in a more complex attack on an X client application.	Mitigated by memory safety (but will cause an application crash).
CVE-2018-14599, CVE-2018-14600, CVE-2018-14598	21 August 2018	Critical	Multiple libX11 library memory-safety bugs can lead to arbitrary code execution in X client applications.	Buffer overruns are mitigated by memory safety (but will cause an application crash). Most likely compartmentalization would not mitigate the effects of those crashes.
CVE-2017-12176, CVE-2017-12177, CVE-2017-12178, CVE-2017-12179, CVE-2017-12180, CVE-2017-12181, CVE-2017-12182, CVE-2017-12183, CVE-2017-12184, CVE-2017-12185, CVE-2017-12186, CVE-2017-12187	12 October 2017	Critical	Multiple buffer overruns in the X server's protocol processing can lead to arbitrary code execution.	Mitigated by memory safety (but will cause an X server crash).
CVE-2017-13721, CVE-2017-13723	4 October 2017	Critical	Buffer overruns in the X server's handling of SHM and XKB client requests can lead to arbitrary code execution.	Mitigated by memory safety (but will cause an X server crash).
CVE-2016-5407. CVE-2016-7942, CVE-2016-7943, CVE-2016-7944, CVE-2016-7945, CVE-2016-7946, CVE-2016-7947, CVE-2016-7948, CVE-2016-7949, CVE-2016-7950, CVE-2016-7951, CVE-2016-7952, CVE-2016-5953	4 October 2016	Critical	Multiple buffer overruns in the X client libraries can lead to arbitrary code execution in X client applications.	Mitigated by memory safety (but will cause an application crash).

We reviewed 11 X.org security advisories to understand the potential applicability of CHERI vulnerability mitigation, and found that:

- 10 (91%) were likely mitigated by memory safety.

Overall, CHERI mitigation would likely have achieved roughly a **91% mitigation rate** for these collections of security issues. Note that many X.org security advisories covered multiple underlying vulnerabilities, and so the vulnerability mitigation rate is likely substantially higher.

Qt

Qt is an open-source GUI toolkit and framework, as well as design tools, providing a variety of services such as GUI widgets, I/O handling including audio and video file formats and rendering, networking. Qt is used as the baseline set of class libraries for KDE. Qt releases regular vulnerability advisories, from which we have gathered these data.[53] Due to a lack of recent published Qt security advisories, we extended our investigation back through to 2011.

Vulnerability	Date	Severity	Description	Assessment
CVE-2020-0570	14 September 2020	High (7.3)	Library (plugin) search including current working directory may allow elevation of privilege via local access.	Unmitigated (software design error). Possibly mitigated by compartmentalization.
CVE-2017-10904CVE-2017-10905 JVN#67389262 JVN#27342829	22 November 2017	High	Logic error allows malicious users to enable a custom debugger binary which can result in arbitrary code execution on Android devices	Unmitigated (software design error). Possibly mitigated by compartmentalization.
CVE-2015-1858, CVE-2015-1859, CVE-2015-1860	12 April 2015	High	Incorrect parsing of BMP, ICO and GIF files results in denial of service crashes and/or buffer overflows.	Mitigated (buffer overflows), denial-of-service mitigation possible with well-designed compartmentalization.
CVE-2015-0295	22 February 2015	Low	Library logic error results in division by zero (denial-of-service) when decoding invalid BMP images	Unmitigated, partial mitigation possible with well-designed compartmentalization.
CVE-2015-1290	9 January 2015	High	Memory corruption bug in V8 JavaScript engine (included in QtWebEngine) allows for arbitrary code execution.	Mitigated by memory protection (downgraded to DoS).
CVE-2014-0190	24 April 2014	Low	Library logic error results in NULL-pointer dereference (denial-of-service) while decoding	Unmitigated, partial mitigation possible with well-designed compartmentalization.

[53] https://www.qt.io/blog/tag/security and the announcements mailing list https://lists.qt-project.org/pipermail/announce/

			invalid GIF images	
CVE-2013-4549	5 December 2013	Low	Library logic error in XML parsing could result in infinite memory usage and thereby denial-of-service	Unmitigated, partial mitigation possible with well-designed compartmentalization.
CVE-2013-0254	4 February 2013	Low to Medium	Library logic error created POSIX shared memory segments world-writable	Unmitigated (software design error).
CVE-2012-6093	2 January 2013	Low	QSslSocket will load error code from wrong memory location when run with a different OpenSSL version than the one Qt was compiled against	Likely unmitigated (or converted to a crash depending on structure sizes). Irrelevant on a standard Linux desktop deployment.
CVE-2012-5624	17 November 2012	Low	XMLHttpRequest allows redirection from HTTP to file:// scheme which can expose local file contents to QML applications	Unmitigated (software design error).
CVE-2011-3194	21 September 2011	High	Buffer overflow in the TIFF image reader allows for arbitrary code execution	Mitigated by memory protection (downgraded to DoS). DoS mitigated by compartmentalization.
CVE-2011-3193	22 August 2011	High	Buffer overflow in the HarfBuzz text rendering engine allows for arbitrary code execution	Mitigated by memory protection (downgraded to DoS). DoS possibly mitigated by compartmentalization.

We reviewed 11 Qt security advisories to understand the potential applicability of CHERI vulnerability mitigation, and found that:

- 6 (55%) were likely mitigated by straightforward software compartmentalization.
- 4 (36%) (overlapping with some of the above) might or would have been mitigated by memory protection.

Collectively, memory protection and compartmentalization would likely have achieved roughly a **82% mitigation rate** for these collections of security issues.

KDE

KDE is an open-source desktop environment including window manager, web browser, file manager, contact manager, mail reader, and other applications such as an office suite and graphics package. KDE releases regular vulnerability advisories, from which we have gathered these data.[54]

Vulnerability	Date	Severity	Description	Assessment
CVE-2021-31855	29 April 2021	Low	Application logical error in KDE e-mail reader incorrectly uploads decrypted and then deleted attachment to mail server.	Unmitigated (software design error).
CVE-2021-28117	10 March 2021	Low	Application logical error in the KDE package manager fails to limit rendered links to http/https.	Unmitigated (software design error).
CVE-2020-27187	17 October 2020	Important	Application logical error in KDE Partition Manager could lead to local privilege escalation.	Unmitigated (software design error).
CVE-2020-26164	2 October 2020	Important	Application logical errors in KDE Connect may lead to local denial-of-service.	Unmitigated, possibly except for one use-after-free vulnerability that could have been further exploitable for privilege escalation, and would be mitigated by memory protection.
CVE-2020-24654	27 August 2020	Important	Application logical error in KDE archiving tool may install files outside of target directory.	Mitigated by straightforward compartmentalization.
CVE-2020-24654	30 July 2020	Important	Application logical error in KDE archiving tool may install files outside of target directory.	Mitigated by straightforward compartmentalization.
CVE-2020-12755	10 May 2020	Low	Application logical error in KDE password wallet improperly saves password when not asked to.	Unmitigated (software design error).
CVE-2020-9359	12 March 2019	Low	Application logical error allows arbitrary binary execution by KDE PDF viewer Okular.	Mitigated by straightforward compartmentalization.
CVE-2019-14744	7 August 2019	High	Application design error executes arbitrary command lines in .desktop and .directory files	Unmitigated (software design error).
CVE-2019-7443	9 February 2019	Medium	Unprivileged users can trigger parsing of arbitrary	Mitigated by both memory protection

[54] https://kde.org/info/security/

			data, such as images, in privileged daemons via kauth framework, which can lead to arbitrary code execution as root.	and straightforward compartmentalization (memory protection coerces arbitrary code execution into a crash).
CVE-2018-19516	28 November 2018	Low	KMail can be tricked into opening a remote web page even with HTML parsing disabled.	Unmitigated (software design error).
CVE-2018-19120	12 November 2018	Low	HTML thumbnail previewer improperly accessed remote files.	Mitigated by straightforward compartmentalization.
CVE-2018-10380	4 May 2018	High	Filesystem API race allows unprivileged user to own any file in the system.	Unmitigated (software design error).
CVE-2018-6791	8 February 2018	High	Shell syntax embedded in VFAT volume labels allow arbitrary command execution.	Unmitigated (software design error).
CVE-2018-6790	8 February 2018	Low	Desktop notifications rendered as HTML improperly access remote files.	Mitigated by straightforward software compartmentalization.
CVE-2017-15923	12 November 2017	High	An invalid message can crash Konversation IRC client.	Unmitigated (software design error).
CVE-2017-9604	15 June 2017	Medium	Delayed message send in KMail disabled OpenPGP signing and encryption.	Unmitigated (software design error).
CVE-2017-8849	10 May 2017	High	CIFS filesystem browser permits running arbitrary binaries as root.	Unmitigated (software design error).
CVE-2017-8422	10 May 2017	High	Kauth framework fails to check remote process identity properly, allowing arbitrary binary execution as root.	Unmitigated (software design error).
CVE-2017-6410	28 February 2017	Medium	Proxy Auto-Configuration (PAC) files may trigger the leak of full https URL information to proxies, which can be triggered remotely.	Unmitigated (software design error).
(None assigned?)	27 February 2017	Medium	A bug in KMail handling of Outlook file attachments allows attackers to write arbitrary files in the filesystem when the attachment is opened.	Mitigated by straightforward software compartmentalization.

CVE-2017-5593	14 February 2017	Important	The Kopete instant messaging client allows Jabber identity impersonation.	Unmitigated (software design error).
CVE-2017-5330	12 January 2017	Important	The Ark file archiving tool allowed maliciously constructed tar files to trigger execution of arbitrary binaries.	Mitigated by straightforward software compartmentalization.
CVE-2016-7966	6 October 2016	Important	A bug in the KMail text viewer allowed HTML parsing to be enabled (otherwise disabled by default), which might expose other exploitable vulnerabilities.	Some potential vulnerabilities might be mitigated by memory protection or straightforward software compartmentalization (memory protection coerces arbitrary code execution into a crash).
CVE-2016-7967	6 October 2016	Critical	KMail improperly executed received Javascript embedded in HTML messages in the local execution context, including allowing local file access.	Mitigated by straightforward software compartmentalization.
CVE-2016-7968	6 October 2016	Normal	KMail improperly executed Javascript embedded in HTML messages, which might expose other exploitable vulnerabilities.	Some potential vulnerabilities might be mitigated by memory protection or straightforward software compartmentalization (memory protection coerces arbitrary code execution into a crash).
CVE-2016-7787	30 September 2016	Important	A maliciously crafted command line intended to be run as root may be partially masked, causing the user to run commands they do not intend.	Unmitigated (software design error).
CVE-2016-6323	24 July 2016	Important	The KNewStuff framework allowed maliciously constructed tar and zip files to install files outside of the target extraction directory.	Mitigated by straightforward software compartmentalization.

We reviewed 28 KDE security advisories to understand the potential applicability of CHERI vulnerability mitigation, and found that:

- 12 (43%) were likely mitigated by straightforward software compartmentalization.
- 4 (14%) (fully overlapping with the above 12) might or would also have been mitigated by memory protection.

Collectively, memory protection and compartmentalization would likely have achieved roughly a **43% mitigation rate** for these vulnerabilities.

Other libraries

Beyond the windowing system and toolkit libraries (see below), the open-source desktop environment depends heavily on a set of libraries that handle common data formats. We included in our analysis a sample of these[55] including:

Software module(s)	Description	Vulnerability information source(s)
freetype2	Font rendering library (C)	Vendor website[56]
giflib*	GIF image rendering library (C)	Vendor issue tracker[57], CVE Details[58]
libjpeg-turbo*	JPEG image rendering library (C)	CVE Details[59]
libpng	PNG image rendering library (C)	Vendor website[60], CVE Details[61]
libxml2*	XML parsing library (C)	CVE Details[62]

Unlike the larger structured open-source projects (X11, Qt, KDE), these libraries typically do not come with established vulnerability disclosure processes, nor documented threat models. We therefore:

- Report at the granularity of assigned CVEs rather than announced vulnerability sets, which may be finer grained than for reporting for other software components, and may also suffer reduced accuracy in terms of our analysis, as the software vendor themselves may not have contributed to the explanation of its potential implications (e.g., denial of service vs. arbitrary code execution as an outcome, or with respect to pertinent threat models).

[55] Due to the limited time for this project and the large number of supporting libraries used in a contemporary desktop environment, we limited this analysis to five representative libraries. Looking at the vulnerability summaries from cvedetails.com, we believe that further interesting case studies would have been libtiff (at least 176 CVEs), ffmpeg (365 CVEs), and poppler (at least 68 CVEs).
[56] https://www.freetype.org/index.html#news
[57] https://sourceforge.net/p/giflib/bugs/
[58] https://www.cvedetails.com/product/33654/Giflib-Project-Giflib.html
[59] https://www.cvedetails.com/product/40849/Libjpeg-turbo-Libjpeg-turbo.html
[60] http://www.libpng.org/pub/png/libpng.html
[61] https://www.cvedetails.com/vendor/7294/Libpng.html
[62] https://www.cvedetails.com/product/3311/Xmlsoft-Libxml2.html

- Work with a *de facto* threat model assuming that data processed by the libraries will be from untrustworthy (and potentially malicious) sources attempting to achieve arbitrary code execution, private data disclosure, or denial of service.

These libraries are marked with a ‘*’ above.

Vulnerability	**Date**	**Severity**	**Description**	**Assessment**
CVE-2016-4658	25 September 2016	Critical	Libxml2 use-after-free on invalid input can lead to denial of service or arbitrary code execution.	Mitigated by memory protection, but will lead to application crash, which in turn may be mitigated by straightforward compartmentalization.
CVE-2016-3177	23 January 2017	Critical	Giflib double free on invalid input can lead to denial of service or arbitrary code execution.	Mitigated by memory protection, but will lead to application crash, which in turn may be mitigated by straightforward compartmentalization.
CVE-2016-10087	29 January 2017	Moderate	Libpng NULL pointer vulnerability on invalid input can lead to denial of service.	Mitigated by straightforward compartmentalization.
CVE-2017-15232	11 October 2017	Moderate	Libjpeg-turbo NULL-pointer dereference on invalid input can lead to denial of service.	Mitigated by straightforward compartmentalization.
CVE-2018-13785	5 April 2018	Moderate	Linpng integer overflow and divide-by-zero on invalid input can lead to denial of service.	Mitigated by straightforward compartmentalization.
CVE-2018-11489	26 May 2018	Critical	Giflib buffer overflow on invalid input can lead to denial of service or arbitrary code execution.	Mitigated by memory protection, but will lead to application crash, which in turn may be mitigated by straightforward compartmentalization.
CVE-2018-11490	26 May 2018	Moderate	Giflib buffer overflow on invalid input can lead to denial of service.	Mitigated by straightforward compartmentalization.
CVE-2018-1152	18 June 2018	Moderate	Libjpeg-turbo divide-by-zero exception on invalid input can lead to denial of service.	Mitigated by straightforward compartmentalization.
CVE-2018-14048	13 July 2018	Moderate	Libpng unspecified vulnerability can lead to denial of service.	Mitigated by straightforward compartmentalization. (This is a low confidence

				assessment.)
CVE-2018-19664	29 November 2018	Moderate	Libjpeg-turbo heap-based buffer over read on invalid input can lead to denial of service.	Mitigated by straightforward compartmentalization.
CVE-2018-20330	21 December 2018	Critical	Libjpeg-turbo integer overflow permits heap-based buffer overwrite that can lead to denial of service or arbitrary code execution.	Mitigated by memory protection, but will lead to application crash, which in turn may be mitigated by straightforward compartmentalization.
CVE-2019-7317	29 January 2019	Critical	Libpng use-after-free on invalid input can lead to denial of service or arbitrary code execution.	Mitigated by memory protection, but will lead to application crash, which in turn may be mitigated by straightforward compartmentalization.
CVE-2018-14498	7 March 2019	Moderate	Libjpeg-turbo heap-based buffer overread on invalid input can lead to denial of service.	Mitigated by straightforward compartmentalization.
CVE-2017-12652	10 July 2019	Low	Libpng could allocate undesirably large amounts of memory due to a missing resource limit check against a user-specified limit, leading to denial of service.	Mitigated by straightforward compartmentalization. (This is a low confidence assessment.)
CVE-2018-14550	10 July 2019	Critical	Libpng utility stack-based buffer overwrite that can lead to arbitrary code execution.	Mitigated by memory protection, but will lead to application crash, which in turn may be mitigated by straightforward compartmentalization.
CVE-2019-15133	17 August 2019	Moderate	Giflib divide-by-zero exception on invalid input can lead to denial of service.	Mitigated by straightforward compartmentalization.
CVE-2020-13790	9 June 2020	Moderate	Libjpeg-turbo heap-based buffer over read on invalid input can lead to denial of service.	Mitigated by straightforward compartmentalization.
CVE-2020-17541	15 June 2020	Critical	Libjpeg-turbo stack-based buffer overwrite on invalid input can lead to denial of service or arbitrary code execution.	Mitigated by memory protection, but will lead to application crash, which in turn may be mitigated by straightforward compartmentalization.

CVE-2020-15999	19 October 2020	Critical	FreeType heap-based buffer overwrite on invalid input can lead to denial of service or arbitrary code execution.	Mitigated by memory protection, but will lead to application crash, which in turn may be mitigated by straightforward compartmentalization.
CVE-2020-27818	8 December 2020	Moderate	Libpng command-line tool global variable over read on invalid input can lead to denial of service.	Mitigated by straightforward compartmentalization.
CVE-2021-20205	10 March 2021	Moderate	Libjpeg-turbo divide-by-zero exception on invalid input can lead to denial of service.	Mitigated by straightforward compartmentalization.
CVE-2020-23922	21 April 2021	Moderate	Giflib buffer overread on invalid input can lead to denial of service.	Mitigated by straightforward compartmentalization.

We reviewed 22 vulnerabilities in these supporting libraries to understand the potential applicability of CHERI vulnerability mitigation, and found that:

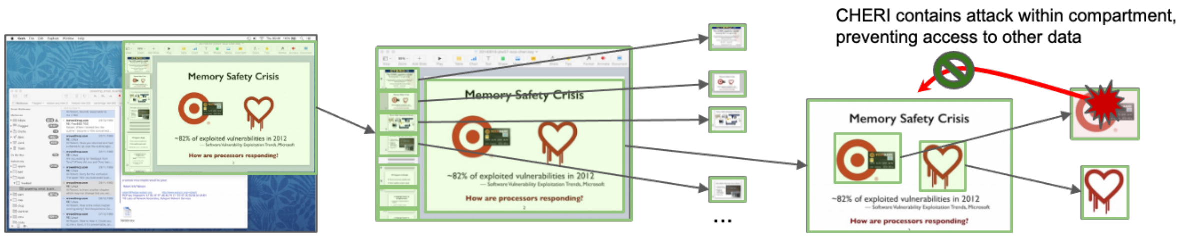
- 8 (36%) were likely mitigated by memory safety; this was 100% of critical vulnerabilities potentially leading to arbitrary code execution.
- 22 (100%) were likely mitigated by straightforward software compartmentalization, including the potential denial-of-service instances arising from mitigation using memory safety.

Collectively, memory protection and compartmentalization would likely have achieved roughly a **100% mitigation rate** for these vulnerabilities.

10. Desktop demonstration narrative

Our investigation suggests a strong argument can be made for CHERI memory protection and compartmentalization within a desktop environment. This is because there are substantial quantities of memory-related vulnerabilities mitigated by CHERI C/C++ memory safety, and also because there is significant useful software modularity that appears to lend itself to useful fine-grained software compartmentalization. In particular, components dealing with risky data from untrustworthy sources (e.g., media files, email and instant messages, network protocol processing, crypto, etc.) are often well modularised, so compartmentalization would likely be effective both in directly mitigating vulnerabilities, and also in limiting the potential for denial-of-service impact when memory-safety violations are detected. Compartmentalization in key class libraries would have a substantial impact across the full application corpus.

Take for example a presentation package (see figure below). Compartmentalization might be productive when implemented at many granularities: sandboxing the application as a whole, the processing of each slide, and the processing of each image or other media embedded in the slide. Memory safety within image processing compartments would eliminate arbitrary code execution arising from the vast majority of exploitable vulnerabilities, coercing them into crashes. Compartmentalization would prevent a crash in processing an image from crashing the full application, and also mitigate other types of non-memory-safety vulnerabilities, such as inserted backdoors or higher-level logical errors. CHERI allows that compartmentalization


CHERI contains attack within compartment, preventing access to other data
Memory Safety Crisis
~82% of exploited vulnerabilities in 2012
How are processors responding?
...

to be substantially more efficient than existing technologies, and therefore potentially be affordable at many more boundaries.

We believe that this approach applies to many other applications, including web browsers, mail readers, and messaging applications that process complex data from multiple origins during that execution -- all key desktop attack surfaces.

11. Constructive plan

Large-scale adaptation of an open-source desktop ecosystem will be a non-trivial undertaking due to the size and complexity of the code base, as well as the potentially considerable difficulty of thoroughly testing the resulting software stack. We partition our analysis and plan in two parts: memory safety and compartmentalization. Although not universally true, we generally consider memory safety as a baseline and prerequisite for effective compartmentalization. Because of the number of largely independent software components in a contemporary desktop stack, there is the opportunity for a highly collaborative effort approaching both aspects with considerable concurrency. We do not attempt to assign a level of effort to this overall program of work.

Memory safety

Our preliminary investigation suggests that, if a developer has even a modest background with CHERI C/C++, adapting components from such a stack for memory safety is relatively straightforward. Further, memory safety appears to offer a relatively easily achieved, yet substantial security win. Many of the changes required fixed actual software bugs (e.g., with respect to undefined C behaviour, buffer overflows, and so on).

The areas of greatest potential concern lie in:

- legacy code bases suffering substantial type-related confusion or misuse between pointers and integers;
- code bases that rely on specific properties of the architecture due to, for example, implementing a Just-in-Time (JIT) compiler;
- code bases that are particularly sensitive to the performance overheads associated with expanded pointer size, such as language runtimes (some of which already use pointer compression techniques to avoid the overheads of 64-bit pointers); or
- code bases making very flexible or extensive use of function pointers, such as language runtimes, where simple recompilation will not be sufficient to eliminate memory-safety vulnerabilities (e.g., WebKit [26]).

We feel that there is a particular gap in knowledge around language runtimes, where it is currently difficult to predict the potential scope (and cost) of changes that will need to be made.

A practical engineering challenge to adapting software packages to CHERI C/C++ is the difficulty of rigorous dynamic testing, given that many applications have only modest coverage in their test suites. This will place a heavy focus on manual testing, which is time consuming.

To properly assess the security impact of CHERI memory protection on this larger software corpus, we would also recommend an investment in security review and adversarial testing based on a richer retrospective vulnerability analysis than we have been able to perform as part of this study.

Compartmentalization

Due to the current state of software use of CHERI and Morello's compartmentalization feature, it is harder to envision the engineering challenges in this area. It is likely that an initial investment in improving software compartmentalization support in operating systems and the toolchain will pay dividends in simplifying compartmentalization work. Another concern is the potentially greater software disruption from introducing compartmentalization that may perform poorly on conventional hardware platforms, while vendors need to continue to support both. Minimising that disruption may be key to software upstreamability (or longer-term maintainability for a diverged implementation). Tools such as RLBox [23, 24], as they mature, may substantially simplify this task.

That said, this work has shown that a relatively small set of compartmentalization activities would likely substantially mitigate a majority of known vulnerabilities. These should ideally focus on key data processing and communication libraries and classes; for example, compartmentalising:

- Low-level data processing libraries such as giflib, libpng, freetype2, libxml2, and so on, would likely mitigate 100% of known vulnerabilities including denial of service issues. This would benefit multiple complete software stacks including KDE and Gnome.
- Archiving and file management libraries, such as those decompressing file archives that may lack adequate filtering of filenames and files that should be accessible would eliminate several past application-level vulnerabilities.

- Network protocol and processing libraries such as HTTP, HTTPS, and so on would offer protection when interacting with potentially malicious remote servers and clients.
- Higher-level XML and HTML rendering spanning multiple data files (e.g., HTML source, images, …) with embedded language runtimes (e.g., JavaScript) would limit the extent to which security enforced by those interpreters is the only line of defence, by taking steps to mitigate vulnerabilities when processing instant messages, notifications, and so on.

It is not yet clear to us the extent to which compartmentalization should be applied to higher-level class libraries (e.g., image rendering in Qt) vs the lower-level open-source libraries they wrap (e.g., giflib, libpng). Placing compartmentalization in the former offers the great footprint of benefit in terms of consuming applications, but is potentially substantially more disruptive due to the potential need for substantial API change. Placing compartmentalization in the latter, especially when using cleanly engineered C++ APIs, will be vastly easier but limit impact to desktop components that use them -- e.g., KDE, but not Gnome or Chromium. In the interests of rapid deployment of compartmentalization, higher-level compartmentalization might prove a more efficient investment of engineering effort.

12. Limitations of this study

This report describes a pilot study performed over only three months, constraining the approaches we could use, and level of depth we could pursue. It was also constrained by a number of external limitations. It is easy to imagine a larger-scale study engaging specifically with these concerns to increase confidence in the results:

Software stack

Due to time constraints, we were able to focus only on one specific (and narrow) desktop software stack. However, the chosen X11/Qt/KDE stack is a rich software exemplar containing many key elements including a broad spread of C and C++ implementation, code of different vintages and programming styles, and many (sometimes competing) objectives. We had to exclude some key types of software including GPU rendering code such as the kernel Panfrost driver framework and OpenGL in userspace. We were also unable to include more complex applications such as full web browsers and office suites (too complex for our timeline) and video conferencing packages (which are primarily closed source). While we found that the level of effort required to perform our adaptation work was quite low, it is important not to project those results to software components such as language runtimes.

Software adaptations

We identified challenges to CHERI memory-safety adaptation via two means: compiler warnings and dynamic testing. With the former, we were able to rigorously review warnings and correct all problems, which typically included issues such as poor integer-pointer type use. However, some issues could be detected only dynamically, such as certain types of buffer overflows or poor pointer alignment. Our modest enhancements (e.g., CHERI UBsan) improved our ability to debug problems found during dynamic testing. However, we were constrained by test suites with generally poor coverage, and had to rely on manual exercising, which, given the timeline, was necessarily limited. We are not able to estimate the degree to which problems might remain beyond those picked up in our work to date.

Compartmentalization sketches

Our compartmentalization sketches were based on existing familiarity with, and careful analysis of, the affected code bases. However, experience suggests that one can understand the implications of compartmentalized software design only through a detailed implementation effort. It is conceivable that compartmentalization boundaries might need to be placed differently than we have recommended in order to reduce implementation complexity, achieve acceptable performance, and so on. It's also possible that the boundaries we have recommended are not viable in a practical sense.

Because the QEMU software models don't attempt to simulate microarchitecture, they are of limited value for predictions regarding the performance impact of compartmentalization.[63] We have reasonable confidence that our proposed decomposition is placed along sensible lines so as to have acceptable performance, but future experimentation involving a full software elaboration and Morello hardware will be required to determine whether that is the case.

It is also important to understand that, in assembling these sketches, we have not determined that they cannot be implemented realistically without access to CHERI-enabled hardware. It is possible that existing process-based compartmentalization would suffice in some cases, although experience from web-browser compartmentalization research suggests that this is unlikely.

Vulnerability information

Vulnerability information was surprisingly variable in completeness, quality, and level of detail across the software projects we investigated. On the whole, larger projects such as X.org, KDE, and Ubuntu had highly structured vulnerability management procedures making it easy to identify past vulnerabilities and analyse their impacts, even if they did not provide specific threat models or explanations of severity. This made it relatively easy to research vulnerabilities they reported on, including exploring the source-code implications. Some smaller projects, such as giflib and libjpeg-turbo, typically did not have any structured record of past vulnerabilities, causing us to rely on third-party sources of mixed reliability. Others, such as libpng, provided detailed history and analysis, so this is not universally the case.

Throughout our research, it would have been incredibly valuable if a uniform (ideally, machine readable) presentation of vulnerability data had been available. This would have made it easy to find details of the vulnerability and its analysis, any associated patches, severity information, context that allowed the vendor to decide that it required remediation, and cross references to tools used to discover the vulnerabilities, etc.

[63] It is possible to perform limited forms of performance analysis (e.g., instruction-count overheads), but realistic results depend on many other factors such as changes to cache misses, etc.

Vulnerability analysis

Time constraints on this project impacted both the breadth and depth of our past vulnerability analysis. For larger software components with structured vulnerability management and well documented vulnerabilities, reviewing vulnerabilities was straightforward, allowing us to review all potentially relevant vulnerabilities. However, with respect to supporting software libraries, we had to select a sample of libraries, and spend far more time to identify and analyse their vulnerabilities. It is unlikely that our sample is perfectly representative, and care should be taken in extrapolating from this study to other open-source software stacks, or even to the broader set of supporting libraries that we were not able to analyse within this effort.

In general, we had to limit our analysis of past vulnerabilities to relatively coursory classification, relying on existing vendor and third-party analyses to understand the potential impact of each vulnerability. We were not able to inspect source-code changes for each reported CVE, which we would ideally be able to do in a more in-depth study. This may have led to incorrect analyses of the potential impact of CHERI, especially as related to the mitigating effects of software compartmentalization.

If time had permitted, it would have been useful to select a sample of reported vulnerabilities and explore them in greater detail using adversarial techniques (i.e., attempt to exploit them without CHERI protection, to better understand them, and also more concretely explore how CHERI would have helped). This is especially true where vulnerabilities might constitute part of a larger exploit chain, but in isolation are not exploitable to achieve code execution (e.g., some buffer over reads). This would also have allowed us to better estimate the analysis accuracy across the broader corpus.

More generally, some care is required in using past vulnerabilities to predict potential future success for mitigation technologies. New vulnerability classes and exploit techniques arise with moderate frequency, especially as relates to memory-safety vulnerabilities, and it will need to be assessed, as they arise, the extent to which CHERI can be effective in mitigating them.

13. Reproducing our results

We have extended the cheribuild framework to allow the adapted software described throughout this report to be built and used on the QEMU emulator[64] -- albeit very slowly as compared to running it on the forthcoming Morello chip.

On a system that has all required system packages (e.g. compiler, CMake, etc.) installed, the following command should allow compiling a pure-capability KDE plasma desktop[65]:

```
cheribuild.py --include-dependencies kde-x11-desktop-morello-purecap
```

[64] The Arm FVP will also work but is significantly slower, so we prototyped on QEMU.
[65] At the time of writing, there were still outstanding compiler bugs that prevented successful compilation. We have submitted pull requests to the Morello and CHERI LLVM repositories and expect these to be merged shortly,

It is possible this command does not complete successfully, as cheribuild will attempt to build the latest git snapshot for almost all projects. Certain upstream commits (e.g., API changes) could result in a compilation failure for the projects that are currently still using our forked repositories.

In order to run this desktop, you will have to build a disk image that can be started in QEMU, boot that image, start the XVNC server and applications, and finally connect to the VNC server using TigerVNC (or equivalent software) on your host system:

Build a disk image, start it and expose the VNC port to the host:

```
cheribuild.py disk-image-morello-purecap run-morello-purecap \
  --run/extra-tcp-forwarding=5900=5900
```

Once QEMU presents a login prompt, enter "root" to log in (no password required) and then start the VNC server:

```
Xvnc -geometry 1024x768 -SecurityTypes=None &
```

Start a shell with the required environment variables:

```
kde-shell-x11
```

Start the window manager (kwin_x11) and the Plasma desktop:

```
kwin_x11 &
plasmashell
```

Once these programs are running, you should be able to view the CHERI desktop by starting TigerVNC and connecting to `localhost:5900`. Some VNC viewers (such as the built-in macOS VNC viewer) do not allow connections without a password (`-SecurityTypes=None`), so we recommend TigerVNC for now.

14. Recommendations for future research

Our work has highlighted some known, but still key, challenges in CHERI adoption, which would ideally see further research investment:

- As described in an earlier section, **time and other practical concerns imposed a number of limitations on this study**. Continuing this work -- e.g., by increasing the level of breadth and depth of our analysis of vulnerabilities, or prototyping the sketched compartmentalizations, would increase our confidence in, and generality of, the results, as well as our ability to give recommendations about the viability of a CHERI-enabled open-source desktop software stack.
- The adoption of C/C++ memory safety appears relatively straightforward across most application stacks except for **language runtimes**, which require both non-trivial changes, and also would benefit from greater understanding as to how CHERI can improve their robustness. There are also open questions about how to mitigate **potential performance overhead arising from increased pointer sizes**.
- The current **software operational models for compartmentalization are immature**, and require substantial work to make them ready for mainstream use. Further, there is a significant gap around software tooling to assist with designing, implementing, evaluating, and maintaining software compartmentalization of libraries and applications.

- It is also known that compartmentalizing software can be a substantial engineering task, if the software was not originally designed with this in mind. It is not clear to what extent developers are effective in designing and implementing vulnerability-mitigating compartmentalizations, or for that matter maintaining them over time in a changing baseline source tree. The general topic of how to compartmentalize software, and how to evaluate the result, is a topic for substantial further research.
- **This study was unable to concretely engage with performance considerations**, as it took place in advance of Morello board availability. Based on prior work, it is likely that performance overhead will arise primarily in applications with more densely pointer-centric memory access patterns. How best to measure and optimize that impact is currently unclear, and is an important next step in evaluating the suitability of CHERI for vulnerability mitigation in the desktop environment.
- In CHERI memory-protection work to date, the primary focus has been on limiting privilege escalation -- i.e., achieving arbitrary code execution with full user rights, root privilege, or kernel privilege. However, it is clear from reviewing vulnerability advisories that **open-source desktop projects also consider denial of service to be a significant concern**. Software compartmentalization has the potential to address denial-of-service concerns by limiting the scope of a crash or fail-stop (e.g., due to CHERI memory protection), which we believe has been under-addressed in existing CHERI compartmentalization research. This is an area deserving of more research attention, especially in the desktop context.
- Throughout, there remains a key question regarding the **potential for disruption to existing software stacks** -- from Application Binary Interfaces (ABIs), to modest source-code level changes required for memory safety, to more substantial structural changes required for compartmentalization. Evaluating this impact, as well as investigating techniques to reduce it, will be key to successful CHERI software deployment not just in desktop software stacks, but also throughout the broader software corpus and ecosystem.
- In this work we have focused solely on memory protection and compartmentalization for software executing on general-purpose CPUs. Graphics Processing Units (GPUs) are essential parts of contemporary desktop workstations and mobile devices. **There is currently not a good understanding of how CHERI should be integrated with GPUs** -- or how CHERI-enabled software on a general-purpose CPU attached to a CHERI-unaware GPU (such as on the Morello SoC) should protect itself.
- The focus of this work has been software vulnerability mitigation, but hardware vulnerabilities affecting isolation are also important. **Transient execution attacks on CHERI compartments can leak information via side channels** (e.g., caches). Further hardware and software research is required to mitigate this vulnerability using a combination of hardware and software mechanisms. Formal contracts and associated proofs need to be investigated to ensure that key invariant properties of capability-based compartmentalization are preserved at every level of abstraction.

15. Related work

This project takes place within, and extends, a large body of research around vulnerability mitigation, memory safety, and software compartmentalization.

The Microsoft Security Response Center (MSRC)'s 2020 study of the potential impact of CHERI on the Microsoft software stack [27] is the most closely related work, which both analyzes the impact of past vulnerabilities and performs an adversarial analysis of CHERI C and C++, although does not look at the potential cost of software adaptation, nor at compartmentalization. More generally, the CHERI software adaptation literature (e.g., CheriABI [6] and Richardson's 2019 PhD dissertation [17]) considers many of the same concerns with respect to software compatibility, but has been focused on a more classical UNIX server environment; it has provided a less detailed vulnerability analysis, however.

Current directions in CHERI software compartmentalization are not only inspired by, but also directly depend on, the compartmentalization approach developed in our earlier work on Capsicum [21], an OS capability model for sandboxing software. It is also entirely reasonable to contemplate a composition of the co-process compartmentalization model with other OS access-control and sandboxing schemes, such as SELinux [28] or the iOS Sandbox model [29].

Beyond the CHERI research ecosystem, practical memory-safety techniques have played an important role in mitigating vulnerabilities, with the space well described by Szekeres, et al. [30]. Pure software techniques such as Control-Flow Integrity (CFI) [31], and also hardware-based techniques such as Arm's Pointer Authentication Codes (PAC) [32], are starting to see widespread deployment; e.g., within the Windows, Android, and Apple ecosystems. The scopes of these techniques tends to be limited to narrower interventions into attacks on control flow such as Return Oriented Programming (ROP) [33] and Jump Oriented Programming (JOP) [34], rather than seeking to provide more general spatial and temporal memory protection as found in CHERI C and C++. They also tend to be probabilistic and secrets based, so can themselves be subject to vulnerabilities through information leakage or simple brute forcing.

Compiler-based memory-safety and other "sanitizers", such as LLVM's Address Sanitizer (ASAN) [35] and Undefined Behavior Sanitizer (UBSan) [20] have been extremely valuable in identifying potential vulnerabilities during development or in post-development fuzzing. It was clear during our vulnerability review that many had been found using extensive fuzzing exploration while employing ASAN. These techniques tend not to be suitable for more than limited production deployment due to their performance overheads or risks of false positives -- however, it is clear that any vulnerability discovered and prevented before a product is fielded is one that does not need to be mitigated in the field!

16. Acknowledgments

This work was supported by UKRI's Digital Security by Design Research Programme InnovateUK project 107145. We thank Ricardo Almeida, Saar Amar, John Baldwin, Thomas Bauereiss, Kevin Brodsky, Brian Campbell, David Chisnall, Jessica Clarke, Brooks Davis, Nathaniel Wesley Filardo, Brett Gutstein, Simon W. Moore, Peter G. Neumann, George Neville-Neil, and Konrad Witaszczyk for their detailed and extremely helpful feedback on this report. This project would not have been possible without the work of the broader CHERI and Morello teams across SRI International, the University of Cambridge, Arm, and Linaro, as well as their sponsors including DARPA, UKRI, Arm, Microsoft, and Google; we are deeply grateful for their contributions and support.

17. Conclusion

In this report, we've described a three-month long three-staff-month research study into the potential applicability of CHERI memory protection and CHERI software compartmentalization to an open-source desktop stack based on X11 and KDE. During this effort, we adapted a substantial volume of desktop and supporting code to run memory-safe using CHERI C/C++, improved the CHERI C/C++ compiler, and also created a set of "compartmentalization sketches" exploring potential software compartmentalization opportunities within that stack.

We also performed a detailed retrospective study of the potential impact of CHERI vulnerability mitigation on five years worth of vulnerabilities across a subset of the adapted software stack. Our results are extremely exciting: It appears that key elements of the stack were adapted for memory-safe execution with little difficulty (0.026% LoC changed), and that CHERI memory safety and CHERI software compartmentalization had a strong potential mitigation impact (ranging from 40% of past vulnerabilities for KDE up to 100% in key supporting libraries). We described a number of limitations to our study, and key directions for ongoing research and development.

We also recommended a future development strategy premised on rapidly deploying memory safety to see immediate and strong vulnerability mitigation, allowing selected compartmentalization projects to mature the CHERI software compartmentalization infrastructure while offering substantial further mitigation. Overall, we have found that creating a more secure desktop environment for the Arm Morello board, shipping in 2022, is both feasible and likely to offer substantial security improvements over conventional hardware platforms.

18. References

[1] R. N. M. Watson, S. W. Moore, P. Sewell, and P. G. Neumann, 'An Introduction to CHERI', University of Cambridge, Computer Laboratory, Technical Report UCAM-CL-TR-941, 2019. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf

[2] R. N. M. Watson *et al.*, 'Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)', University of Cambridge, Computer Laboratory, UCAM-CL-TR-951, Oct. 2020. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-951.html

[3] R. N. M. Watson *et al.*, 'CHERI C/C++ Programming Guide', University of Cambridge, Computer Laboratory, Cambridge, UK, Technical Report UCAM-CL-TR-947, Jun. 2020. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-947.pdf

[4] Arm Ltd, 'Arm Morello Program', *Arm Developer*, 2020. Available: https://developer.arm.com/architectures/cpu-architecture/a-profile/morello

[5] Arm Ltd, 'Arm® Architecture Reference Manual Supplement Morello for A-profile Architecture', Arm, Manual DDI0606, Jun. 2021. Available: https://developer.arm.com/documentation/ddi0606/latest

[6] B. Davis *et al.*, 'CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment', in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2019, pp. 379–393, doi: 10.1145/3297858.3304042. Available: https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201904-asplos-cheriabi.pdf

[7] N. Filardo *et al.*, ‘Cornucopia: Temporal Safety for CHERI Heaps’, in *2020 IEEE symposium on security and privacy (SP)*, Los Alamitos, CA, USA, 2020, pp. 1507–1524, doi: 10.1109/SP40000.2020.00098. Available: https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/2020oakland-cornucopia.pdf

[8] ‘Department of Computer Science and Technology: CHERI Clang/LLVM and LLD’. Available: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-llvm.html

[9] ‘Department of Computer Science and Technology: CheriBSD’. Available: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheribsd.html

[10] R. N. M. Watson *et al.*, ‘CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization’, in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2015, pp. 20–37, doi: 10.1109/SP.2015.9. Available: https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201505-oakland2015-cheri-compartmentalization.pdf

[11] ‘Department of Computer Science and Technology: CHERI-QEMU’. Available: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-qemu.html

[12] ‘KDE’s Applications’, *KDE Applications*. Available: https://apps.kde.org/

[13] A. Danial, *CLOC*. 2019. Available: https://github.com/AlDanial/cloc

[14] A. Richardson, ‘CTSRD-CHERI/cheri-change-analysis at cheri-desktop’. Available: https://github.com/CTSRD-CHERI/cheri-change-analysis

[15] J. Woodruff *et al.*, ‘CHERI Concentrate: Practical Compressed Capabilities’, *IEEE Trans. Comput.*, vol. 68, no. 10, pp. 1455–1469, Oct. 2019, doi: 10.1109/TC.2019.2914037. Available: https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/2019tc-cheri-concentrate.pdf

[16] A. Richardson, ‘X11 on pure-capability CHERI’, 16-Jun-2021. Available: https://www.alexrichardson.me/post/x11-cheri/

[17] A. Richardson, ‘Complete spatial safety for C and C++ using CHERI capabilities’, University of Cambridge, Computer Laboratory, Cambridge, UK, Technical Report UCAM-CL-TR-949, Jun. 2020. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-949.pdf

[18] ‘Qt | Cross-platform software development for embedded & desktop’. Available: https://www.qt.io

[19] ‘KDE Frameworks’, *Developer*. Available: https://develop.kde.org/products/frameworks/

[20] ‘UndefinedBehaviorSanitizer — Clang 13 documentation’. Available: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[21] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway, ‘Capsicum: Practical Capabilities for UNIX’, in *Proceedings of the 19th USENIX Conference on Security*, Berkeley, CA, USA, 2010, pp. 3–3. Available: https://www.cl.cam.ac.uk/research/security/capsicum/papers/2010usenix-security-capsicum-website.pdf

[22] K. Gudka *et al.*, ‘Clean Application Compartmentalization with SOAAP’, in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2015, pp. 1016–1031, doi: 10.1145/2810103.2813611. Available: http://doi.acm.org/10.1145/2810103.2813611

[23] T. Garfinkel, ‘The Road to Less Trusted Code: Lowering the Barrier to In-Process Sandboxing.’, *;login: the USENIX Magazine*, vol. 45, no. 4, pp. 15–22, Dec-2020. Available: https://www.usenix.org/publications/login/winter2020/garfinkel-tal

[24] S. Narayan *et al.*, ‘Retrofitting Fine Grain Isolation in the Firefox Renderer’, presented at the 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 699–716. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/narayan

[25] A. Richardson, ‘Analysis and compartmentalization of large C and C++ applications’, MPhil Thesis, University of Cambridge, Cambridge, UK, 2015.

[26] S. Amar and N. Joly, ‘Security Analysis of CHERI ISA’, Las Vegas, NV, USA, Wednesday, August 4. Available: https://www.blackhat.com/us-21/briefings/schedule/#security-analysis-of-cheri-isa-23374

[27] N. Joly, S. ElSherei, and S. Amar, ‘SECURITY ANALYSIS OF CHERI ISA’, Microsoft

Security Response Center (MSRC), 2020.
[28] P. Loscocco and S. Smalley, ‘Integrating Flexible Support for Security Policies into the Linux Operating System’, in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, USA, 2001, pp. 29–42. Available: https://www.usenix.org/conference/2001-usenix-annual-technical-conference/integrating-flexible-support-security-policies
[29] Apple, ‘Apple Platform Security’, May 2021. Available: https://manuals.info.apple.com/MANUALS/1000/MA1902/en_GB/apple-platform-security-guide-b.pdf
[30] L. Szekeres, M. Payer, T. Wei, and D. Song, ‘SoK: Eternal War in Memory’, in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2013, pp. 48–62, doi: 10.1109/SP.2013.13. Available: http://dx.doi.org/10.1109/SP.2013.13
[31] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, ‘Control-flow Integrity’, in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, New York, NY, USA, 2005, pp. 340–353, doi: 10.1145/1102120.1102165. Available: http://doi.acm.org/10.1145/1102120.1102165
[32] Qualcomm Product Security, ‘Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions’, Qualcomm Technologies, Inc, Jan. 2017.
[33] H. Shacham, ‘The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)’, in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, New York, NY, USA, 2007, pp. 552–561, doi: 10.1145/1315245.1315313. Available: http://doi.acm.org/10.1145/1315245.1315313
[34] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, ‘Jump-oriented programming: a new class of code-reuse attack’, in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security - ASIACCS ’11*, Hong Kong, China, 2011, doi: 10.1145/1966913.1966919. Available: http://portal.acm.org/citation.cfm?doid=1966913.1966919
[35] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, ‘AddressSanitizer: A Fast Address Sanity Checker’, in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, Berkeley, CA, USA, 2012, pp. 309–318. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

Appendix A: full list of build targets for the desktop

For our desktop demonstrator software stack we compiled the following cheribuild build targets. Each of these targets maps to an upstream Git repository that has successfully been compiled and run as CHERI C/C++. For certain targets we also have to build a version for the host operating system as it provides, for example, build tools required for cross-compilation. These targets are highlighted by a *-native* suffix. In total, we have adapted 150 repositories (with over 6 million lines of C and C++ code) for this work. Two of these repositories had already been ported in previous work and two further repositories had a partial pre-existing port of an older version that we updated and improved. The remaining 146 were newly adapted in this three staff-month project.

attica

breeze

breeze-icons

dejavu-fonts

dolphin

epoll-shim

exiv2

extra-cmake-modules

extra-cmake-modules-native

fontconfig

freetype2

gwenview

icewm

kactivities

kactivities-stats

karchive

karchive-native

kauth

kbookmarks

kcmutils

kcodecs

kcompletion

kconfig

kconfig-native

kconfigwidgets

kcoreaddons

kcoreaddons-native

kcrash

kdbusaddons

kde-x11-desktop

kdeclarative

kdecoration

kded

kfilemetadata

kframeworkintegration

kglobalaccel

kguiaddons

ki18n

ki18n-native

kiconthemes

kidletime

kimageformats

kinit

kio

kio-extras

kirigami

kitemmodels

kitemviews

kjobwidgets

knewstuff

knotifications

knotifyconfig

kpackage

kpackage-native

kparts

kpeople

krunner

kscreenlocker

kservice

ksyndication

ksyntaxhighlighting

ksyntaxhighlighting-native

ktextwidgets

kunitconversion

kwidgetsaddons

kwin

kwindowsystem

kxmlgui

lcms2

libevdev

libexpat *(pre-existing)*

libfontenc

libice

libinput

libintl-lite

libintl-lite-native

libjpeg-turbo

libkscreen

libksysguard

libpng

libqrencode

libsm

libudev-devd

libx11

libxau

libxcb

libxcb-cursor

libxcb-image

libxcb-keysyms

libxcb-render-util

libxcb-util

libxcb-wm

libxcursor

libxcomposite

libxdamage

libxext

libxfixes

libxfont

libxft

libxi

libxkbcommon

libxkbfile

libxmu

libxpm

libxrandr

libxrender

libxt

libxtrans

libxtst

mtdev

okular

openjpeg

phonon

pixman

plasma-desktop

plasma-framework

plasma-workspace

poppler

prison

qqc2-desktop-style

qtbase *(pre-existing, but updated)*

qtbase-native *(pre-existing, but updated)*

qtdeclarative

qtgraphicaleffects

qtquickcontrols

qtquickcontrols2

qtsvg

qttools

qtx11extras

shared-mime-info

shared-mime-info-native

solid

sonnet

sqlite *(pre-existing)*

systemsettings

threadweaver

tigervnc

twm

xbitmaps

xcbproto

xev

xeyes

xkbcomp

xkeyboard-config

xorg-font-util

xorg-macros

xorg-pthread-stubs

xorgproto

xprop

xsetroot

xvnc-server