CHERI-based memory protection and compartmentalisation for web services on Morello

Public Evaluation Report

Dr Graeme Jenkinson, Alfredo Mazzinghi,
and Prof. Robert N.M. Watson
Capabilities Limited

CAPABILITIES
LIMITED

Original version: 2nd October 2023
Last updated: 5th April 2024

This work was supported by Defence Science and Technology Laboratory (DSTL) through the Defence and Security Accelerator (DASA) funded project ACC6036483.

Table of Contents

Executive Summary

This document presents the results of a pilot research project of approximately 4 staff months, with 2 research engineers over 6 calendar months duration, into the challenges and opportunities for the use of CHERI memory safety and compartmentalisation in multi-tenant server environments. In this project, we focus on web services – a software component or system designed to support interoperable machine- or application- oriented interaction over a network. The project's goals are to provide memory safety and fine-grained isolation to web service software components which would be difficult, non-performant, and disruptive to achieve with conventional approaches.

This project has ported a total of 1.7 million lines of server-side software to memory safe CHERI C/C++. The porting effort has been for the most part straightforward, affecting approximately 1% of the total lines of code. The scope and size of the code changes is consistent with previous reports on CHERI C/C++ software porting efforts.

To assess potential performance impact, we ran experiments on Arm's Morello prototype board. Morello is a first-generation implementation of CHERI in an out-of-order processor, and reported overheads should therefore be treated as strict upper bounds in estimating performance on future, optimised processor designs. Detailed microarchitectural experiments at Arm and the University of Cambridge, published in late 2023, suggest that a production design would experience a 80%-88% reduction in CHERI overhead relative to performance on the Morello prototype.

Keeping in mind the limitations of the Morello prototype, our experimental, memory-safe port of the nginx web-server to Morello exhibits a 2% reduction in requests/sec compared to the baseline, non-memory safe version serving a 1kB file of random data from a HTTPs endpoint. The initially unoptimised library compartmentalisation significantly impacted nginx's performance. This was improved to a 4% reduction using a compartmentalisation policy that elides domain transitions that don't benefit security. The performance overhead for our gRPC (a high-performance Remote Procedure Call framework) exhibits a 16% reduction in messages/sec for the insecure and SSL workloads when compared to the baseline non-memory safe version. Library compartmentalisation introduces an additional 12% to 14% reduction in messages/sec, which was reduced down to about 6% to 9% with the use of a custom library compartmentalisation policy. The performance results show that our prototype was able to scale to many millions of domain transitions per second with relatively low overheads on top of CHERI memory safety. Although the measured performance impacts are significant, without further detailed performance work to optimise the current implementation and explore the potential for improvement on a more mature microarchitecture, the root causes for the measured performance overheads remains to be fully explained.

We have reviewed past vulnerabilities for the main components of our software stack. We estimate the impact of CHERI memory safety on a relevant subset of past vulnerabilities, limiting the analysis to those vulnerabilities that are part of the assumed threat model for server-side software components: remote code execution, disclosure of private data, and denial of service. The nginx webserver exhibits a potential mitigation rate of around 46% and Redis (an in-memory cache) between 38% to 52%. Both are in line with previous analysis of past vulnerabilities with respect to CHERI memory safety. For the Postgres database only

between 12% and 18% of vulnerabilities are considered to be mitigated by the CHERI memory safety protections, which can be attributed to the prevalence of access-control vulnerabilities in this software's vulnerability history. We also evaluate the effects of library compartmentalisation on past vulnerabilities observing that a further 15% of nginx vulnerabilities may be mitigated by compartmentalising nginx dynamic modules, increasing the total mitigation rate to 61%; this is under the assumption that the compartmentalisation model can provide at least partial mitigation for denial of service attacks.

As a result of this project, the maturity of the library compartmentalisation model has significantly increased. We provide the first real-world C and C++ use cases of library compartmentalisation which motivated improvements in domain transition observability, as well as performance improvements in the form of globally applied and specialised compartmentalisation policies. In future work, we aim to refine our performance measurements and analysis, building on work by Arm and the University of Cambridge to characterise Morello's performance behaviour, and also by pursuing a measurement and optimization cycle against our software case studies.

1. Introduction

This document presents the results of a pilot research study of approximately 4 staff months, with 2 research engineers over 6 calendar months duration, into the challenges and opportunities for the use of CHERI memory safety and compartmentalisation in multi-tenant server environments, providing memory safety and fine-grained isolation which would be difficult, non-performant, and disruptive to achieve with conventional non-CHERI approaches. In this project we focus on web services - a software component or system designed to support interoperable machine- or application- oriented interaction over a network. In addition to providing a broad scope for exploitation of the project's outputs (both in Defence applications and for the wider Arm Morello community), the focus on web service software stacks was chosen because:

1. They use a relatively small set of common open-source components, constraining the pilot project's scope to something feasible.
2. Memory-safety vulnerabilities are still common within these components. Furthermore, in many cases, modern memory-safe languages only provide a wrapper around these components and some of the benefits of using memory-safe languages are lost. An example of this approach is given by the gRPC project.
3. Many components commonly trade off isolation for performance (processing multiple requests within a single thread); such tradeoffs can be revisited with CHERI-based isolation and systematically evaluated.
4. Individual requests present a natural isolation boundary that is challenging to enforce with conventional isolation technologies without significantly degrading performance.

CHERI memory safety can be applied to C- and C++-language software providing high-levels of mitigation for memory safety issues typically with a low level of disruption to the source code; with the risk of higher levels of "friction" for specific types of software such as language runtimes. This pilot project is the first research study that we are aware of that explores the application of the CHERI protection model in server-side environments. Therefore, one of the project's key goals was to conduct a preliminary exploration to understand whether previously reported estimates of porting effort and security mitigation levels with the CHERI protection model - for example, in desktop software - remain broadly representative for server software. Secondly, the project provides a real-world use case to explore experimental compartmentalisation models. This exploration was performed as a co-design activity with the compartmentalisation model developer, driving improvements based on the feedback and experience of real users.

This project is among first explorations of the security and performance of the CHERI library compartmentalisation model developed at the University of Cambridge. Library compartmentalisation is an experimental feature implemented by the CheriBSD runtime linker by adding an additional level of indirection to the normal runtime linking process via a CHERI domain transition trampoline. As dynamic libraries typically encapsulate a well-defined and distinct set of responsibilities, such as image decoding or cryptographic operations, they present a natural isolation boundary based on subject matter. Furthermore, as many libraries perform operations that are common sources of vulnerabilities and also operate on untrusted data, isolating libraries has the potential to mitigate a significant percentage of real-world vulnerabilities.

Web services are a demanding use case for compartmentalisation, as they seek to balance performance and security goals. CHERI-based compartmentalisation models have potential to significantly disrupt established tradeoffs, for example allowing much-finer grained isolation that would be possible with process or hardware enclaves based approaches. This project will provide a preliminary assessment as to whether CHERI-based isolation can realise that potential in real-world usage.

This report is structured as follows:

- **Background**: provides some background on CHERI C/C++, Arm Morello, and the Library Compartmentalisation model explored in this project.
- **Memory safety**: describes the porting of web service stack software components to CHERI C/C++ on Arm's Morello board, providing a qualitative and quantitative analysis of the porting experience.
- **Library compartmentalisation**: prototyping work exploring the feasibility of employing the CHERI-based *library compartmentalisation* model in web server software.
- **Performance Evaluation**: preliminary performance evaluation results of the gRPC and nginx ports under various software configurations including an optimised default policy developed in response to early project findings. For the purpose of better characterising CHERI performance, we compile the gRPC and nginx benchmarks using the CHERI *benchmark ABI*, this is a modified version of the *pure-capability ABI* that works around a number of shortcomings of the Morello architecture and implementation that are deemed to be fixable in a mature architecture and processor design.
- **Security Evaluation**: an analytical study of past vulnerabilities and their potential mitigation with CHERI-based memory-safety and compartmentalisation for each of the main software components ported in the project.
- **Future research and roadmap**: provides recommendations for future research challenges and opportunities for the use of CHERI memory safety and compartmentalisation technologies in server side.
- **Conclusions**: conclusions of the project.

2. 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 compartmentalisation. 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.

Arm's Morello board, processor, and System-on-Chip (SoC), a Digital Security by Design technology, shipped in early 2022. The Morello SoC is an experimental CHERI-enabled, high- performance, multi-core, multi-GHz design that includes a GPU, and will be the first platform suitable for use as an experimental CHERI-extended server system. As Morello is a first-generation implementation developed as an experimental prototype, it has a number of limitations, including relating to performance, documented in a technical report from Arm and the University of Cambridge, which will contextualise performance and other results in our report.[1]

CHERI C and C++

The **CHERI C** and **CHERI C++** programming languages utilise 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. 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. This provides strong referential safety (protecting pointers) and spatial safety, which we assume in our retrospective vulnerability analysis. In general, when we describe "porting software to Morello," we mean adapting it to compile and run correctly with CHERI C/C++.

Adapting software to CHERI C and C++

Adaptation of contemporary C and C++ source code to CHERI C and C++ is often straightforward, requiring limited minor improvements in C type usage detected by the compiler (for example, to deconflate integer and pointer values).[2] Where the compiler emits a warning or error we are able to rigorously review this and correct. However, some issues only manifest dynamically (at runtime), such as invalidation of capabilities by pointer arithmetic, insufficiently strong alignment in custom memory allocator, non-blessed memory copies (for example, in sorting algorithms), or insufficient pointer alignment. Enhancements such as CHERI UBsan have modestly improved the ability to identify problems previously only found during dynamic testing. However, we are still greatly reliant on dynamic testing. This testing is constrained by both the completeness of the test suites (which in some cases provide poor coverage) and the time available within the project to perform testing. We are not able to estimate what problems might remain beyond those resolved in the scope of the project.

CheriBSD and CheriABI

CheriBSD, 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. Under CheriABI, the kernel, run-time linker, and system libraries cooperate to support the compiler in implementing strong and fine-grained memory protection. In addition to

[1] Robert N. M. Watson, Jessica Clarke, Peter Sewell, Jonathan Woodruff, Simon W. Moore, Graeme Barnes, Richard Grisenthwaite, Kathryn Stacer, Silviu Baranga, Alexander Richardson. *Early performance results from the prototype Morello microarchitecture*, Technical Report UCAM-CL-TR-986, Computer Laboratory, September 2023. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-986.pdf
[2] Robert N. M. Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W. Moore, Edward Napierala, Peter Sewell, Peter G. Neumann. *CHERI C/C++ Programming Guide*. Technical Report UCAM-CL-TR-947, University of Cambridge, Computer Laboratory, June 2020. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-947.pdf

fine-grained memory protection since v22.12 CheriBSD has included experimental support for library compartmentalisation.

Library Compartmentalisation

In the face of sophisticated and well resourced threat actors, it is difficult to prevent the exploitation of a catastrophic software security vulnerability. Software compartmentalisation is a strong form of vulnerability- and attack-independent security mitigation that decomposes larger software applications into isolated components, limiting the privileges gained and further exposed attack surfaces reached by attackers. As dynamic libraries typically encapsulate a well-defined and distinct set of responsibilities, such as image decoding or cryptographic operations, they present a natural isolation boundary based on subject matter. Furthermore, as many libraries perform operations that are common sources of vulnerabilities and also operate on untrusted data, isolating libraries has the potential to mitigate a significant percentage of real-world vulnerabilities. However, it is currently unclear how well the software engineering imposed boundaries of libraries align with the desired security boundaries in practice.

Library compartmentalisation is an experimental feature implemented by the CheriBSD runtime linker by adding an additional level of indirection to the normal runtime linking process via a CHERI domain transition trampoline. In this model, as with models found in managed languages, arbitrary code executing within a library is able to reach only the interfaces and global variables explicitly exposed during the linking process. Policies may then be imposed on appropriate dependencies for a library – for example, permitting side-effect free APIs (pure functions) such as `memcpy` and `memset` unconditionally, but restricting use of I/O APIs for a compute-only library.

By applying the library compartmentalisation to large and complex server SW components we expect that further limitations and implementation issues will be discovered. By working closely with the Cambridge University developer, Dapeng Gao, we will seek to address such issues and improve the maturity of the prototype. In addition, our work is expected to identify opportunities to shape the future development direction of the compartmentalisation model in areas such as the specification of security policy, performance optimisations, and observability.

3. Memory safety

Our starting point in this project was an open-source web service software stack similar to a traditional LAMP (Linux/Apache/MySQL/PHP) running without memory safety on conventional, non-CHERI hardware. In our design, we have replaced Linux with CheriBSD and have chosen a set of comparable software components that are expected to be straightforward to port to CHERI C/C++ on Morello within the scope of the project. The SW stack used in the project includes several popular web service software components such as the nginx web server, Redis cache, and Postgres database in order to analyse the portability and security benefits for real-world, high-performance server software. Porting this SW stack to CHERI C/C++ brings referential and spatial memory safety to code bases that are routinely used in production environments for mission critical applications that demand high-levels of performance and availability.

Initially we had proposed that the popular MySQL server database was included in our software stack. However, during a preliminary investigation of the porting effort, it was found that porting of MySQL resulted in a significant level of friction. Therefore, this component was substituted for PostgreSQL, for which a legacy port to CHERI (performed by Alex Richardson at the University of Cambridge) was already available. Although MySQL server is not part of the project's software deliverables, our experience with porting MySQL is discussed in the qualitative evaluation.

The web service stack ported to CHERI C/C++ is shown in the Figure below:

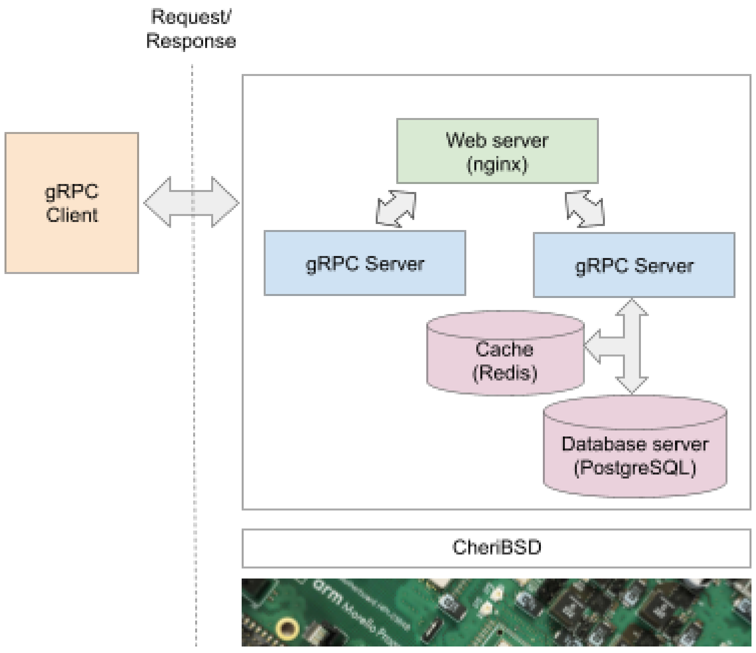


Figure - Web services stack
Summary of the role of the software components that are ported to the Morello platform as part of this project.

The following table shows the web service stack software components and their versions that have been ported to memory-safe CHERI C/C++ within the project:

Component	Description	Version	Status of ported software
nginx	nginx is a web server that can also be used as a reverse proxy, load balancer, mail proxy and HTTP cache (currently most deployed web server accounting for 34.1%[3]).	1.22.0	https://github.com/CTSRD-CHERI/nginx/tree/release-1.22.0-with-cheri-fixes To be included in a future CheriBSD ports release, with support for the core HTTP/s and HTTP v2/3 modules.

[3] Zubin Tavaria. 2021. Now the world's #1 web server, NGINX looks forward to an even brighter future. *NGINX*. Retrieved March 14, 2024 from https://www.nginx.com/blog/now-worlds-1-web-server-nginx-looks-forward-to-even-brighter-future/

Component	**Description**	**Version**	**Status of ported software**
Redis	Redis is an open-source in-memory storage, used as a distributed, in-memory key–value database, cache and message broker, with optional durability (currently most deployed NoSQL database, and 6th most commonly deployed database globally[4]).	7.0.5	https://github.com/CTSRD-CHERI/redis/tree/7.0.5-with-cheri-fixes To be included in a future CheriBSD ports release.
Protobuf	**Google's** language-neutral, platform-neutral, extensible mechanism for serialising structured data.	3.20.1	https://github.com/CTSRD-CHERI/protobuf/tree/v3.20.1-with-cheri-fixes To be included in a future CheriBSD ports release.
μpb	μpb (aka upb) a small **protobuf** implementation written in C.	Commit bef5368	https://github.com/CTSRD-CHERI/upb/tree/upb-grpc-cheri To be included in a future CheriBSD ports release.
gRPC	Cross-platform open source high performance remote procedure call framework, initially created by Google.	1.48.1	https://github.com/CTSRD-CHERI/postgres To be included in a future CheriBSD ports release.

[4] DB-Engines Ranking. *DB-Engines*. Retrieved March 14, 2024 from https://db-engines.com/en/ranking

Component	Description	Version	Status of ported software
Abseil-cpp	Open source collection of C++ libraries drawn from the most fundamental pieces of **Google's** internal codebase.	20220623.0	https://github.com/CTSRD-CHERI/abseil-cpp/tree/cheri-20220623.0 Further porting work required to more cleanly address optimisations in Abseil.
PostgreSQL	Open-source relational database management system (RDBMS) emphasising extensibility and SQL compliance (currently 4th most commonly deployed database globally[5]).	15 beta 4	https://github.com/CTSRD-CHERI/postgres To be included in a future CheriBSD ports release.

Quantitative evaluation

CHERI memory safety is a relatively mature research technology and can be applied to C- and C++-language typically with a low level of disruption (for example as in our port of a desktop software stack), although with the risk of higher "friction" for specific types of software such as language runtimes. It has been shown that the cost of introducing CHERI spatial memory safety is not straightforward to characterise as it depends on factors such as the nature of the software being ported and whether the code base already cleanly supports multiple architectures. For example, high-level software abstractions tend to cause fewer incompatible behaviours when compared to memory allocators and data structures that make assumptions about size, alignment, and binary representation of a pointer.

We provide two estimates for the cost of porting web software stacks to CHERI. First, the total number of staff-hours invested in the porting effort for this project. And second, the number of lines of code (SLoC) that have been changed as a result of the porting work. The former provides a direct measure of the effort. The latter metric provides an indication of the disruption to the code base, this is important for the integration of changes to the upstream project and acts as a proxy measure of the complexity of the changes. We also consider qualitative aspects of the porting experience.

Approximately 1.7 million lines of code are ported to memory safe CHERI C/C++ in the project with a total effort approaching 4 staff months (this is a conservative estimate of effort as it includes our compartmentalisation work and writing of engineering reports). Estimates

[5] DB-Engines Ranking. *DB-Engines*. Retrieved March 14, 2024 from https://db-engines.com/en/ranking

of programmer productivity can be somewhat contentious, but if we assume an optimistic value of a few tens of thousands of lines of code per year (20-35k LoC) it is clear that porting to CHERI C/C++ offers the potential for a very significant productivity benefit. That is not to say that new code shouldn't be written in a memory safe language such as Rust. But CHERI can provide significant memory safety guarantees for existing C/C++ code at a fraction of the cost of fully rewriting the whole code base, complementing existing memory-safe languages. Within this pilot project the measured porting effort for server side software appears to be consistent with results reported for porting other software.

The number of lines of code changed is measured for each repository using the open source `cloc`[6] tool. In all cases, we compare our changes with respect to the unmodified code base. It should be noted that the number of SLoC (source lines of code) changes reported here include test code. This is intentional because in some cases tests have broken due to behavioural differences in the ported code. These behavioural differences are particularly interesting to analyse and are described in the following sections. Furthermore, changes to test code should be considered part of the code-base evaluation because they contribute to the adoption and maintenance friction from CHERI C/C++ features.

The percentage of lines that required changes to adapt the software to the pure-capability CheriABI is minimal. In fact, the number of lines affected by changes sits below 1% in most cases. The figure below shows the number of lines affected by CHERI changes for each source code repository. The table below further differentiates the amount of changes that occurred within the project main software artefacts and the test suites.

Limitations

It should be noted that we had no prior experience or familiarity with the ported code bases prior to the project. Therefore, this estimate of total person hours also includes effort to understand the code bases and the impact of potential changes. Our qualitative assessment of the porting effort suggests that some additional familiarity with the code bases would have been beneficial to reduce the engineering effort required. However, in contrast, our team's significant experience with porting a wide variety of other software to CHERI provides productivity gains that may be absent in measurements taken by other third parties.

The SLoC changed percentage measurements reported in this document have the following limitations:

1. The Morello ports of the aforementioned software are functional but have different degrees of maturity. In particular, issues around CHERI capability representability are known to arise at run-time and (so far) can only be detected during testing. The test suites integrated in the software we ported may not exercise enough functionality to catch all CHERI run-time violations.
2. Some of the ported software components contain dedicated memory allocators. These must be modified to properly enforce capability bounds and representability alignment in order to obtain the full benefits of CHERI spatial memory protection. This is left to future work.

[6] *Cloc: Cloc counts blank lines, comment lines, and physical lines of source code in many programming languages*. https://github.com/AlDanial/cloc

3. We did not test any of the code bases with sub-object bounds enforcement modes. The security impact of sub-object bounds on the software components ported here is also unclear. This is left for future work.

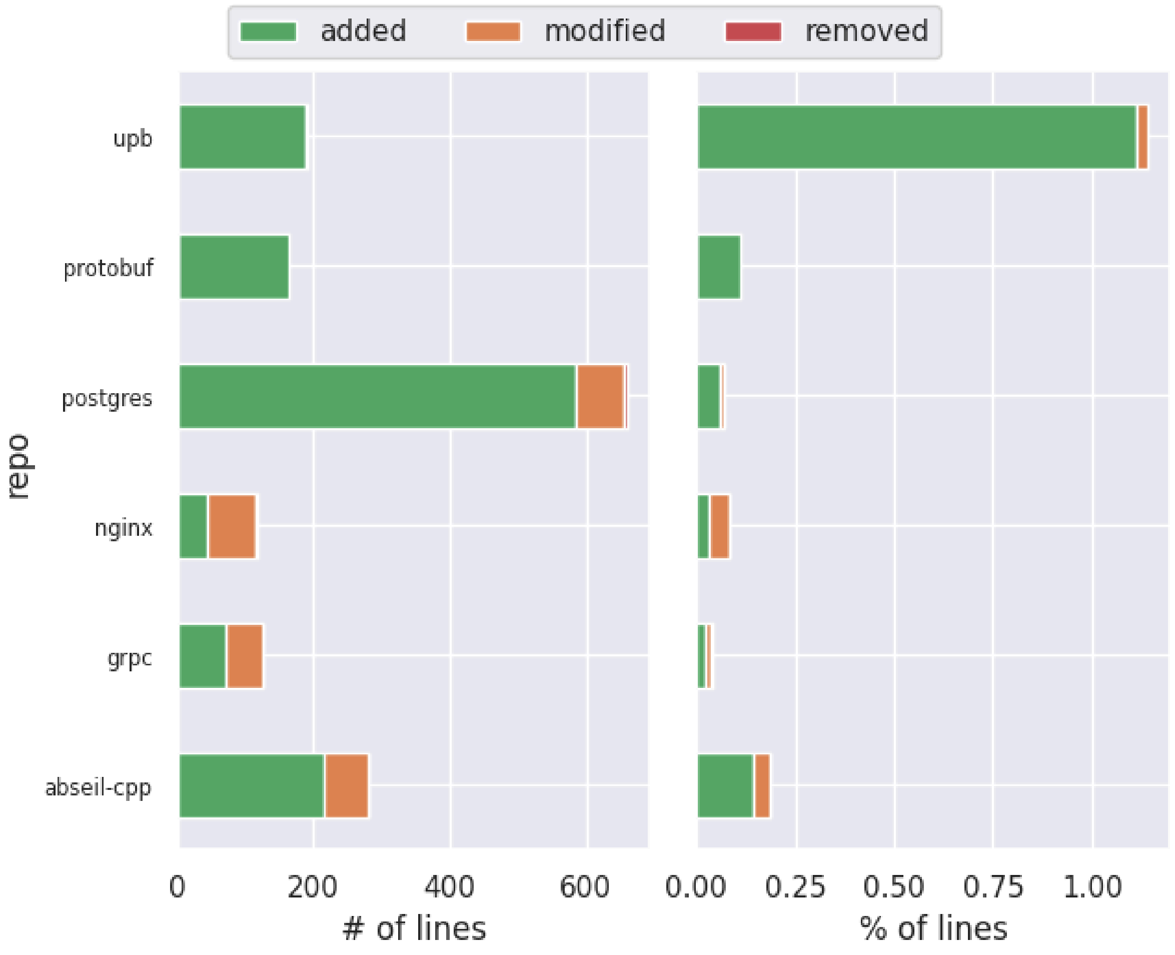


Figure - SLoC changes across projects
This chart shows the number of Source Lines of Code (SLoC) that have been changed as a result of CHERI/Morello support. The chart on the left shows the absolute number of lines that have been affected by changes, while a relative measure of the changes is shown on the right. Because we consider added lines,care should be taken when interpreting the percentage of lines changed. The number of changes here is normalised with respect to the total number of lines of code in the baseline project, however it is conceptually possible to obtain % changes > 100. This would indicate that collectively the size of the changes exceeds the original size of the source code.

Project	**Total SLoC**	**Changed SLoC**	**% Changed SLoC**	**Total files**	**Changed files**	**% Changed files**
abseil-cpp tests	72659	15	0.02	200	3	1.50
abseil-cpp w/o tests	76972	265	0.30	454	15	3.30
grpc tests	165016	32	0.02	830	12	1.40
grpc w/o tests	165102	94	0.10	1350	19	1.40
nginx w/o tests	139804	118	0.10	337	20	5.90
postgres tests	77768	9	0.01	330	4	1.20
postgres w/o tests	872342	651	0.10	2036	33	1.60

Project	Total SLoC	Changed SLoC	% Changed SLoC	Total files	Changed files	% Changed files
protobuf tests	47858	31	0.10	99	3	3.00
protobuf w/o tests	102931	133	0.10	355	10	2.80
upb tests	1353	0	0.00	3	0	0.00
upb w/o tests	15421	191	1.20	44	9	20.50
Total	1737226	1539	0.09	6038	128	2.12

Qualitative evaluation

This section outlines a number of interesting qualitative observations we have made during the porting work. Many of the source code changes are fairly straightforward and stem from well-known patterns and behaviours that are known to not translate cleanly to CHERI. For instance, the Protobuf library was fairly straightforward to port, having only minor issues due to the assumption that a maximum alignment to 8-bytes was sufficient for all architectures. The remainder of this section details portability issues and lessons learned that are particularly interesting or can not easily be categorised as a common CHERI portability issue (such as those covered by guidance in the *CHERI C/C++ Programming Guide*[7]). In general, we observe that server software does not seem to be different from other desktop software in terms of common CHERI portability issues discovered. In particular, support for multiple architectures and proper use of portable data types have been confirmed to be beneficial to the porting effort in this project.

Abseil

The Abseil C++ library provides common abstractions (for example, the `absl::string_view` type defines a common interface to handle reading string data), some of which have been incorporated into or are API compatible with the C++11, C++14 and C++17 standards. The library is maintained by Google and it is used by a number of internal and public Google projects. Abseil is a fairly large C++ code base (77k LoC), which is compiled into a number of component libraries (much like other C++ utility libraries such as Boost). In the context of our Morello web software stack, it is used as a dependency of the Google gRPC library. The majority of this code base is not affected by any portability issues, however we found a few cases in which Abseil makes assumptions about pointer size and representation that require changes. Furthermore, we discovered a possible platform bug involving access to the AArch64 `cntfrq_el0` register on Morello.

Cord optimizations

In the Abseil C++ library, a "cord" is a data structure that represents a sequence of bytes. It is similar to a string in that it represents a sequence of characters, but it is optimised for efficient concatenation and substring operations. Unlike a traditional string, a cord is

[7] Robert N. M. Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W. Moore, Edward Napierala, Peter Sewell, Peter G. Neumann. *CHERI C/C++ Programming Guide*. Technical Report UCAM- CL-TR-947, University of Cambridge, Computer Laboratory, June 2020. Retrieved March 14, 2024 from https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-947.pdf

implemented as a tree of small strings, where each leaf node represents a short (usually less than 256 bytes) substring, and internal nodes represent concatenations of their child nodes. This allows cords to be concatenated efficiently without the need to allocate a new buffer and copy data as only the tree structure needs to be updated. Cords are commonly used in performance-critical applications where string manipulation is a bottleneck, such as text processing, networking, and data serialisation. Abseil's cord library provides a rich set of operations for manipulating cords, including substring extraction, search, and transformation functions, as well as support for conversion to and from other string types.

The Abseil `Cord` implementation assumes that the `sizeof(intptr_t) == 8` (for 64-bit architectures). Furthermore, the `Cord` small-string representation stores a pointer in big-endian order, assuming that the pointer representation will be the same as an unsigned integer. These assumptions do not hold on Morello and lead to the following consequences for the Morello adaptation:

1. Some data types must be changed to use `uintptr_t` in place of `uint64_t`. Misuse of an integer type for a variable that is meant to hold a pointer is a common portability issue for CHERI C/C++.
2. Constants that assume `sizeof(uintptr_t) == 8` must be updated for portability.
3. The size of the `Cord` inline representation as well as the maximum inlined string size are dependent on the pointer size. This means that on Morello there is a behavioural difference in the `Cord` implementation because it will be possible to inline longer strings. In order to minimise behavioural differences, we opted to maintain the same inline string size, despite using twice the space for the inline representation.
4. When the string represented by the `Cord` structure is not inlined, the inline data space holds two pointers. One pointer is used for the actual string buffer allocation; the other points to a `CordzInfo` object which tracks `Cord` statistics for profiling. The `CordzInfo` pointer is stored as a big-endian `uint64_t` value in order to ensure that the low pointer byte aliases with the last byte of the inlined string data. This is no longer possible on Morello, therefore instead the pointer is stored separately after the inline string data buffer; moreover the low pointer bits can still be used to hold the required bit flag. This solution was deemed to balance source code disruption and portability.

The `Cord InlineData` object members change slightly as shown in the following snippet.

```
C/C++
+#if defined(__CHERI_PURE_CAPABILITY__)
+  union {
+      struct {
+      char as_chars_[kMaxInline + 1];
+      cordz_info_t cordz_info_;
+      };
+      AsTree as_tree_;
+  };
```

```
+#else
   union {
       char as_chars_[kMaxInline + 1];
       AsTree as_tree_;
   };
+#endif
```

The `as_chars_` buffer holds the inlined string representation, which is implicitly aliased with the `CordzInfo` pointer in big-endian encoding. The `cordz_info_` member of this implicit union becomes explicit on Morello. This is done to properly account for the pointer field without assumptions about its encoding. The original implementation uses a bit flag, stored as the least significant bit of the `CordzInfo` pointer, to signal whether the string is inlined or not. This is permitted by CHERI and the existing code already guarantees that the `CordzInfo` object is sufficiently aligned, therefore the flag is maintained in the least significant bit of the address in the `cordz_info_` capability.

The technical implementation of the Morello-compatible `Cord` is not complex. It is, however, a good example of how assumptions about the shape and size of a pointer can quickly cause cascading changes to the source code. In this instance, we attempted to avoid behavioural changes due to the changing size of the `Cord` inline string buffer; other implementations with different tradeoffs are possible. Nevertheless, the assumptions on the size of a `Cord` were widespread enough in the Abseil string library that it required changes to many other places, including an optimised copy routine tailored to copying two 64-bit words.

Cords are likely to be used in performance sensitive environments, as a result it is possible that the growth of the `Cord` will have an effect on some workloads. The trade-off between `Cord` structure size growth and the effect of additional space in the inline string buffer is not straightforward to characterise. In particular, it is not known whether the original inline string buffer size was chosen deliberately at an inflection point where increasing the size leads to diminishing returns, or whether the sizing derives from the width of 2 pointers on common 64-bit architectures. A thorough performance evaluation of the `Cord` performance is outside of the scope of this project.

The cntfrq_el0 register

The Armv8-A architecture specification, upon which the Morello CPU is based, includes the *cntfrq_el0* register, which is used to allow unprivileged software to discover the frequency of the system counter. Abseil uses this register as part of its time abstraction interface. While access for the register is configured correctly by the system, it was discovered that Morello requires the current PCC (Program Counter Capability) to grant the *System* permission. This permission is used to enable access to a wide range of system registers and, in CheriBSD, it is never granted to user programs. The requirement of the *System* permission to access an EL0 register is, in our opinion, a platform bug that should be fixed in future architectures.

Within the scope of this project, we modified the CheriBSD Arm timer driver to provide emulation for user accesses of *cntfrq_el0*.

Mutex implementations

Abseil includes a synchronisation library with mutexes and atomic operations. Mutex implementations tend to cause portability issues in the presence of CHERI because in some cases pointer provenance is not preserved correctly. In particular, mutex implementations tend to use a number of the least significant bits of its pointer value to store flags. These flags maintain information on the state of the mutex and are managed with bitwise operations. While these bitwise operations on pointer types are normally commutative, in the presence of CHERI the propagation of pointer provenance is not commutative. In particular, the compiler currently makes a pragmatic choice for compatibility and uses the first operand as the source of pointer provenance.

Consider the following snippet of code showing a simplified part of Abseil mutex code.

```C/C++
PerThreadSync h;
intptr_t nv = (v & kMuEvent) | kMuDesig;

/*
 * This is perfectly legal in C++ and fairly clear code.
 * With CHERI it has the subtle effect of propagating provenance
 * from nv, instead of thread_pointer, leading to tag loss.
 */
nv |= kMuWait | reinterpret_cast<intptr_t>(h);
```

Here, the nv variable is used to construct the new value for the mutex. The variable h holds a pointer to some per-thread state for the thread owning the mutex. Note that nv is initialised from some value v previously read from the mutex. Regardless of whether v is a pointer or not, nv holds a combination of flags because of the bitwise masking operation. When nv is OR-ed with the pointer in h, the provenance will not be propagated from h. Instead, nv is used as the source of provenance for the bitwise OR operation. This occurs because the compiler can not unambiguously determine at compile time which `intptr_t` is the intended source of provenance, therefore the heuristic of using the first operand kicks in and results in propagating provenance from the NULL-derived capability in nv.

The confusion between pointer flags and provenance-carrying values, as well as the operand ordering for `intptr_t` arithmetic are resolved in the Morello port. The following snippet shows how the nv assignment explicitly casts non pointer-bearing values to the appropriate integer type in order to avoid confusion.

```C/C++
/*
 * Note how nv is now cast to ptraddr_t to ensure that only
 * h carries the provenance. In this case the order of the operands
 * does not matter.
 */
nv = reinterpret_cast<intptr_t>(h) | static_cast<ptraddr_t>(nv) |
    kMuWait;
```

It should be noted that the problem of provenance loss, especially around mutex code, is well-known and has been explored in the context of the CheriBSD kernel synchronisation primitives, as well as other libraries. In general, there is no universal solution to fix the confusion regarding the source of provenance that arises from `intptr_t` arithmetic. The current behaviour of the compiler to generate warnings and use the first operand as the default source of provenance is considered sufficient to allow fixing these issues during porting effort.

Limitations

The Abseil Morello port is mature enough to be functional, however there are some limitations of this work that mainly arise from time constraints. These involve both incomplete support for some Abseil functionalities, as well as CHERI-specific concerns. First, there are some known unit-test failures that need to be addressed to complete the porting effort. These involve some features that are not commonly used and indeed are not critical for gRPC. Secondly, the outstanding CHERI-specific issues need addressing including the following:

1. Support for CHERI specific format strings in `absl::StrFormat`.
2. Adding bounds enforcement in the internal allocator interface.

A complete evaluation of the internal allocator interface is particularly important because bounds must be explicitly narrowed by allocator interfaces in order to enforce spatial memory safety among allocations. This is left as future work.

gRPC

The gRPC framework is a modern high-performance language-agnostic remote procedure call (RPC) protocol implementation. gRPC provides efficient, high-performance communication across multi-language environments for both synchronous and asynchronous messaging and streaming of data supporting interoperable machine- or application- oriented interaction over a network. The gRPC framework uses HTTP2 as a transport and Protocol Buffers (or *protobufs*) for its interface definition language. Taking advantage of the cross-language portability of the protocol buffers binary format, it is possible to easily interoperate between clients and services in a variety of programming languages, however, this project focuses on the C++ gRPC library implementation.

The gRPC framework operates with a server and a client. Both the server and client code stubs are generated by the `protoc` compiler from the protobuf interface definition files. The server exposes a number of gRPC services to its clients and each service defines a number of RPC endpoints that can be invoked by clients. The gRPC code base consists of a core library written in a mix of C and C++, alongside the implementation of language-specific bindings. In addition, gRPC depends on a number of external libraries that are also required to be ported to Morello. In particular, the Protobuf library/framework for message serialisation, as well as the internal μpb (micro protobuf) implementation, have necessitated some changes due to assumptions about pointer sizes and alignment requirements.

The gRPC framework itself did not require extensive adaptations to Morello. We have encountered a small number of instances where `intptr_t` has been misused to store integers that are unrelated to the pointer size, such as sequential identifiers, which should use `size_t`, `intmax_t` or a fixed-width integer type if a minimum width is required (this is common issue in code originating from Google). Furthermore, there have been a number of cases where we explicitly introduced casts to remove ambiguity about provenance in `intptr_t` arithmetic operations. This is another common CHERI-related portability problem that has emerged in Abseil as well.

Overall, the gRPC port has shown lower friction than other software projects such as the Abseil library. We believe that this can be attributed to the fact that lower-level abstractions and optimizations are delegated to its dependencies, while the gRPC code base does not make many assumptions about pointer size, alignment and binary representation.

MySQL Server

The Morello software ecosystem currently lacks an up-to-date and high-quality port of a production database, therefore we sought to address that gap in the project by porting MySQL server (version 8.0). As a precondition for the port, various runtime dependencies first needed to be ported to Morello. The most significant engineering effort required in porting dependencies was required by Protobuf (a framework for serialisation of structured data). The upstream Protobuf implementation only supports architectures with a maximum alignment of 8-bytes, and therefore the majority of the changes made to Protobuf introduce support for stronger alignment (to `max_align_t`) as shown below[8]. In addition it was necessary to work around a missing `ostream<<` operator in the platform's standard C++ library for printing `intptr_r/unitptr_t` types; this issue has subsequently been resolved in CheriBSD 23.11.

```
C/C++
#if defined(__CHERI_PURE_CAPABILITY__)
struct alignas(max_align_t) TcParseTableBase {
#else
struct alignas(uint64_t) TcParseTableBase {
#endif
```

[8] *Protobuf@f98f86d*. Retrieved March 14, 2024 from https://github.com/CTSRD-CHERI/protobuf/commit/f98f86d2659e5ad34b0d82773b560827f4eeded5

During the porting of MySQL server undefined behaviour (that is, behaviour where the compiler is free to do whatever it chooses) was identified when configuring options. For example, the configuration options for SSL (TLS) that configure the file containing the certification authority certificate:

```
C/C++
static Sys_var_charptr Sys_ssl_ca(
  "ssl_ca", "CA file in PEM format (check OpenSSL docs, implies --ssl)",
  PERSIST_AS_READONLY GLOBAL_VAR(opt_ssl_ca),
  CMD_LINE(REQUIRED_ARG, OPT_SSL_CA), IN_FS_CHARSET, DEFAULT(nullptr),
  &lock_ssl_ctx);
```

The `Sys_var_charptr` type used to store the configuration option is a C++ class, sub-classed from the `sys_var` type (declared in `set_var.h`). The `sys_var` class is used to set configuration variables that both affect SQL sessions (stored in a `System_variable` structure), and global variables such as those to configure SSL (TLS) settings. In a flawed attempt to to handle both of these cases with the same code, the `sys_var` type stores an offset (of type `ptrdiff_t`) to the configuration variable. For configuration options that affect the sessions, this is valid as the offset is within the same data structure. However, for global variables the offset used is from an arbitrary and unrelated address in memory. For example, in the case of the `Sys_ssl_ca` configuration parameter the offset to the configuration variable is computed using the `GLOBAL_VAR` macro shown below:

```
C/C++
#define GLOBAL_VAR(X)                                  \
 sys_var::GLOBAL, (((const char *)&(X)) - (char *)&global_system_variables), \
   sizeof(X)
```

Computing values using the difference of two unrelated pointers (that is pointers not within the same data structure) results in undefined behaviour in C. In this instance, when accessing a global configuration variable its address must be recomputed by adding its offset to the address of the `global_system_variables`. As CHERI capabilities cannot be calculated by arbitrary arithmetic operations this invalidates the capability tag, which results in a CHERI protection fault on dereferencing the capability.

Removing the identified undefined behaviour from MySQL requires substantial changes to the configuration option handling code. After some effort, attempts to work around the issue did eventually allow progress to be made. However at this point a further issue was identified with data representations across layers in the database, and it was felt that the port of MySQL server was likely to remain a high-friction activity and uncontainable within the budget and timescales of the existing project. Therefore, it was decided (in agreement with the technical authority at DSTL) to pivot to the database porting activity to Postgres; as an earlier port of Postgres had been performed at the University of Cambridge the risk of this was believed to be lower. However, the selection of easier, low-friction SW for porting potentially biases the quantitative evaluation work potentially producing an overly optimistic estimate of the man hours required to adapt software to CHERI C/C++. We are keen to explore issues around SW selection and its impact on porting effort in further detail in future studies.

Postgres

The existing CHERI port of Postgres (created by Alex Richardson at the University of Cambridge) is both significantly out-of-date (version 9.6 which is now unsupported) and was performed with an obsolete CHERI C/C++ interpretation of capabilities as offsets from a base rather than addresses, which resulted in a large number of changes that are no longer necessary (for further details see ‘Complete spatial safety for C and C++ using CHERI capabilities’[9]). Therefore, it was decided to start from scratch with a new port of Postgres 15 beta 4, matching the version found in the CheriBSD 23.11 ports collection.

One of the changes made previously that is also required for the new port is modification of the Postgres quicksort implementation to preserve capability tags (specifically in the case where the input to the sort is an array of capabilities to the objects that are to be sorted). The quicksort implementation found in PostgreSQL is based on the design described by Bentley and McIlroy in their paper “Engineering of Sort Function”.[10] This design is also found in other software components that have been ported to CHERI C/C++ such as in glib and Redis, and it is also found in CheriBSD `libc`. As the CheriBSD `libc` quicksort has already been successfully modified to preserve capability tags, this version can be easily substituted for the version provided by Postgres. However, the sort comparator API differs between the two implementations resulting in the requirement to adapt each of the sorting comparator functions in Postgres. Since Postgres 9.6, several new implementations of sorting algorithms have been introduced, including a new “interruptible” quicksort (that is, the sort can be stopped at set checkpoints) and a partial quicksort. Given the proliferation of sort algorithms, breaking the sort comparator API has become highly disruptive requiring numerous changes across the code base. However, the CheriBSD `qsort_s` implementation does match the comparator API used by Postgres. Thus our new Postgres 15 beta 4 port replaces the internal `qsort` with `qsort_s` in CheriBSD’s `libc`.

Adapting the new interruptible quicksort to preserve capability tags is more problematic. The interruptible `qsort` is created by instantiating a templated algorithm implemented with preprocessor macros. Although adapting the templated `qsort` is possible, a pragmatic approach (requiring less effort) was taken where the templated sort is instantiated for each different type where capability tags need to be preserved. This change may need revisiting when looking to make the project’s outputs available to the wider Morello community.

[9] Alexander Richardson. *Complete spatial safety for C and C++ using CHERI capabilities. Technical Report UCAM- CL-TR-949, University of Cambridge, Computer Laboratory, June 2020*. Retrieved March 14, 2024 from: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-949.pdf
[10] Jon L. Bentley M. Douglas Mc ILROY. *Engineering a Sort Function*. *Fit.edu*. Retrieved March 14, 2024 from https://cs.fit.edu/~pkc/classes/writing/papers/bentley93engineering.pdf

Given the recurrence of issues with preserving capabilities in sort algorithms, common advice should be added to the *CHERI C/C++ Programming Guide*[11] to prevent repeated re-engineering.

The `Datum` type (defined as `uintptr_t`) is used internally in Postgres for representing any valid SQL type. Increasing the `Datum` size, which has doubled in size in the CHERI C/C++ port, is likely to have a significant impact on performance and memory utilisation of Postgres. However, evaluation of those impacts and exploration of potential optimisations minimising their impact is beyond the scope of the existing project.

Executing the Postgres test suite shows a number of tests where the query plan differs in the Morello port from its expected value. For example, in the example shown below the query plan favours a performing sequential scan (`Seq Scan`) of the database over constructing and using an index that does not reference the table's main data area (`Index Only`). Determining the query plan uses a set of heuristics that are themselves dependent on the performance of the platform, for example, the time to perform either a sequential or random page read. Therefore, it is possible that any differences in query plans are simply the result of a legitimate behavioural change. However, such differences could also simply result from a bug introduced by our porting. Within the budget and time constraints of the project it has not been possible to investigate all of the changes to the query plans (5 in total) witnessed in the Postgres test suite. However, we have looked in some detail at the example shown below and present the findings here.

```
Unset
Conflict Filter: (SubPlan 1)
    ->  Result
    SubPlan 1
-     ->  Index Only Scan using both_index_expr_key on insertconflicttest ii
-           Index Cond: (key = excluded.key)
+     ->  Seq Scan on insertconflicttest ii
+           Filter: (key = excluded.key)
```

To analyse the differences, the output from the heuristics used to construct the query plan on Morello was compared to a build of Postgres running on MacOS. This showed that the estimated costs of performing an `Index Only` scan were identical in both builds but that the cost of a `Seq Scan` was estimated as being lower on Morello. Furthermore, the small difference in the cost of `Seq Scan` on Morello was sufficient to favour this over an `Index Only` scan. On Morello the estimated cost of performing a `Seq Scan` is lower as a result of the *tuple density* being lower; a Postgres tuple corresponds to a row within the database and the tuple density being the number of tuples per block/page (where block/pages are 8kB subdivisions of physical table storage). Each Postgres tuple is aligned to the value

[11] Robert N. M. Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W. Moore, Edward Napierala, Peter Sewell, Peter G. Neumann. *CHERI C/C++ Programming Guide*. Technical Report UCAM- CL-TR-947, University of Cambridge, Computer Laboratory, June 2020. Retrieved March 14, 2024 from https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-947.pdf

`MAXALIGN`. As CHERI capabilities require a stronger maximum alignment, this results in a higher value of `HEAP_OVERHEAD_BYTES_PER_TUPLE`, which in turn results in the lower tuple density and lower estimate for the `Seq Scan`. Whilst this analysis may not be generalisable to the other case where the query plans differ, it does suggest that they are most likely legitimate behavioural changes resulting from changes to heuristics used for query planning. It should be noted that in this example the test case itself is fragile as the test outcome is perturbed by minor changes to the heuristics from the platform's stronger alignment requirements.

The Postgres regression test suite currently shows one unresolved crash when processing indirect *toasts*. A toast is a large blob of data managed outside the database row, and indirect toast is one in which the memory for the data blob is managed independently of Postgres. In the test indirect toasts are referenced from the database tuple with a pointer/capability. However as this value isn't guaranteed to be correctly aligned, dereferencing results in a CHERI protection fault. As this is a very minor feature we have not undertaken a fix for this, although it is thought to be relatively straightforward.

4. Library compartmentalisation

One of the main results of this work is the first exploratory analysis of the implications of CHERI/Morello compartmentalisation models for web software stacks. For the purposes of this analysis, we focused on the library compartmentalisation model prototype, which aims to provide the ability to enforce isolation boundaries at the shared library granularity. We hypothesise that library compartmentalisation provides a good trade-off between ease of adoption, flexibility of bounds enforcement, performance and security guarantees. It should be noted that a complete in-depth analysis of this trade-off space offered by library compartmentalisation is impossible at this point. This is because the compartmentalisation infrastructure is in its early stages of development and this project has been its first "real-world" consumer. Nevertheless, it has been possible to provide valuable early insight and feedback that has driven further development and refinement of the library compartmentalisation model.

Library compartmentalisation and C++

The existing work on library compartmentalisation has so far focused on C applications, along with the C standard library and run-time infrastructure. One of this project's main contributions consists of the first test and characterization of the existing library compartmentalisation model for C++ libraries and applications. In particular, both gRPC and Abseil exercise a significant portion of the C++ run-time and standard libraries.

It is possible to identify two main areas in which this work has improved our support and understanding of the library compartmentalisation model for C++:

1. Several bug fixes and improvements to the domain transition trampolines have been motivated by failures triggered by gRPC.

2. gRPC demo services and test workloads provide exploratory data to understand the interactions of C++ language features, such as lambda functions and virtual inheritance, with the library compartmentalisation approach.

The gRPC demonstrator has shown that it is possible to run C++ applications under library compartmentalisation with minimal effort, however it has raised a number of questions about how the compartmentalisation boundary interacts with C++ features. As previously introduced in the Background section, the current library compartmentalisation model relies on the run-time linker to generate domain transition trampolines that intercept cross-library function calls via the ELF PLT (Procedure Linkage Table). While this mechanism works for both C and C++, it has a known limitation when function pointers are shared across libraries and called directly bypassing the PLT. Recent C++ specifications introduce support for lambda functions and the `std::function` abstraction of function pointers. These are expected to suffer from the same limitation as raw function pointers with respect to domain transitions. Moreover, depending on lambda capture and implementation details, it is unclear whether it may be possible that some state is unexpectedly leaked to another compartment.

The handling of C++ exceptions across library compartments is not yet supported, however this is not considered to be a fundamental limitation of the model and will eventually be implemented.

Finally, C++ class method calls across compartment (library) boundaries require some consideration. As an example, assume that a class B in the main program inherits from class A which is part of a library. Consider now the call of a method of class B from the main program compartment. Methods defined by A will be called normally through the PLT, regardless whether these are virtual or not. Methods defined in B will not cause any domain transition. Furthermore, the class data layout on AArch64 is such that both the main program and the library will have access to all public and private members of the class, regardless of whether the member is inherited from A or declared in B. As a result, we believe that the use of C++ classes across compartment boundaries should be assessed carefully to ensure that the application is not accidentally leaking state into a library compartment through shared object instances.

Limitations

Exploratory work on library compartmentalisation with C++ is currently limited to the gRPC demonstrator, therefore there are clear limitations on the generality of our observations. Nevertheless, this is an important step towards improving the library compartmentalisation model maturity for C++ applications, as well as exploring the ways in which compartment boundaries can be defined effectively.

nginx compartmentalisation

Since version 1.9.11, nginx supports dynamic modules compiled as shared libraries. nginx's first party modules encompass core functions such as handling the mail and streaming protocols as well as a variety of other functions from profiling to geolocation of IP addresses, in addition there are many third-party modules. As nginx modules encapsulate a well-defined and distinct set of responsibilities, such as decompression or authorization, they present a natural isolation boundary based on subject matter that can be enforced by library compartmentalisation. Whilst several first party nginx modules can be built as dynamic libraries, the core nginx modules such as protocol state machines for http currently can not (largely because up to this point there has been no motivation to do so). However, this is not a fundamental limitation and could be resolved with some small additional engineering effort (though this effort was not containable within the scope of this project).

Library compartmentalisation was successfully applied to nginx by modifying the binary's runtime linker as described in the CheriBSD c18n man page. Although adoption of library compartmentalisation is extremely straightforward it remains unclear what this achieves for security or how it impacts performance. Therefore Sections 5 and 6 present an initial analysis of performance and security for nginx with library compartmentalisation. In our security analysis of nginx we assume that compartmentalisation of core modules such as http state machines is present, but other than that there are no further attempts to modify or adjust existing library boundaries. In our performance analysis no attempt is made to actually compartmentalise the http protocol state machine though other supporting libraries for example for cryptography are compartmentalised.

gRPC compartmentalisation

Our exploration of compartmentalisation opportunities for gRPC applications focuses on two aspects of the library compartmentalisation model. First, we want to assess the feasibility of using library compartmentalisation to isolate all libraries in an existing gRPC application. Second, we explore a technique that allows us to enforce CHERI compartment boundaries around gRPC services running within the same server. Both these experiments have allowed us to provide feedback for the development of library compartmentalisation in multiple co-design cycles.

In order to simplify our understanding of library compartmentalisation with gRPC, we considered a simple service that responds to ping requests. This demonstrator server program links a total of 64 libraries, including the gRPC core and C++ libraries, a number of Abseil component libraries and an SSL library. Each of these libraries becomes a separate compartment.

This first experiment confirmed that no disruption to the source code was needed to enable library compartmentalisation. This is one of the key advantages of this model, although it remains important to evaluate the security implications.

Early results from compartment transition tracing has revealed that there are various opportunities for optimization.

1. A relatively large number of the domain transitions are caused by system functions in `libthr` and `libc`, such as `memset()` and `memcpy()`. Some of these functions do not access any state in the respective library, these are either *pure functions* or operate only on a state object provided as one of the arguments. We hypothesise that such functions may be executed without a domain transition by incurring in minimal or no loss of security guarantees.
2. As previously mentioned, a simple gRPC service links a large number of libraries. We observe that the library boundary sometimes does not coincide with the desired compartment boundary. For example, 45 out of 64 libraries linked by our gRPC demo

program are Abseil components. We argue that it makes little sense to isolate all Abseil components from each other, because multiple shared libraries make up one logical compartment.

As a result of these two observations, the compartmentalisation run-time linker has been extended to provide a few important features. First, there is initial support to group libraries into compartments by specifying a policy. Second, libraries that are part of the TCB (Trusted Computing Base), such as `libc` and `libthr` are automatically grouped. Finally, a number of well-known pure functions, such as `memcpy()`, are executed in the context of the calling compartment, without a domain transition. These changes initially made to in an experimental version of CheriBSD are available in the 23.11 release,

Initial observations from compartment transition traces suggest that there are substantial gains to be obtained by optimising the grouping of libraries in logical compartments, as well as identifying functions that do not necessitate a domain transition. In our simple ping demonstrator program, the optimisations stated above reduce the number of domain transitions by approximately 78%, from about 490,000 to just below 105,000 (these preliminary results were recorded without the Benchmark ABI). The same set of optimisations have a lesser effect on the QPS benchmark ping-pong unauthenticated streaming workload, where we observe a 40% reduction in the number of domain transitions.

Library compartmentalisation policies allow to group multiple libraries into a single logical compartment. This is a technique useful to tune the trade-off between security and number of domain transitions. In this case the shared library granularity for compartments allows to create a meaningful logical compartment boundary. It is interesting to discuss the opposite situation, where a single program or library must be split in order to isolate one of its components. In the context of gRPC, we argue that the *service* abstraction provides a good example of how it may be possible to leverage library compartmentalisation to isolate different API surfaces. Consider for example a gRPC server that exposes two services, depicted in the figure below. The *chat* service manages less sensitive information, while the *payments* service handles the exchange of money between users. It is desirable to limit access to the libraries and interfaces used for financial transactions to only the *payments* service.

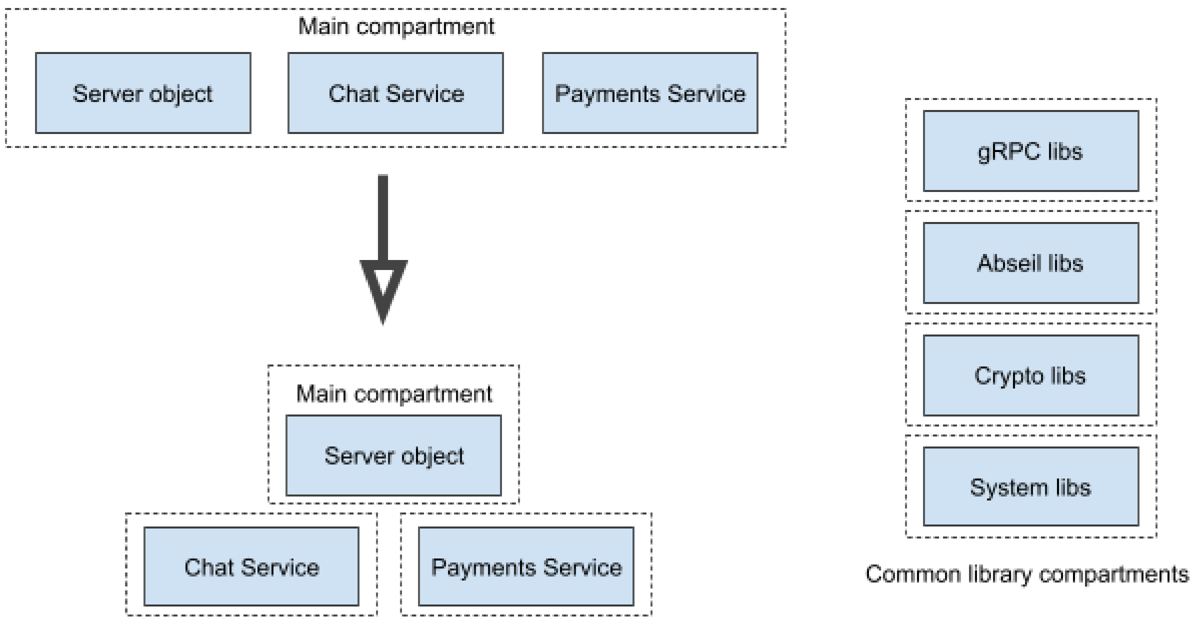


Figure - Use of library compartmentalisation for custom isolation boundaries
Here the main compartment of a gRPC server application is split into two separate compartments. This is possible with minor source code disruption because the gRPC services interface allows to build each service as a separate shared library.

In the first configuration, the *chat* and *payments* gRPC services are built within the main application. It is relatively straightforward to separate the services implementation into two separate libraries. In this way, the *payments* service may maintain exclusive access to cryptographic material and API keys that allow financial operations, while any vulnerability in the *chat* service has a reduced impact.

It is important to note that this approach does not require a major redesign of the application and leverages the existing design and abstractions of gRPC. It is interesting to show that library compartmentalisation provides opportunities for a more arbitrary definition of compartment boundaries, however it is important to caution against overgeneralization. Software compartmentalisation is a notoriously difficult task in general, due to performance, source code disruption and maintenance trade-offs. The findings in this project suggest that CHERI/Morello library compartmentalisation provides a new trade-off space to further explore.

5. Performance Evaluation

Our memory safe experimental port of the nginx web-server (configured with HTTPs) exhibits a modest 2% reduction in requests/sec compared to the baseline, non-memory safe version. The performance overhead for our gRPC (a high-performance Remote Procedure Call framework) port is more significant. The QPS throughput benchmark exhibits approximately a 16% reduction in messages/sec for the insecure and SSL workloads, when compared to the baseline non-memory safe version.

The performance of nginx and gRPC with library compartmentalisation degrades by about 2.5% and 12-14%, when compared to the memory-safe configuration. We show that significant amounts of the performance overhead can be recovered by carefully optimising away domain transitions that do not significantly impact the security properties of the model. This is implemented by a policy system that can be specified at the system level, as well as optimised specifically for an application. When the policies are applied, the performance overhead of library compartmentalisation is reduced to within 5-9% of the memory-safe gRPC configuration.

Finally, we simulate a lower-bound overhead for an equivalent compartmentalisation model built on top of process compartmentalisation. The simulated IPC yields an approximate 65-73% reduction in throughput on top of the memory-safety overhead for gRPC using non-CHERI compartmentalisation. With a slightly smaller overhead of around 40% for nginx. Further exploring the differences in nginx and gRPC with respect to the performance of library compartmentalisation policy is outside the scope of the current project.

Performance of the Morello hardware prototype

Arm's Morello processor, System-on-Chip, and board are experimental prototypes developed on a shortened timescale as part of UKRI's Digital Security by Design research programme. Recent work on Morello performance[12] has sought to isolate the fundamental performance overheads of the CHERI protection model from the microarchitectural limitations on performance introduced by the first-generation integration of the protection model with the baseline Armv8.2-A architecture in Morello. That work has identified a number of microarchitecture bottlenecks that significantly impact performance, for example, untuned **store throughput and buffer sizes** for 128-bit capability values. To better isolate performance measurements from such limitations of Morello, a set of ABI specifically for performance measurements have been developed: Benchmarking, P128, and P128 Forced-Got compilation ABIs; these ABIs should only be used for performance measurements as their code-generation weaken or remove the security properties of the CHERI protection model.

This performance evaluation of nginx and gRPC uses the newly developed performance evaluation ABI (referred to as *Benchmark ABI),* shipped in CheriBSD v23.11, in order to provide a more accurate evaluation of the overhead of both memory protection and library compartmentalisation. These benchmarks are expected to be near worst-case scenarios for CHERI, because the selected message size is small and the workload should be dominated by metadata processing, which is known to be sensitive to pointer size increase. However, with the fundamental overheads of CHERI capabilities are estimated at around 1-2%, the performance impact we see for nginx and gRPC are both significantly higher than might be expected. It is therefore valuable to undertake more detailed performance work that can seek to understand these results.

In the performance report, Arm and the University of Cambridge describe a series of performance optimizations performed on a modified FPGA instantiation of the Morello design, combined with the P128 ABIs, that achieve a 80%-88% reduction in CHERI overheads compared to what is achievable on the shipped Morello board using the Benchmark ABI. However, this modified design was not available for experimental work in

[12] *Early performance results from the prototype Morello microarchitecture*. *Github.io*. Retrieved March 14, 2024 from https://ctsrd-cheri.github.io/morello-early-performance-results/cover/index.html

the current version of our report. It is not possible to naively assume that the same overhead reduction would be applicable to these workloads, but it is reasonable to presume that some significant performance improvement would be achieved. Early performance results presented in this section should be interpreted in this light. We hope that the workloads we describe may be run on Arm's modified FPGA implementation in the future.

nginx

The performance evaluation of the experimental nginx port uses the `wrk` benchmarking command line tool[13]; replicating the performance setup used previously by F5 to benchmark nginx[14]. The `wrk` tool is used to request a 1kb binary file (containing random bytes) from the nginx server, a performance measurement is taken with a single thread and 50 concurrent connections over a duration of five minutes, as shown below. The nginx configuration has been tuned to achieve a representative level of performance, for example, disabling the access log and logging only critical errors, and increasing the worker connections based on the expected load. The `wrk` command line tool is on the same Morello board as the nginx server, connecting via the localhost interface. For the duration of the performance measurements the Morello box is performing no other user activities. The Morello board runs a CheriBSD development kernel preview as the compartmentalisation (c18n) features are experimental at this time.The CheriBSD version used for this benchmark run corresponds to the git SHA c7c430099bbe.

```
Unset
./wrk -t 1 -c 50 -d 1m --latency https://192.168.2.2/rps/1kb-random.bin
```

In our setup, the `wrk` benchmark program is always compiled using the *hybrid ABI*, as the focus of the benchmark is to measure server performance. Furthermore, there are two patches applied to `libcrypto` and `rtld`. The former disables ARMv8 cryptographic instructions for the *hybrid ABI* version of `libcrypto`; this is necessary because CheriBSD doesn't yet support such extensions for the *pure-capability* and *benchmark* ABIs. The latter produces a variant of `rtld` where the IPC overhead emulation is always enabled; this is necessary because the `nginx` process forks and the `LD_C18N_COMPARTMENT_OVERHEAD` environment variable is not propagated.

The chart below shows the measured performance for four software configurations:

1. Purecap with memory safety - the project's experimental port.
2. Purecap with memory safety and default library compartmentalisation policy. In this configuration libraries that are part of the TCB (Trusted Computing Base), such as `libc` and `libthr` are automatically grouped and a number of well-known pure

[13] Will Glozer. *wrk: Modern HTTP benchmarking tool*. Retrieved March 14, 2024 from https://github.com/wg/wrk
[14] Amir Rawdat. 2017. *Testing the performance of NGINX and NGINX Plus web servers. NGINX*. Retrieved March 14, 2024 from https://www.nginx.com/blog/testing-the-performance-of-nginx-and-nginx-plus-web-servers/

functions, such as `memcpy()`, are executed in the context of the calling compartment, without a domain transition.
3. Purecap with memory safety and a fixed overhead approximating a lower-bound of conventional IPC overhead (IPC overhead is emulated by a single syscall executed in the domain transition and configured by the `LD_C18N_COMPARTMENT_OVERHEAD` environmental variable).

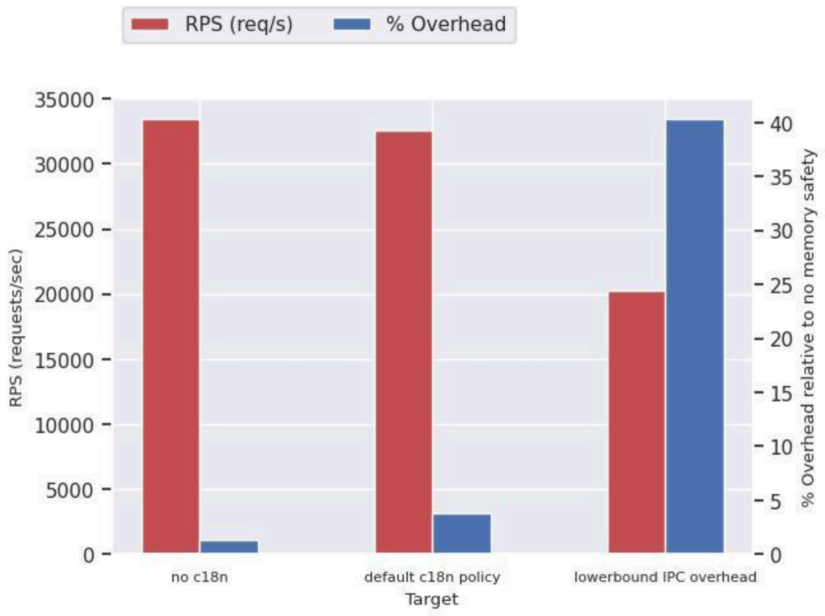


Figure - WRK benchmark results over an encrypted connection.
This figure shows the absolute number of requests/sec obtained by statically serving a random 1KiB file. The right Y axis shows the relative performance overhead with respect to the WRK benchmark built for the CHERI hybrid ABI.

The project's memory safe experimental port exhibits a ~2% reduction in requests/sec compared to the baseline hybrid version running without memory safety. This can be considered a negligible impact that falls within the expected overhead observed in other CHERI-enabled applications evaluated using the Benchmark ABI.

During our co-design activity, we discovered that the initially unoptimised library compartmentalisation significantly impacted nginx's performance highlighting the importance of exploring potential optimisation opportunities. To guide this analysis an experimental tracing feature, logging entry and exit of each compartment using `utrace`, was added (at our request) to the library compartmentalisation prototype. To enable the tracing set the environmental variable `LD_C18N_UTRACE_COMPARMENT=1`, as shown below:

```
Unset
sudo LD_C18N_LIBRARY_PATH=/usr/local/lib LD_C18N_UTRACE_COMPARTMENT=1 ktrace -t
u /usr/local/nginx/sbin/nginx -c /usr/local/etc/nginx/nginx.conf
```

On analysing the collected traces, it was clear that many transitions occur into trusted libraries such as `libc` and/or to pure functions (without side effects) such as `memset` or memcpy. Gao was able to use this collected trace data to develop a default policy that reduces the overall number of compartment transitions and therefore improves performance without introducing significant changes to the security properties provided by compartmentalisation. In this policy the libc and libthread are considered part of the Trusted Computing Base (TCB) and located in the same compartment. In addition, pure functions such as memset are located in a global compartment and can be called without the overhead of a compartment transition.

```
Unset
61645 nginx  USER RTLD: c18n: enter libc.so.7 on thread 0x0 at [0] memset (0x40e141f9)
61645 nginx  USER RTLD: c18n: leave libc.so.7 on thread 0x0 at [0] memset
61645 nginx  USER RTLD: c18n: enter libc.so.7 on thread 0x0 at [0] readlink (0x40d7ebcd)
61645 nginx  USER RTLD: c18n: leave libc.so.7 on thread 0x0 at [0] readlink
61645 nginx  USER RTLD: c18n: enter libc.so.7 on thread 0x0 at [0] issetugid (0x40d7f311)
61645 nginx  USER RTLD: c18n: leave libc.so.7 on thread 0x0 at [0] issetugid
61645 nginx  USER RTLD: c18n: enter libc.so.7 on thread 0x0 at [0] getenv (0x40da7c1d)
61645 nginx  USER RTLD: c18n: enter libc.so.7 on thread 0x0 at [0] strncmp (0x40e152f1)
61645 nginx  USER RTLD: c18n: leave libc.so.7 on thread 0x0 at [0] strncmp
61645 nginx  USER RTLD: c18n: enter libc.so.7 on thread 0x0 at [0] strncmp (0x40e152f1)
61645 nginx  USER RTLD: c18n: leave libc.so.7 on thread 0x0 at [0] strncmp
```

Library compartmentalisation with the new default policy incurs approximately a 4% performance decrease over the non-memory safety (baseline) version. This is significant improvement for early optimisation work and suggests further work on optimisation may produce further significant improvements.

Finally, we attempt to estimate the impact of a traditional process-based compartmentalisation technology, which uses IPC for communication between compartments. This is simulated by inserting a system call in the CHERI domain transition trampolines, with the goal of measuring the cost of entering the kernel for the volume of domain transitions that occur at the granularity level offered by the CHERI library compartmentalisation model. In the case of this nginx workload, a traditional IPC mechanism would impose at least 40% overhead with respect to the baseline, without memory safety. It is important to highlight that the estimated overhead for a traditional IPC mechanism is a strict lower bound. In fact, the cost of a real IPC implementation will always be higher than our estimate, because we only consider the cost of transitioning to the kernel. In a real-world implementation, there would be additional overhead to context switch to another process, copy the data and handle synchronisation for the IPC channel.

gRPC

The performance evaluation of the gRPC port uses the QPS benchmark bundled with the gRPC internal test suite. This benchmark is designed to run in a Google Kubernetes Engine cluster in the Google Cloud infrastructure, however it is possible to run locally with a minimal number of benchmark workers on the same Morello machine. The QPS benchmark measures the messages/second rate combinations of workloads. Our evaluation focuses on a throughput benchmark scenario that sets up 64 concurrent channels between the client and the server and maintains a queue of 100 outstanding 8-bytes asynchronous streaming gRPC messages for the duration of the benchmark. This same scenario is tested both with and without SSL for gRPC transport security. Each measurement is repeated 10 times.

Methodology

The QPS benchmark is implemented as a worker service `qps_worker` and a driver `qps_json_driver`. The driver loads a JSON scenario file describing the benchmark to run. We use two running instances of `qps_worker`, one will act as the gRPC server, the other as the client. The benchmark is executed as shown below:

```
Unset
qps_worker --driver_port=10000 &
qps_worker --driver_port=10001 &
export QPS_WORKERS=localhost:10000,localhost:10001
qps_json_driver --scenarios_file /path/to/scenario.json
```

The scenario files are obtained from the `scenario_config_exporter.py` tool in the gRPC project repository; the scenario files are unmodified. For the duration of the benchmark, no other activity is running on the Morello board. The only active service is `sshd`, which is used to connect to the board and run the benchmark script. The Morello board runs a CheriBSD development kernel preview as the compartmentalisation (c18n) features are experimental at this time. The CheriBSD version used for this benchmark run corresponds to the git SHA c7c430099bbe.

Similarly to the nginx workload study, we disable cryptographic instructions acceleration for all targets, so that there is no difference in `libcrypto` across the *hybrid* and *benchmark* ABIs.

The benchmarks are used to measure the performance for six software configurations:

1. Hybrid without memory safety - baseline.
2. Purecap with memory safety - the project's experimental port.
3. Purecap with memory safety and default library compartmentalisation policy. This is the same policy described for `nginx`, which has been introduced as a result of the co-design activity during this project.
4. Purecap with memory safety and application-specific compartmentalisation policy. This policy extends the default compartmentalisation policy to address application-specific needs (described below).
5. Purecap with memory safety and a fixed overhead approximating a lower-bound of conventional IPC overhead (IPC overhead is emulated by a single syscall executed in the domain transition and configured by the `LD_C18N_COMPARTMENT_OVERHEAD` environmental variable).

Results

The charts below show a summary of the performance results from the QPS benchmark. The first observation is that the memory-safety performance overhead is more significant compared to nginx. The QPS benchmark throughput exhibits a significant reduction of around 16% for both the secure and insecure communication channels, compared to the baseline without memory safety. In contrast, library compartmentalisation incurs a more modest performance overhead on top of the memory-safety common overhead. The default c18n compartmentalisation policy incurs an additional throughput overhead of about 15% for the SSL case and 12% for the insecure case, on top of the memory safety overhead.

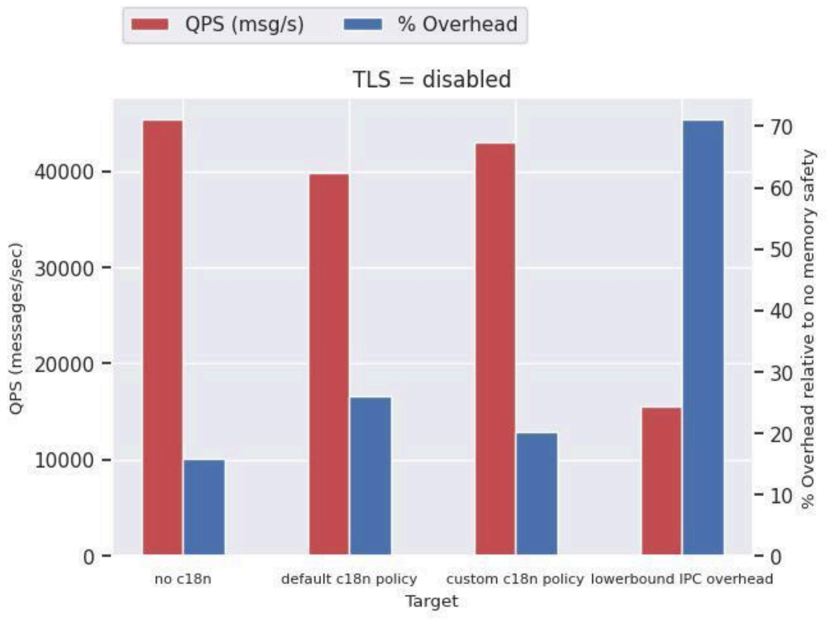


Figure - gRPC QPS benchmark results over an un-encrypted connection.
This figure shows the absolute number of messages/sec for the QPS 8-Byte message workload over an unencrypted channel. The right Y axis shows the relative performance overhead with respect to the QPS benchmark built for the CHERI hybrid ABI.

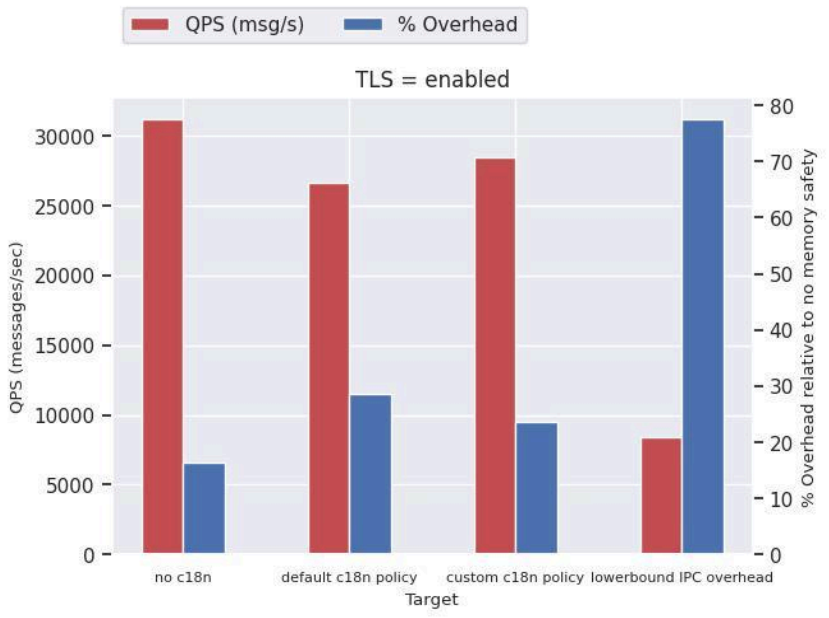


Figure - gRPC QPS benchmark results over an encrypted connection.
This figure shows the absolute number of messages/sec for the QPS 8-Byte message workload over an unencrypted channel. The right Y axis shows the relative performance overhead with respect to the QPS benchmark built for the CHERI hybrid ABI.

The gRPC library links a relatively large number of libraries, which in the default compartmentalisation policy are mapped 1:1 to CHERI compartments. Upon analysis of domain transition traces, we noticed that it would be possible to group multiple libraries into the same logical compartment, so that function calls between libraries within the same group do not cause a domain transition. This is an extension of the default policy, which implicitly creates a group with the core system libraries, currently including `libc` and `libthr`. We created a new policy explicitly tailored for gRPC. This policy creates 4 additional library groups:

1. A group containing all Abseil libraries. This is justified by the fact that these libraries belong to the same software component and there is little security benefit in isolating them from one another.
2. A group containing `libssl` and `libcrypto`.
3. A group containing all gRPC libraries. This includes the core grpc and C++ language-specific interface.
4. A group containing the core C++ run-time libraries, including `libc++`, `libcxxrt` and `libgcc_s`.

The custom policy shows that it is possible to reduce the overhead to around 9% and 5% for the secure and insecure QPS benchmarks respectively, on top of the memory-safe prototype.

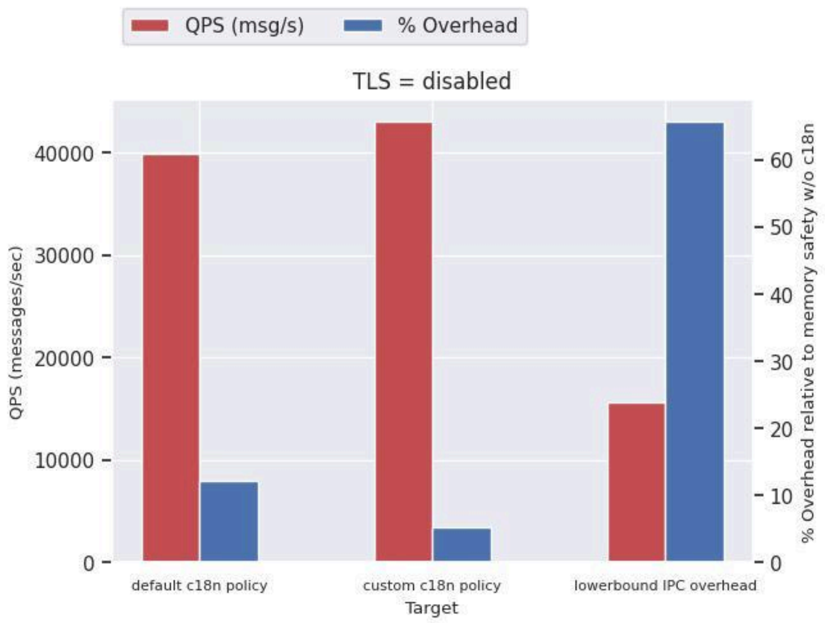


Figure - gRPC QPS benchmark results with overhead computed with respect to the memory-safe prototype. The use of a custom compartmentalisation policy is effective at bringing the library compartmentalisation overhead under 10%, while a traditional process compartmentalisation is estimated to surpass 65~70% overhead.

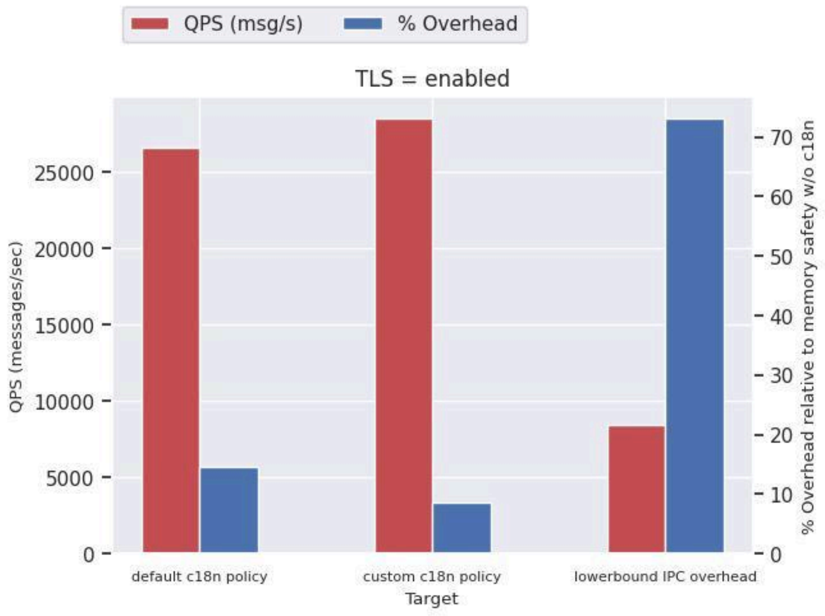


Figure - gRPC QPS benchmark results with overhead computed with respect to the memory-safe prototype. The use of a custom compartmentalisation policy is effective at bringing the library compartmentalisation overhead under 10%, while a traditional process compartmentalisation is estimated to surpass 65~70% overhead.

Finally, it is important to note that the library compartmentalisation model enables a domain transition rate that would not be possible with process-based compartmentalisation. In order to estimate the cost of process-based compartmentalisation in this prototype, we show the effect of a single (`getpid`) system call inserted in the domain transition trampoline. The goal of this experiment is to show a lower-bound estimate of the cost of a process-compartment switch, which would at least incur the cost of entering the kernel once for every domain transition. The `getpid` system call is a good candidate for this, as it does not require any synchronisation within the kernel and only introduces the system call overhead. As shown in the figure above, the IPC overhead simulated in this way exhibits a 65% to 73% throughput reduction compared to the memory-safe prototype, for the un-encrypted and encrypted channels respectively.

The library compartmentalisation model is implemented on Morello using the *executive-restricted* mechanism; when a library function is called, the CPU switches to *executive* mode and executes the domain transition trampoline. The trampoline proceeds to switch the compartment state and returns to *restricted* mode into the target library function. By counting the number of transitions from *restricted to executive* mode, we obtain an estimate of the number of compartment switches. In order to estimate the number of compartment switches, we use the `EXECUTIVE_ENTRY` hardware performance counter available on the Morello platform. This counter measures the number of transitions from the restricted to executive modes that are speculatively executed.

There are three limitations of this approach:

1. The compartmentalisation trampolines are not the only cause for entering *executive* mode. In particular, the run-time linker runs entirely in *executive* mode, therefore whenever execution is transferred to `rtld` (e.g. for lazy symbol resolution) we will

observe a mode switch to *executive*. The effect of these additional switches should be insignificant for the benchmark steady-state.

2. The `EXECUTIVE_ENTRY` counter measures speculative events. This means that the actual number of retired instructions that involve a mode switch may be lower than the amount reported.
3. Due to implementation limitations, the `EXECUTIVE_ENTRY` counter is not meaningful when using the *Benchmark ABI*. We use measurements from the *pure-capability* QPS variant to measure the counter. This should not significantly affect the number of transitions per message, because the number of compartment switches should primarily depend on the amount of library function calls performed by the software. We do not expect this to change between the QPS benchmark variations under test.

The QPS benchmark over an insecure channel performs an estimated 900 domain transitions per RPC message exchanged. These correspond to around 450 function calls, as the `EXECUTIVE_ENTRY` counter accounts for both the forward and return edges of the call. The use of a custom compartmentalisation policy saves around 200 ex_entry/msg.

The same workload over a secure channel performs approximately 1500 domain transitions per RPC message (corresponding to ~750 function calls) using the default compartmentalisation policy. A custom policy results in 300 fewer ex_entry/msg, this is slightly higher than the insecure case, however it is consistent with the observation that our policy places `libcrypto` and `libssl` in the same logical compartment, therefore saving some additional compartment switches.

These numbers, together with the traditional IPC overhead simulation, show that there is a significant amount of additional work (amounting to 12 million domain transitions) performed as a result of library compartmentalisation. With this experiment, we demonstrate that the CHERI library compartmentalisation prototype being developed is able to scale to many millions of domain transitions per second with relatively low overheads on top of CHERI memory safety.

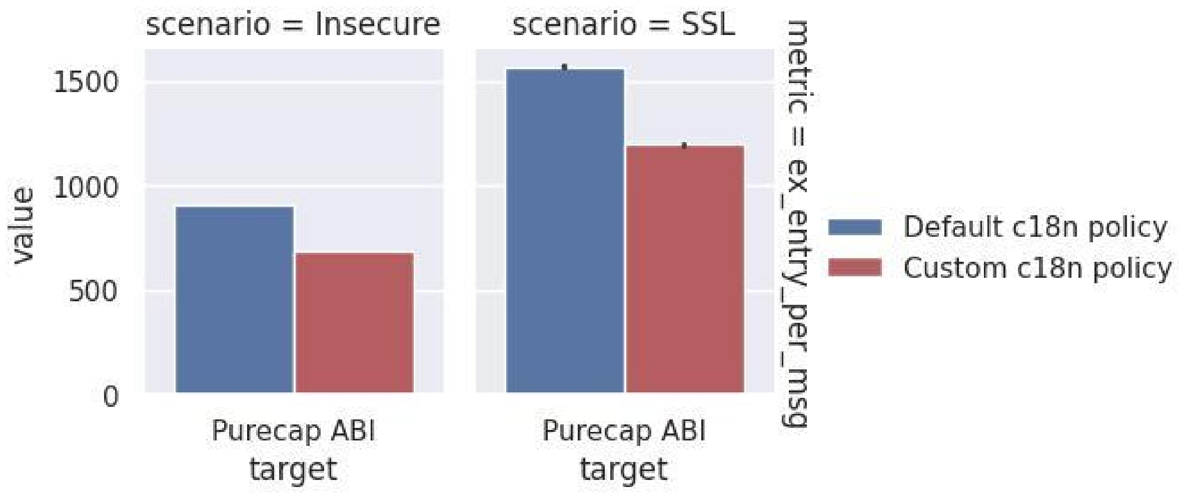


Figure - Reported value of the `EXECUTIVE_ENTRY` per RPC transaction.
The figure shows the ratio of the `EXECUTIVE_ENTRY` performance counter over the total number of RPC transactions completed. Similarly to previous measurements, we report the data for 8-byte messages over both insecure and SSL-encrypted channels. The measurements are however performed using the pure-capability ABI

QPS variant. The score reported in this plot takes into account both the forward and return edges of a library function call; therefore a score of 1000 ex_entry/msg corresponds to 500 function calls.

6. Security Evaluation

The security evaluation will review historic security vulnerabilities from nginx, Redis, Postgres to assess whether our proposed or actual adaptations would have reduced the severity of the vulnerabilities. This evaluation will primarily be performed as an analytical study rather than an adversarial penetration testing exercise due to time and scope constraints, but reflects reasonable best estimates of the impacts of CHERI memory protection and compartmentalisation (as described). In the study we consider:

- Actual adaptations
  - Referential and spatial memory safety resulting from porting the software component and its dependencies to CHERI C/C++.
  - Library compartmentalisation of existing dynamically loadable nginx modules and nginx shared libraries.
- Proposed adaptations:
  - Heap temporal memory-safety, released in CheriBSD 23.11.
  - Library compartmentalisation of core nginx modules that are not currently dynamically loadable such as the http module.

The raw data collected and used in our security evaluation can be found in Appendix B.

Security advisory information sources

For each software component, 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 (for example, nginx security advisories[15]). However, in cases where a project doesn't maintain such a list, we turn to externally maintained lists such as CVE Details[16] or the National Vulnerability Database (NVD).[17] Vulnerabilities in supporting libraries are rarely documented by the vendors themselves, in these cases OS vendor advisories and issue trackers such as those from Ubuntu[18] and RedHat[19] were used. In our analysis, we indicate the primary source(s) of vulnerability information used for each software component.

[15] nginx security advisories. *Nginx.org*. Retrieved March 14, 2024 from https://nginx.org/en/security_advisories.html
[16] CVE security vulnerability database. Security vulnerabilities, exploits, references and more. *Cvedetails.com*. Retrieved March 14, 2024 from https://www.cvedetails.com
[17] National Vulnerability Database (NVD) - home. *Nist.gov*. Retrieved March 14, 2024 from https://nvd.nist.gov
[18] Security notices. *Ubuntu*. Retrieved March 14, 2024 from https://ubuntu.com/security/notices
[19] Product Security Center. RedHat. Retrieved March 14, 2024 from 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.
- **Threat model:** the assignment of the vulnerability to the project's stated thread model (described below).
- **CWE(s):** The unique software/hardware weakness identifier(s) reported by National Institute of Standards and Technology (NIST) for the CVE.
- **Assessment**: Our brief assessment of the potential impact of CHERI memory protection and compartmentalisation on the vulnerabilities. Where vulnerabilities contain insufficient information, 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 we also note that here.

Threat model

Most open-source projects do 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, we assume projects are concerned with remote code execution, private data disclosure, and denial of service.

Remote code execution

For example, in nginx CVE-2021-23017 in which a malicious DNS response can trigger an off-by-one error within the `ngx_resolver_copy()` leading to arbitrary code execution. Or CVE-2022-31144 in which a remote attacker can pass specially crafted data to Redis, triggering an heap-based buffer overflow and execution of arbitrary code on the target system.

Private data disclosure

The Private Data Disclosure category relates to library or application logical bugs in which private data is improperly disclosed. For example, in nginx CVE-2018-16845 in which a specially crafted mp4 file can result in the `ngx_http_mp4_module` disclosing worker process memory.

Denial of service

Denial-of-service vulnerabilities are often assigned a lower severity than vulnerabilities leading to arbitrary code execution. However, DoS vulnerabilities still featured prominently in security advisories for web service stack components. This is particularly true for web servers such as nginx, where the implications of a crash could be significant for the larger application or set of applications being served. 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. A partial mitigation is recorded where the CHERI protections reduce the severity of an issue, for example downgrading a RCE to denial of service resulting from a deterministic crash caused by the CHERI protection fault.

Vulnerability analysis

Due to the limited timeline and scope of this project, we will rely heavily on the vulnerability analyses provided by the software vendors (for example, in the vendor's own revision-control history or vulnerability announcement), or by a downstream software distribution (for example, Ubuntu or Redhat analysis). A more rigorous study would perform more in-depth studies of the specific code paths, ideally with an adversarial element to evaluate practical exploitability.

Advisories that fall outside of our stated threat model such as: downgrading of the security of the network connection, MITM attacks, and installer vulnerabilities are not considered in this analysis. Any other advisories excluded from the analysis are explicitly mentioned along with a justification for their exclusion.

Previous studies

Several previous whiteboard security evaluations of the CHERI protection model have been performed, for the purposes of comparison of historical reported mitigation rates we refer to two previous studies: "Security Analysis of the CHERI ISA"[20] performed by Microsoft and "Assessing the Viability of an Open-Source CHERI Desktop Software Ecosystem"[21]

[20] Nicolas Joly, Saif ElSherei, Saar Amar. Microsoft Security Response Center (MSRC) . SECURITY ANALYSIS OF CHERI ISA. Retrieved March 14, 2024 from: https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Security%20analysis%20of%20CHERI%20ISA.pdf

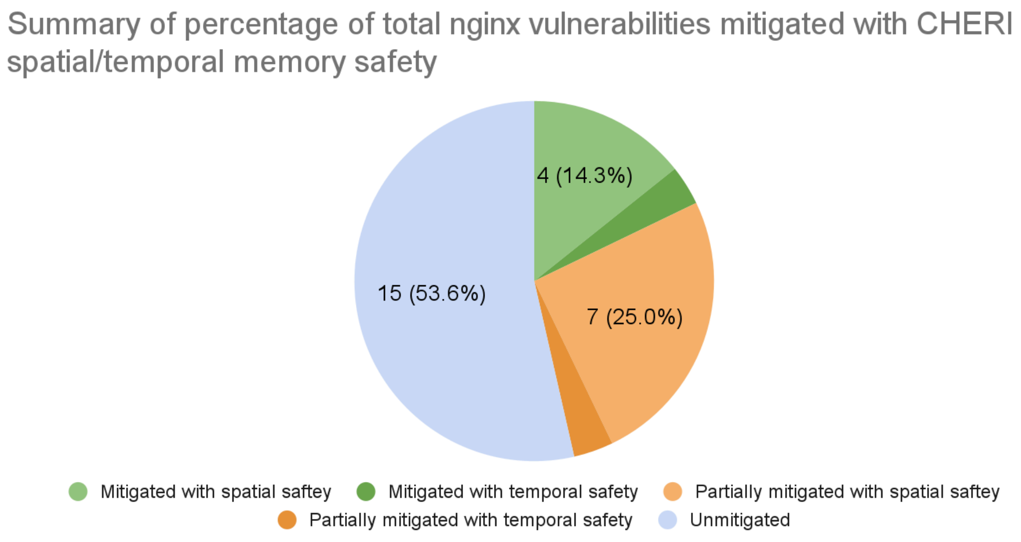
[21] Robert N. M. Watson, Ben Laurie, and Alex Richardson. Capabilities Limited. Assessing the Viability of an Open- Source CHERI Desktop Software Ecosystem. Retrieved March 14, 2024 from https://www.capabilitieslimited.co.uk/_files/ugd/f4d681_e0f23245dace466297f20a0dbd22d371.pdf

performed by Capabilities Limited. In Microsoft's evaluation they report that CHERI memory safety had a strong potential mitigation impact: “If we consider combining CHERI's current protections with the additional mitigations recommended in this document, such as stack initialization and heap initialization, including padding added to allocations due to bound compression, we estimate that the protections would extend to deterministically mitigating nearly half of the MSRC vulnerabilities we addressed through a security update in 2019.”. Similarly Capabilities Limited's desktop study reports that CHERI memory safety and CHERI software compartmentalisation had a strong potential mitigation impact, ranging from 40% of past vulnerabilities for KDE up to 100% in key supporting libraries.

In the desktop study compartmentalisation is assumed to provide a degree of mitigation for some classes of denial of service vulnerabilities by limiting portions of applications affected by software termination, including when crashes originate with memory-safety violations. However, in the library compartmentalisation model library failure has the scope of the process and therefore cannot (currently) provide any such mitigation; a key ongoing area of research at Cambridge and SRI is how to provide for recovery following library failure, which this project will help motivate. For the purposes of this evaluation we assume that the compartmentalisation model can limit software termination to a specific compartment without requiring restarting of the application. However, for anything other than stateless libraries cleanly restarting a library on failure presents significant challenges, and may either result in loss of data or require significant changes to the application to prevent data loss. Therefore, we classify this a partial mitigation.

nginx

The figure below shows a breakdown of nginx vulnerabilities and their mitigation by spatial/temporal memory safety provided by the CHERI protection model; note that this analysis excludes the following Windows specific advisories: CVE-2011-4963, CVE-2010-2263, and CORE-2010-0121. This shows that the mitigation rate of security vulnerabilities in nginx with CHERI spatial/temporal memory protection is approximately 46%. This mitigation rate is broadly consistent with the ranges reported in previous studies described above.

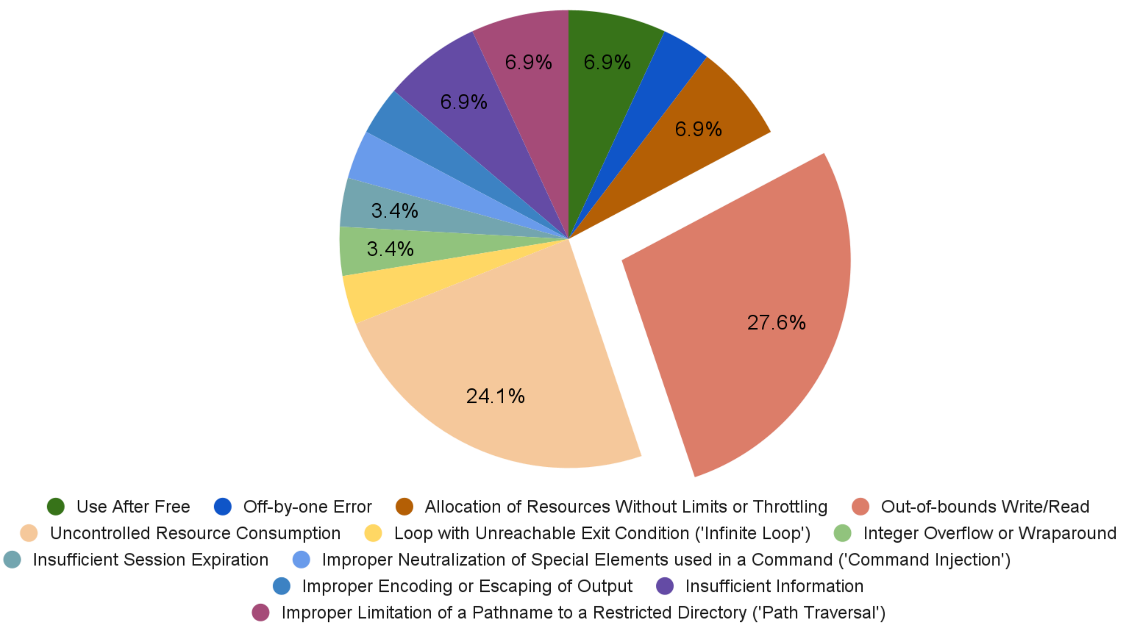


Approximately 28% of CWEs (Common Weakness Enumeration) assigned to nginx security advisories relate to buffer overwrites and overreads[22]. The prevalence of buffer overwrites and over reads in the nginx security advisories accounts for the potential high levels of mitigation provided by the CHERI protection model. It should also be noted that multiple CWEs are often recorded against a single vulnerability, for example where an integer overflow results in the buffer overwrite this is often recorded with both CWE-190 (Integer Overflow) and CWE-787 (out-of-Bounds Write). This accounts for the overall mitigation rate for nginx being higher than the percentage of CWEs for buffer overwrite/overreads.

Use after free weaknesses are relatively rare, with just two in the analysed data set. However, it is believed that CHERI temporal memory safety protections would provide strong mitigation for both of these issues.

Uncontrolled resource consumption is the second largest weakness assigned to nginx vulnerabilities; with many of these being in the http2 protocol module. As the denial of service of a webserver has the potential to impact a large number of users, the incentives for a denial of service (DoS) attack on a server differ significantly than, for example, in a desktop environment. The differing incentives may result in DoS being more prevalent in server software with such attacks being assigned greater significance and thus have more prestige for security researchers. However, without further analysis it is impossible to say whether server software stacks in general possess greater rates of denial of service attacks, or what impact this has on potential rates of mitigation from the CHERI protection model.

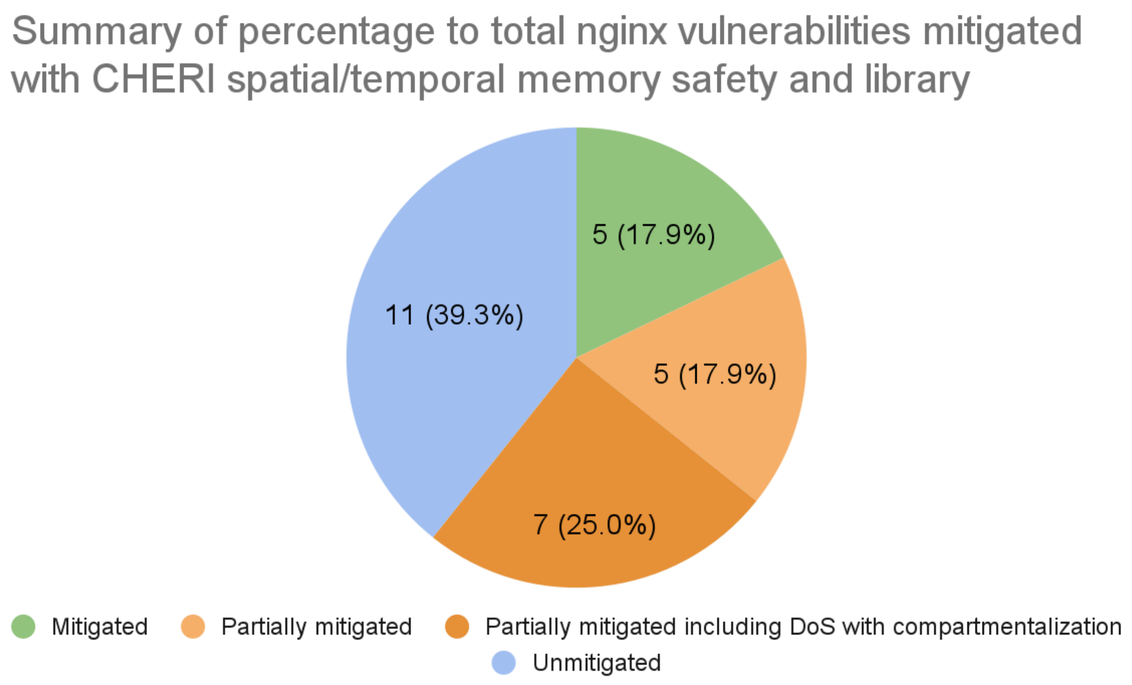
Summary of the assigned CWE (Common Weakness Enumeration) to nginx security



Applying library compartmentalisation to nginx modules improves the potential total mitigation rate to 61% (up from 46% with CHERI spatial and temporal memory safety alone), see the figure below. As discussed above where compartmentalisation is used to mitigate a

[22] Note that several of the assigned CWEs address very similar weaknesses “Out-of-bounds Write” (CWE-787), “Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')” (CWE-120), and “Improper Restriction of Operations within the Bounds of a Memory Buffer” (CWE-119) all of these CWEs are grouped in our analysis as buffer overwrites or overreads

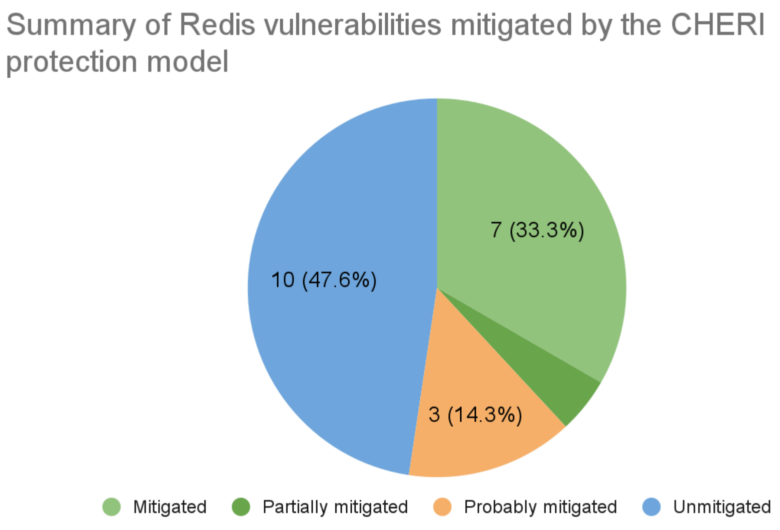
denial of service attack this is shown as a partial mitigation as significant questions remain as to restarting a compartment without loss of data on failure. Furthermore, without supporting mitigations such as blocking firewall an attacker can simply repeatedly retrigger a given vulnerability resulting in denial of service.



Redis

The figure below shows a breakdown of Redis vulnerabilities and their mitigation by spatial/temporal memory safety provided by the CHERI protection model. This shows that mitigation of security vulnerabilities in Redis with CHERI spatial/temporal memory protection is approximately in the range of 38% to 52% (note that CVE-2021-32762 was ignored in this analysis as it was mitigated by platform memory allocator and not specifically by the CHERI protection model). This mitigation rate seems broadly consistent with previously reported results described above.

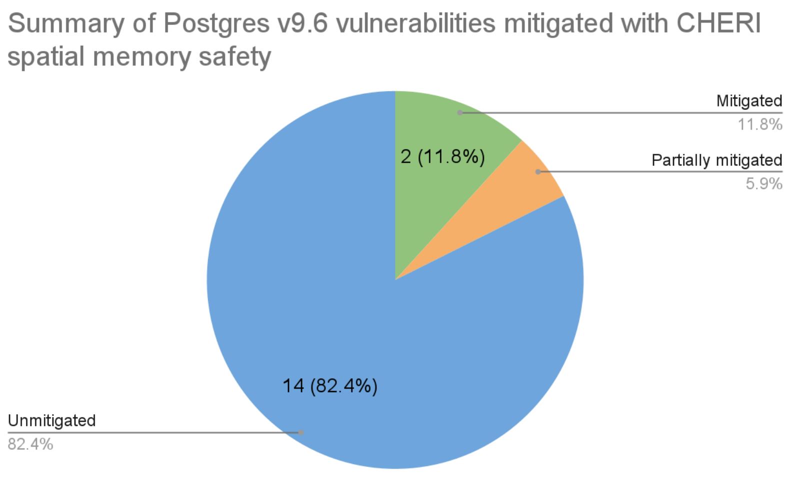
Exploration of compartmentalisation opportunities in Redis has not been systematically explored within the current project. Though it should be noted that as Redis is a relatively simple key/value store, compartmentalisation may be easier to apply than in the case of Postgres discussed below.



Postgres

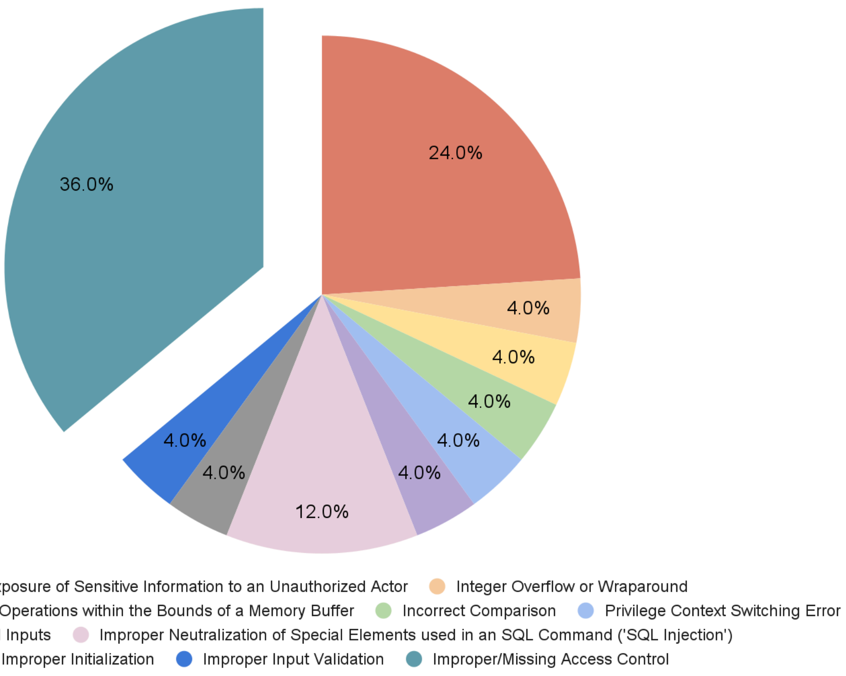
The security evaluation of Postgres has been performed with version 9.6 using the official Postgres security advisories for that version. Postgres version 9.6 was chosen as the target for the security evaluation as this evaluation was performed early in the project prior to the completion of porting work. At that time only the legacy port of Postgres v9.6 by Alex Richardson at the University of Cambridge was available and it was unclear as to whether the attempt to port a latter version (15 beta 4) would be successful.

The figure below shows that mitigation of reported security vulnerabilities in Postgres with CHERI spatial memory protection is significantly lower than for nginx, Redis, and in previously reported studies, with approximately between 12% and 18% of vulnerabilities mitigated. This result is somewhat surprising as Postgres fits the established profile of software likely to contain significant numbers of memory safety issues,its large and complex, with lots of low level manipulation of memory.



Looking at the assigned CWEs approximately 36% of the Postgres security advisories related to private data disclosure or remote code execution, that result from failures of the authentication and authorisation controls. A further 16% of security advisories relate to improper input validation/SQL injection attacks. Therefore, in total, 52% of the security advisories relate to attacks that CHERI-protections are not designed to mitigate. If we consider only the remaining 48% of Postgres security advisories, the potential mitigation rate with CHERI memory protection is between 25% and 37%. These figures are more consistent with previously reported values though remain towards the low side.

Web servers such as nginx also have an explicit access control model preventing arbitrary access of the underlying filesystem and isolating between virtual server instances (nginx server blocks). However, whilst there are a number of nginx security advisories related to the access control model - such as CVE-2010-2266 and CVE-2009-3898 - they are significantly less common than in Postgres. It is possible that this is because the access control model in nginx is simpler or more mature than in Postgres. Repeating the security evaluation for an up to date version of Postgres (for example, 15 beta 4 ported in the project) would perhaps give some insights as to how vulnerabilities evolve over time and the role the CHERI protection model would have played in mitigating those issues. Section 7 discusses the value of conducting a longitudinal study of the CHERI protection model.



Compartmentalisation enforces privilege separation such that code only accesses data to which it is explicitly granted access. In a database such as Postgres, it remains unclear where privilege separation can be usefully applied; as a database query can access all data across the set of database tables. However, library compartmentalisation could be applied to compartmentalise common operations such as, for example, hashing or string manipulation, and can be trivially applied to `libpq` within the client applications. Evaluation of compartmentalisation opportunities and benefits in Postgres has not been systematically explored in the project.

7. Future research and roadmap

This project is structured as a pilot research study similar to our previous study of desktop software stacks. Building on this pilot, a future more comprehensive project would seek to greatly expand the scope of server software stack including classes of software not included in the pilot study. This includes increasing the choice of key components such as databases, adding support for commonly used runtime environments such as NodeJS, including support for languages such as Java and Python, and adding popular web frameworks such as Spring Boot and Flask. In addition, the scope of changes to the server software would be broadened to include enforcement of bounds in allocations by custom memory allocations, enforcement of sub-object bounds within data structures, and changes to support new CheriBSD temporal memory safety features.

Within this study we have only considered the library compartmentalisation model. Coprocess isolation is an alternative, emerging compartmentalisation model built on the CHERI protection model. A future research project would explore the application of the coprocess compartmentalisation model in server software. Furthermore, we would like to revisit the goal of providing per-request isolation, which may require more significant changes to software prototypes and possibly the development of new data focussed compartmentalisation models.

This project is one of the real-world applications of the library compartmentalisation, and it is believed to be the first to explore compartmentalisation of C++ code. Library compartmentalisation of C++ code introduces some significant research challenges. For example, the interactions between the usage of C++ types - such as, lambda types - across compartmentalisation boundaries has subtle properties that directly impact security. Furthermore, these behaviours may not align well with user expectations resulting in unintentional misuse that may introduce security vulnerabilities. Further research is required in compartmentalisation of C++ code. As in this project, a codesign with a real-world use case is likely to produce better and more mature outcomes than performing this work in isolation.

The performance optimisations added to the library compartmentalisation model rely on specifying policy to elide compartment transitions where there is minimal or no additional risk to security. It remains an open research topic to determine where such policies originate. User specified policies bring flexibility and can adapt to the specific needs of the application, as in the case of our gRPC demonstrator where the Abseil libraries are composed into a single compartment. Policy that originated from the user is inherently trusted but may not be trustworthy. Common policy can be specified by the system, as in the case of the default policy used in the nginx performance evaluation. However, if policy comes from the system how is trust established? Trust in policy is different from placing trust in a large, complex library processing user data, but this still raises questions about the increased scope of the trusted computing base.

Compartmentalisation enforces privilege separation such that code only accesses data to which it is explicitly granted access. In some cases it is clear, to someone with intimate knowledge of the system, where opportunities for privilege separation are present. At other times it may be less clear. Tooling that supports both identifying compartmentalisation opportunities and enforcing CHERI compartmentalisation, is an extremely important but currently underexplored area that we would seek to address in a future research project.

Whereas it is relatively straightforward to provide the mechanics for integrity and confidentiality guarantees around shared libraries, it is less clear how to address availability in the presence of a library crash or hang: historically, it has been assumed that a library failure has the scope of the process. As a result the library compartmentalisation model cannot - currently - provide any mitigation (even a partial mitigation) for denial of service attacks. Providing availability guarantees to software using compartmentalisation exposes the application to the fact that it operates as part of a distributed system, for example introducing a requirement to provide idempotency of requests to simplify retrying and improve resilience to failures. Therefore, providing availability guarantees likely requires significant rework of the application. Whilst restarting a stateless compartment may be fairly straightforward, stateful processing requires the restoration of state on failure. Future research on availability may consider the role for new operating abstractions such as co-process isolation to manage compartments with the goal of providing practical availability guarantees.

Whilst a number of paper-based studies have evaluated both aspects of the perceived ease of use and utility of the CHERI protection model (both for spatial and temporal memory safety, and with hypothetical compartmentalisation) these studies are limited in scope and were performed independently of each other presenting questions about the comparability of results. For example, in our previous study on desktop software stacks the compartmentalisation model is assumed to provide mitigation against denial of service attacks. Whereas in this study we assume only a partial mitigation under some future implementation of the compartmentalisation model. Such differences make comparison of the overall potential mitigation rates problematic. A high-quality, longitudinal study conducted using a single consistent methodology, across multiple technology areas is required to provide strong evidence for both perceived utility and ease of use of the CHERI protection model.

8. Conclusions

This project has ported a total of 1.7 million lines of server-side software code to memory safe CHERI C/C++. The porting effort has been for the most part straightforward, affecting approximately 1% of the total lines of code. The scope and size of the code changes is consistent with previous reports on CHERI C/C++ software porting efforts. Additionally, we provide a number of case-studies that show interesting portability challenges we faced during this project.

Our performance measurement and analysis has been constrained by practical limitations of the Morello prototype, which has not undergone significant optimisation of its hardware design. It is important to interpret our performance results in the context of Arm and the University of Cambridge's performance report, which notes the potential for an 80%-88% reduction in overhead from measured results on the Morello board, even using the recently released Benchmark ABI, when further optimization is applied to the hardware implementation as expected of a production design.

Our memory safe experimental port of nginx exhibits a 2% reduction in requests/sec compared to the baseline hybrid version (running without memory safety), running on the Morello board and compiled with the Benchmark ABI. The initially unoptimised library compartmentalisation significantly impacted nginx's performance, prompting a co-design optimisation activity that contributed to the development of compartmentalisation policies. The default compartmentalisation policy achieves about a 2.5% overhead with respect to the memory safe prototype.

The gRPC performance overhead is more significant. The QPS throughput benchmark exhibits a 16% reduction in messages/sec for the insecure and SSL workloads, when compared to the baseline hybrid version. Library compartmentalisation introduces an additional 12% to 14% reduction in messages/sec, which can be reduced down to about 5% to 9% respectively for the insecure and SSL workloads, with the use of a custom library compartmentalisation policy. The performance results show that our prototype was able to scale to many millions of domain transitions per second with relatively low overheads on top of CHERI memory safety.

Although these performance impacts are within previously observed worst-case values, they are significant for high-performance software components and therefore warrant further investigation to determine the root cause and possible optimizations. In particular, we hope that versions of these workloads may be run on Arm's modified FPGA implementations of Morello in the future, allowing a more accurate estimate of potential CHERI performance on future, more mature microarchitectures.

We have reviewed past vulnerabilities for the majority of the components of our software stacks. We estimate the impact of CHERI memory safety on a relevant subset of past vulnerabilities. In particular, we attempt to limit the analysis to those vulnerabilities that are part of the threat model for the software component under consideration. Unsurprisingly, the share of vulnerabilities mitigated with CHERI memory safety depends on the share of memory safety vulnerabilities for each software project. nginx exhibits a mitigation rate of around 46% and Redis between 38% to 52%. Both are in line with previous analysis of past vulnerabilities with respect to CHERI memory safety. Conversely, only between 12% and 18% of Postgres are considered to be mitigated. This can be attributed to the prevalence of access-control vulnerabilities in this software's vulnerability history, which are not addressed by CHERI. It is unclear how to interpret the dominance of non-memory-safety bugs in Postgres, which is atypical of open-source C/C++ software, and further analysis is required.

We also evaluate the effects of library compartmentalisation on past vulnerabilities. To this end, we make the assumption that it will be possible to provide a recovery mechanism from crashes at library granularity. However, for anything other than stateless libraries cleanly restarting a library on failure presents significant challenges (and may either result in loss of data or require significant changes to the application to prevent data loss). Therefore, we classify this a partial mitigation. Under these assumptions, we observe that a further 15% of nginx vulnerabilities may be mitigated giving an overall mitigation rate of 61%.

As a result of this project, the maturity of the library compartmentalisation model implementation in CheriBSD has significantly increased. We provide the first real-world C and C++ use cases of library compartmentalisation which motivated improvements in domain transition observability via traces, as well as performance improvements in the form of compartmentalisation policies. Finally, a number of bug fixes were made to the library compartmentalisation implementation as a result of this project.

Acknowledgments

This work was supported by Defence Science and Technology Laboratory (DSTL) through the Defence and Security Accelerator (DASA) funded project ACC6036483.

We thank Dapeng Gao at the University of Cambridge for his timely support in addressing bug fixes, improvements to the domain transition trampolines, and adding new observability and security policy features to the library compartmentalisation prototype to support our work.

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.

Appendix A: Performance Evaluation Raw Data

nginx

The table below shows measured requests/sec and transfer MB/sec measured using the wrk benchmarking tool ran in the following configuration:

```
Unset
./wrk -t 1 -c 50 -d 1m --latency https://192.168.2.2/rps/1kb-random.bin
```

nginx software configuration	Requests/sec	Transfer MB/sec	Performance overhead relative to Hybrid (baseline)
No-memory safety	33916	43.34	-
Memory safety	33450	42.74	1.37
Memory safety plus library compartmentalisation - default policy	32606	41.67	3.86
Memory safety plus lowerbound IPC overhead	20237	25.86	40.33

gRPC

The table below shows measured message/sec using the QPS benchmark tools that are part of the gRPC test suite. The results here use the following unmodified scenario files:

- grpc_cpp_protobuf_async_streaming_qps_unconstrained_insecure_8b.json
- grpc_cpp_protobuf_async_streaming_qps_unconstrained_secure_8b.json

Both scenario files are generated using the following command:

```
Unset
./tools/run_tests/performance/scenario_config_exporter.py -l c++ -r '.*' -f
grpc_ --export_scenarios --category all
```

gRPC software configuration	**Throughput (QPS)**	**SSL Throughput (QPS)**	**Throughput % overhead relative no memory safety**	**SSL Throughput % overhead relative to no memory safety**
No memory safety	53827	37276	-	-
Spatial memory safety	45342	31175	15.76	16.37
Spatial memory safety + default c18n policy	39851	26597	25.96	28.65
Spatial memory safety + custom c18n policy	42984	28494	20.14	23.56
Spatial memory safety + lowerbound IPC overhead	15567	8396	71.08	77.48

Appendix B: Security Evaluation Raw Data

nginx

Primary source of security advisories: https://nginx.org/en/security_advisories.html

CVE	Date	Severity	Description	Threat model	CWE	Assessment
CVE-2022-41741 CVE-2022-41742	2022-10-19	Medium	NGINX Open Source before versions 1.23.2 and 1.22.1, NGINX Open Source Subscription before versions R2 P1 and R1 P1, and NGINX Plus before versions R27 P1 and R26 P1 have a vulnerability in the module ngx_http_mp4_module that might allow a local attacker to corrupt NGINX worker memory, resulting in its termination or potential other impact using a specially crafted audio or video file. The issue affects only NGINX products that are built with the ngx_http_mp4_module, when the mp4 directive is used in the configuration file. Further, the attack is possible only if an attacker can trigger processing of a specially crafted audio or video file with	Denial-of-Service Private data disclosure	CWE-787 (Out-of-bounds Write)	Partially mitigated. Out-of-bounds write coerced into a deterministic crash. This vulnerability was caused by a failure to enforce a constraint on mp4 atoms (a logical error). A specially crafted mp4 file can then result in an out-of-bounds write corrupting nginx worker memory, leading to a crash or other issues. Though the issue has the potential to result in data disclosure or remote code execution analysis to

			the module ngx_http_mp4_module.			confirm is beyond the current project scope. Compartmentalisation of the nginx mp4 module partially mitigates the denial of service.
CVE-2021-23017	2021-05-25	Medium	A security issue in nginx resolver was identified, which might allow an attacker who is able to forge UDP packets from the DNS server to cause 1-byte memory overwrite, resulting in worker process crash or potential other impact.	Denial-of-Service Private data disclosure	CWE-193 (Off-by-one Error)	Partially mitigated. 1-byte overwrite coerced into a deterministic crash. Although still resulting in denial of service, this prevents other potential impacts. Though the issue has the potential to result in data disclosure or remote code execution analysis to confirm is beyond the current project scope. Issue patches ngx_resolver.c in src/core not a module, so no mitigation by compartmentalisation without changes.

CVE-2019-9511	2019-08-13	Medium	Some HTTP/2 implementations are vulnerable to window size manipulation and stream prioritisation manipulation, potentially leading to a denial of service. The attacker requests a large amount of data from a specified resource over multiple streams. They manipulate window size and stream priority to force the server to queue the data in 1-byte chunks. Depending on how efficiently this data is queued, this can consume excess CPU, memory, or both.	Denial-of-service	CWE-770 (Allocation of Resources Without Limits or Throttling) CWE-400 (Uncontrolled Resource Consumption)	Unmitigated
CVE-2019-9513	2019-08-13	Low	Some HTTP/2 implementations are vulnerable to resource loops, potentially leading to a denial of service. The attacker creates multiple request streams and continually shuffles the priority of the streams in a way that causes substantial churn to the priority tree. This can consume excess CPU.	Denial-of-service	CWE-400 (Uncontrolled Resource Consumption)	Unmitigated
CVE-2019-9516	2019-08-13	Low	Some HTTP/2 implementations are vulnerable to a header leak,	Denial-of-service	CWE-770 (Allocation of Resources	Unmitigated

			potentially leading to a denial of service. The attacker sends a stream of headers with a 0-length header name and 0-length header value, optionally Huffman encoded into 1-byte or greater headers. Some implementations allocate memory for these headers and keep the allocation alive until the session dies. This can consume excess memory.		Without Limits or Throttling) CWE-400 (Uncontrolled Resource Consumption)	
CVE-2018-16843	2018-11-06	Low	nginx before versions 1.15.6 and 1.14.1 has a vulnerability in the implementation of HTTP/2 that can allow for excessive memory consumption. This issue affects nginx compiled with the ngx_http_v2_module (not compiled by default) if the 'http2' option of the 'listen' directive is used in a configuration file.	Denial-of-service	CWE-400 (Uncontrolled Resource Consumption)	Unmitigated
CVE-2018-16844	2018-11-06	Low	nginx before versions 1.15.6 and 1.14.1 has a vulnerability in the implementation of HTTP/2 that can allow for excessive CPU usage. This issue affects nginx compiled with the ngx_http_v2_module	Denial-of-service	CWE-400 (Uncontrolled Resource Consumption)	Unmitigated

			(not compiled by default) if the 'http2' option of the 'listen' directive is used in a configuration file.			
CVE-2018-16845	2018-11-06	Medium	nginx before versions 1.15.6, 1.14.1 has a vulnerability in the ngx_http_mp4_module, which might allow an attacker to cause infinite loop in a worker process, cause a worker process crash, or might result in worker process memory disclosure by using a specially crafted mp4 file. The issue only affects nginx if it is built with the ngx_http_mp4_module (the module is not built by default) and the .mp4. directive is used in the configuration file. Further, the attack is only possible if an attacker is able to trigger processing of a specially crafted mp4 file with the ngx_http_mp4_module.	Denial-of-service	CWE-835 (Loop with Unreachable Exit Condition ('Infinite Loop')) CWE-400 (Uncontrolled Resource Consumption)	Partially mitigated. Out-of-bounds read coerced into a deterministic crash. Although still resulting in denial of service, this prevents other potential impacts. This vulnerability was caused by a failure to enforce constraint on mp4 atom sizes (>= size of the atom header). Compartmentalisation of the nginx mp4 module partially mitigates the denial of service.
CVE-2017-7529	2017-07-11	Medium	Nginx versions since 0.5.6 up to and including 1.13.2 are vulnerable to integer overflow vulnerability in nginx range filter module resulting into leak of potentially sensitive	Private data disclosure	CWE-190 (Integer Overflow or Wraparound)	Mitigated An HTTP Range request asks the server to send only a portion of an HTTP

			information triggered by specially crafted request.			message back to a client. The request for data outside the valid range produces an over read, coerced by CHERI protections to a deterministic crash.
CVE-2016-4450	2016-05-31	Medium	os/unix/ngx_files.c in nginx before 1.10.1 and 1.11.x before 1.11.1 allows remote attackers to cause a denial of service (NULL pointer dereference and worker process crash) via a crafted request, involving writing a client request body to a temporary file.	Denial-of-service	CWE-476 (NULL Pointer Dereference)	Unmitigated Issue not in a dynamically loaded module, so unmitigated by compartmentalisation without reengineering.
CVE-2016-0742	2016-01-26	Medium	The resolver in nginx before 1.8.1 and 1.9.x before 1.9.10 allows remote attackers to cause a denial of service (invalid pointer dereference and worker process crash) via a crafted UDP DNS response.	Denial-of-service	CWE-476 (NULL Pointer Dereference)	Unmitigated Issue not in a dynamically loaded module, so unmitigated by compartmentalisation without reengineering.
CVE-2016-0746	2016-01-26	Medium	Use-after-free vulnerability in the resolver in nginx 0.6.18 through 1.8.0 and 1.9.x before 1.9.10 allows remote attackers to cause a denial of service (worker process crash) or	Denial-of-service	CWE-416 (Use After Free)	Partially mitigated UAF vulnerability would be mitigated with CHERI-backed temporary memory

			possibly have unspecified other impact via a crafted DNS response related to CNAME response processing.			safety protections. Although still resulting in denial of service, this prevents other potential impacts. Issue not in a dynamically loaded module, so unmitigated by compartmentalisation without reengineering.
CVE-2016-0747	2016-01-26	Medium	The resolver in nginx before 1.8.1 and 1.9.x before 1.9.10 does not properly limit CNAME resolution, which allows remote attackers to cause a denial of service (worker process resource consumption) via vectors related to arbitrary name resolution.	Denial-of-service	CWE-400 (Uncontrolled Resource Consumption)	Unmitigated
CVE-2014-3616	2014-09-16	Medium	nginx 0.5.6 through 1.7.4, when using the same shared ssl_session_cache or ssl_session_ticket_key for multiple servers, can reuse a cached SSL session for an unrelated context, which allows remote attackers with certain privileges to conduct "virtual host confusion"	Private data disclosure	CWE-613 (Insufficient Session Expiration)	Unmitigated

			attacks.			
CVE-2014-3556	2014-08-05	Medium	The STARTTLS implementation in mail/ngx_mail_smtp_handler.c in the SMTP proxy in nginx 1.5.x and 1.6.x before 1.6.1 and 1.7.x before 1.7.4 does not properly restrict I/O buffering, which allows man-in-the-middle attackers to insert commands into encrypted SMTP sessions by sending a cleartext command that is processed after TLS is in place, related to a "plaintext command injection" attack, a similar issue to CVE-2011-0411	Remote code execution	CWE-77 (Improper Neutralization of Special Elements used in a Command ('Command Injection'))	Unmitigated
CVE-2014-0133	2014-03-18	Major	Heap-based buffer overflow in the SPDY implementation in nginx 1.3.15 before 1.4.7 and 1.5.x before 1.5.12 allows remote attackers to execute arbitrary code via a crafted request.	Remote code execution	CWE-787 (Out-of-bounds Write)	Mitigated
CVE-2014-0088	2014-03-04	Major	The SPDY implementation in the ngx_http_spdy_module module in nginx 1.5.10 before 1.5.11, when running on a 32-bit platform, allows remote attackers to execute arbitrary code via a crafted request.	Remote code execution	CWE-119 (Improper Restriction of Operations within the Bounds of a Memory	Mitigated

					Buffer)	
CVE-2013-4547	2013-11-19	Medium	nginx 0.8.41 through 1.4.3 and 1.5.x before 1.5.7 allows remote attackers to bypass intended restrictions via an unescaped space character in a URI.	Private data disclosure	CWE-116 (Improper Encoding or Escaping of Output)	Unmitigated
CVE-2013-2070	2013-05-13	Medium	http/modules/ngx_http_proxy_module.c in nginx 1.1.4 through 1.2.8 and 1.3.0 through 1.4.0, when proxy_pass is used with untrusted HTTP servers, allows remote attackers to cause a denial of service (crash) and obtain sensitive information from worker process memory via a crafted proxy response, a similar vulnerability to CVE-2013-2028.	Denial of service Private data disclosure	NVD-CWE-noinfo (Insufficient Information)	Partially mitigated Out-of-bounds read coerced into a deterministic crash. Although still resulting in denial of service, this prevents other potential impacts. Compartmentalisation of the nginx ngx_http_proxy module partially mitigates the denial of service.
CVE-2013-2028	2013-05-07	Major	The ngx_http_parse_chunked function in http/ngx_http_parse.c in nginx 1.3.9 through 1.4.0 allows remote attackers to cause a denial of service (crash) and execute arbitrary code via a chunked Transfer-Encoding request with a large chunk	Denial of service Remote code execution	CWE-787 (Out-of-bounds Write)	Partially mitigated Stack based buffer overflow coerced into deterministic crash. Although still resulting in denial of service, this prevents remote code execution.

			size, which triggers an integer signedness error and a stack-based buffer overflow.			Compartmentalisation of the nginx http module partially mitigates the denial of service.
CVE-2012-2089	2012-04-12	Major	Buffer overflow in ngx_http_mp4_module.c in the ngx_http_mp4_module module in nginx 1.0.7 through 1.0.14 and 1.1.3 through 1.1.18, when the mp4 directive is used, allows remote attackers to cause a denial of service (memory overwrite) or possibly execute arbitrary code via a crafted MP4 file.	Denial of service Remote code execution	CWE-120 (Buffer Copy without Checking Size of Input ('Classic Buffer Overflow'))	Partially mitigated Buffer overflow coerced into a deterministic crash. Although still resulting in denial of service, this prevents other potential impacts. Compartmentalisation of the nginx mp4 module partially mitigates the denial of service.
CVE-2012-1180	2012-03-15	Major	Use-after-free vulnerability in nginx before 1.0.14 and 1.1.x before 1.1.17 allows remote HTTP servers to obtain sensitive information from process memory via a crafted backend response, in conjunction with a client request.	Private data disclosure	CWE-416 (Use After Free)	Mitigated UAF vulnerability would be mitigated with CHERI-backed temporary memory safety protections.
CVE-2011-4315	2011	Medium	Heap-based buffer overflow in compression-pointer	Denial of service	CWE-787 (Out-of-bounds	Partially Mitigated

			processing in core/ngx_resolver.c in nginx before 1.0.10 allows remote resolvers to cause a denial of service (daemon crash) or possibly have unspecified other impact via a long response.		Write)	Buffer overflow coerced into a deterministic crash. Although still resulting in denial of service, this prevents other potential impacts. Issue not in a dynamically loaded module, so unmitigated by compartmentalisation without reengineering.
CVE-2010-2266	2010	Major	nginx 0.8.36 allows remote attackers to cause a denial of service (crash) via certain encoded directory traversal sequences that trigger memory corruption, as demonstrated using the "%c0.%c0." sequence.	Denial-of-service	CWE-22 (Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal'))	Unmitigated Issue not in a dynamically loaded module, so unmitigated by compartmentalisation without reengineering.
CVE-2009-4487	2009	None	nginx 0.7.64 writes data to a log file without sanitizing non-printable characters, which might allow remote attackers to modify a window's title, or possibly execute arbitrary commands or overwrite files, via an HTTP request containing an escape sequence for a terminal	Private data disclosure	NVD-CWE-noinfo (Insufficient Information)	Unmitigated Issue not in a dynamically loaded module, so unmitigated by compartmentalisation without reengineering.

			emulator.			
CVE-2009-3898	2009	Minor	Directory traversal vulnerability in src/http/modules/ngx_http_dav_module.c in nginx (aka Engine X) before 0.7.63, and 0.8.x before 0.8.17, allows remote authenticated users to create or overwrite arbitrary files via a .. (dot dot) in the Destination HTTP header for the WebDAV (1) COPY or (2) MOVE method.	Private data disclosure	CWE-22 (Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal'))	Unmitigated
CVE-2009-2629	2009	Major	Buffer underflow in src/http/ngx_http_parse.c in nginx 0.1.0 through 0.5.37, 0.6.x before 0.6.39, 0.7.x before 0.7.62, and 0.8.x before 0.8.15. Exploitation of this vulnerability would cause the nginx server to write data contained in the URI to heap memory before the allocated buffer.	Private data disclosure	CWE-787 (Out-of-bounds Write)	Mitigated Out-of-bounds write underflowing buffer coerced into a deterministic crash. Compartmentalisation of the nginx http module partially mitigates the denial of service.
CVE-2009-3896	2009	Major	src/http/ngx_http_parse.c in nginx (aka Engine X) 0.1.0 through 0.4.14, 0.5.x before 0.5.38, 0.6.x before 0.6.39, 0.7.x before 0.7.62, and 0.8.x before 0.8.14 allows remote attackers to cause a denial of	Denial-of-service	CWE-119 (Improper Restriction of Operations within the Bounds of a Memory	Unmitigated Compartmentalisation of the nginx mp4 module partially mitigates the denial of

			service (NULL pointer dereference and worker process crash) via a long URI.		Buffer)	service.

Redis

Primary source of security advisories: https://github.com/redis/redis/security/advisories

CVE	Date	Severity	Description	Threat model	CWE	Assessment
CVE-2023-28856	2023-04-18	Moderate	Authenticated users can use the `HINCRBYFLOAT` command to create an invalid hash field that may later crash Redis on access.	Denial of service	CWE-617 (Reachable Assertion) CWE-20 (Improper Input Validation)	Unmitigated
CVE-2023-28425	2023-03-20	Moderate	Authenticated users can use the `MSETNX` command to trigger a runtime assertion and termination of the Redis server process.	Denial of service	CWE-617 (Reachable Assertion) CWE-77 (Improper Neutralization of Special Elements used in a Command ('Command Injection'))	Unmitigated
CVE-2023-25155	2023-02-28	Moderate	Authenticated users issuing specially crafted	Denial of service	CWE-190 (Integer Overflow	Unmitigated

			`SRANDMEMBER`, `ZRANDMEMBER`, and `HRANDFIELD` commands can trigger an integer overflow, resulting in a runtime assertion and termination of the Redis server process.		or Wraparound)	
CVE-2023-22458	2023-01-20	Moderate	Authenticated users can issue a `HRANDFIELD` or `ZRANDMEMBER` command with specially crafted arguments to trigger a denial-of-service by crashing Redis with an assertion.	Denial of service	CWE-190 (Integer Overflow or Wraparound)	Unmitigated
CVE-2022-36021	2023-02-28	Moderate	Authenticated users can use string matching commands (like `SCAN` or `KEYS`) with a specially crafted pattern to trigger a denial-of-service attack on Redis, causing it to hang and consume 100% CPU time.	Denial of service	CWE-407 (Inefficient Algorithmic Complexity)	Unmitigated
CVE-2022-35977	2023-01-20	Medium	Authenticated users issuing specially crafted `SETRANGE` and `SORT(_RO)` commands can trigger an integer overflow, resulting with Redis attempting to allocate impossible amounts of memory and abort with an OOM panic.	Denial of service	CWE-190 (Integer Overflow or Wraparound)	Unmitigated

CVE-2022-35951	2022-09-26	High	Executing a XAUTOCLAIM command on a stream key in a specific state, with a specially crafted COUNT argument may cause an integer overflow, a subsequent heap overflow, and potentially lead to remote code execution. The problem affects Redis versions 7.0.0 or newer.	Remote code execution	CWE-190 (Integer Overflow or Wraparound)	Mitigated Although the root cause is an integer overflow, this is subsequently used to perform a heap overflow from which RCE may be obtained. The heap overflow is coerced into a deterministic crash by CHERI protections. Further, detailed analysis is required to assess the impact of the integer overflow in isolation.
CVE-2022-31144	2022-06-22	High	A specially crafted XAUTOCLAIM command on a stream key in a specific state may result with heap overflow, and potentially remote code execution. The problem affects Redis versions 7.0.0 or newer.	Remote code execution	CWE-787 (Out-of-bounds Write) CWE-122 (Heap-based Buffer Overflow)	Probably Mitigated Within the scope of the project it was not possible to identify the exact commit related to this issue. Whilst it is likely that CHERI spatial memory protections mitigate the issue further investigation is needed to confirm this.

CVE-2022-24735	2022-04-27	Low	By exploiting weaknesses in the Lua script execution environment, an attacker with access to Redis can inject Lua code that will execute with the (potentially higher) privileges of another Redis user. The Lua script execution environment in Redis provides some measures that prevent a script from creating side effects that persist and can affect the execution of the same, or different script, at a later time. Several weaknesses of these measures have been publicly known for a long time, but they had no security impact as the Redis security model did not endorse the concept of users or privileges. With the introduction of ACLs in Redis 6.0, these weaknesses can be exploited by a less privileged users to inject Lua code that will execute at a later time, when a privileged user executes a Lua script.	Remote code execution	CWE-94 (Improper Control of Generation of Code ('Code Injection'))	Unmitigated
CVE-2022-24736	2022-04-27	Low	An attacker attempting to load	Denial of	CWE-476 (NULL	Unmitigated

			a specially crafted Lua script can cause NULL pointer dereference which will result with a crash of the redis-server process. This issue affects all versions of Redis.	service	Pointer Dereference)	
CVE-2021-41099	2021-10-04	High	An integer overflow bug in the underlying string library can be used to corrupt the heap and potentially result with denial of service or remote code execution. The vulnerability involves changing the default `proto-max-bulk-len` configuration parameter to a very large value and constructing specially crafted network payloads or commands.	Denial of service	CWE-190 (Integer Overflow or Wraparound) CWE-680 (Integer Overflow to Buffer Overflow)	Partially mitigated Allocation of `hdrlen+newlen+1` results in integer overflow of size_t type. The following `memcpy` into the buffer is coerced into a deterministic crash by CHERI's spatial memory protections.
CVE-2021-32761	2021-07-21	High	On 32-bit versions, Redis BITFIELD command is vulnerable to integer overflow that can potentially be exploited to corrupt the heap, leak arbitrary heap contents or trigger remote code execution. The vulnerability involves constructing specially crafted bit commands which overflow the	Remote code execution Private data disclosure	CWE-190 (Integer Overflow or Wraparound) CWE-680 (Integer Overflow to Buffer Overflow)	Probably mitigated Further investigation required. The use of an integer overflow to corrupt the heap is likely to be mitigated, but without identifying the scope of the commit fixing the

			bit offset. This problem only affects 32-bit versions of Redis.			issue this is hard to confirm.
CVE-2021-32687	2021-10-04	High	Redis is an open source, in-memory database that persists on disk. An integer overflow bug affecting all versions of Redis can be exploited to corrupt the heap and potentially be used to leak arbitrary contents of the heap or trigger remote code execution. The vulnerability involves changing the default set-max-intset-entries configuration parameter to a very large value and constructing specially crafted commands to manipulate sets. The problem is fixed in Redis versions 6.2.6, 6.0.16 and 5.0.14. An additional workaround to mitigate the problem without patching the redis-server executable is to prevent users from modifying the set-max-intset-entries configuration parameter. This can be done using ACL to restrict unprivileged users from using the CONFIG SET command.	Remote code execution Private data disclosure	CWE-190 (Integer Overflow or Wraparound) CWE-125 (Out-of-bounds Read) CWE-680 (Integer Overflow to Buffer Overflow)	Mitigated The operation `len*intrev32ifbe(is->encoding)` in `intsetResize` results in integer overflow of the `uint32_t` type, resulting in an incorrect buffer reallocation. Subsequent buffer overwrites are coerced to a deterministic crash by CHERI spatial memory

CVE-2021-32675	2021-10-04	High	Redis is an open source, in-memory database that persists on disk. When parsing an incoming Redis Standard Protocol (RESP) request, Redis allocates memory according to user-specified values which determine the number of elements (in the multi-bulk header) and size of each element (in the bulk header). An attacker delivering specially crafted requests over multiple connections can cause the server to allocate significant amount of memory. Because the same parsing mechanism is used to handle authentication requests, this vulnerability can also be exploited by unauthenticated users. The problem is fixed in Redis versions 6.2.6, 6.0.16 and 5.0.14. An additional workaround to mitigate this problem without patching the redis-server executable is to block access to prevent unauthenticated users from connecting to Redis. This can be done in different ways: Using network access control tools like firewalls, iptables, security groups, etc. or	Denial-of-servic e	CWE-770 (Allocation of Resources Without Limits or Throttling)	Unmitigated

			Enabling TLS and requiring users to authenticate using client side certificates.			
CVE-2021-32672	2021-10-04	Low	Redis is an open source, in-memory database that persists on disk. When using the Redis Lua Debugger, users can send malformed requests that cause the debugger's protocol parser to read data beyond the actual buffer. This issue affects all versions of Redis with Lua debugging support (3.2 or newer). The problem is fixed in versions 6.2.6, 6.0.16 and 5.0.14.	Private data disclosure	CWE-125 (Out-of-bounds Read)	Mitigated Lua debugger command parsing (`ldbReplParseCommand`) results in an over read. This is coerced to a deterministic crash by CHERI spatial memory protections.
CVE-2021-32628	2021-10-04	High	Redis is an open source, in-memory database that persists on disk. An integer overflow bug in the ziplist data structure used by all versions of Redis can be exploited to corrupt the heap and potentially result with remote code execution. The vulnerability involves modifying the default ziplist configuration parameters (hash-max-ziplist-entries, hash-max-ziplist-value, zset-max-ziplist-entries or zset-max-ziplist-value) to a	Remote code execution	CWE-190 (Integer Overflow or Wraparound) CWE-680 (Integer Overflow to Buffer Overflow)	Mitigated The addition of the computation of `totelelen` and the check `totelelen > STREAM_LISTPACK_MAX_SIZE` to `streamAppendItem` prevents the buffer overwrite. This overwrite would be coerced to a deterministic crash by CHERI spatial memory protections.

			very large value, and then constructing specially crafted commands to create very large ziplists. The problem is fixed in Redis versions 6.2.6, 6.0.16, 5.0.14. An additional workaround to mitigate the problem without patching the redis-server executable is to prevent users from modifying the above configuration parameters. This can be done using ACL to restrict unprivileged users from using the CONFIG SET command.			
CVE-2021-32627	2021-10-04	High	Redis is an open source, in-memory database that persists on disk. In affected versions an integer overflow bug in Redis can be exploited to corrupt the heap and potentially result with remote code execution. The vulnerability involves changing the default proto-max-bulk-len and client-query-buffer-limit configuration parameters to very large values and constructing specially crafted very large stream elements. The problem is fixed in Redis 6.2.6, 6.0.16 and 5.0.14. For users unable to upgrade an	Remote code execution	CWE-190 (Integer Overflow or Wraparound) CWE-680 (Integer Overflow to Buffer Overflow)	

			additional workaround to mitigate the problem without patching the redis-server executable is to prevent users from modifying the proto-max-bulk-len configuration parameter. This can be done using ACL to restrict unprivileged users from using the CONFIG SET command.			
CVE-2021-32626	2021-10-04	High	Redis is an open source, in-memory database that persists on disk. In affected versions specially crafted Lua scripts executing in Redis can cause the heap-based Lua stack to be overflowed, due to incomplete checks for this condition. This can result with heap corruption and potentially remote code execution. This problem exists in all versions of Redis with Lua scripting support, starting from 2.6. The problem is fixed in versions 6.2.6, 6.0.16 and 5.0.14. For users unable to update an additional workaround to mitigate the problem without patching the redis-server executable is to prevent users from executing Lua scripts.	Remote code execution	CWE-787 (Out-of-bounds Write) CWE-122 (Heap-based Buffer Overflow)	Unmitigated

			This can be done using ACL to restrict EVAL and EVALSHA commands.			
CVE-2021-29478	2021-05-03	High	Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache, and message broker. An integer overflow bug in Redis 6.2 before 6.2.3 could be exploited to corrupt the heap and potentially result with remote code execution. Redis 6.0 and earlier are not directly affected by this issue. The problem is fixed in version 6.2.3. An additional workaround to mitigate the problem without patching the `redis-server` executable is to prevent users from modifying the `set-max-intset-entries` configuration parameter. This can be done using ACL to restrict unprivileged users from using the `CONFIG SET` command.	Remote code execution	CWE-190 (Integer Overflow or Wraparound)	Mitigated Further analysis identifying the scope of the commit fixing the issue is required to confirm the mitigation of resultant heap corruption.
CVE-2021-29477	2021-05-03	High	Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache, and message broker. An integer	Remote code execution	CWE-190 (Integer Overflow or Wraparound)	Probably Mitigated Further analysis identifying the scope of the commit fixing

			overflow bug in Redis version 6.0 or newer could be exploited using the `STRALGO LCS` command to corrupt the heap and potentially result with remote code execution. The problem is fixed in version 6.2.3 and 6.0.13. An additional workaround to mitigate the problem without patching the redis-server executable is to use ACL configuration to prevent clients from using the `STRALGO LCS` command.			the issue is required to confirm the mitigation of resultant heap corruption.
CVE-2021-21309	2021-02-23	High	Redis is an open-source, in-memory database that persists on disk. In affected versions of Redis an integer overflow bug in 32-bit Redis version 4.0 or newer could be exploited to corrupt the heap and potentially result with remote code execution. Redis 4.0 or newer uses a configurable limit for the maximum supported bulk input size. By default, it is 512MB which is a safe value for all platforms. If the limit is significantly increased, receiving a large request from a client may trigger several integer overflow scenarios,	Remote code execution	CWE-190 (Integer Overflow or Wraparound)	Mitigated The operation `hdrlen + newlen + 1` can result in an integer overleaf of the size_t type. This, for example in the function `sdsMakeRoomFor` results in the incorrect reallocation of the memory buffer. Concatenation into the new buffer using `sdscatlen` produces a buffer overwrite coerced into a deterministic crash

			which would result with buffer overflow and heap corruption. We believe this could in certain conditions be exploited for remote code execution. By default, authenticated Redis users have access to all configuration parameters and can therefore use the “CONFIG SET proto-max-bulk-len” to change the safe default, making the system vulnerable. **This problem only affects 32-bit Redis (on a 32-bit system, or as a 32-bit executable running on a 64-bit system).** The problem is fixed in version 6.2, and the fix is back ported to 6.0.11 and 5.0.11. Make sure you use one of these versions if you are running 32-bit Redis. An additional workaround to mitigate the problem without patching the redis-server executable is to prevent clients from directly executing `CONFIG SET`: Using Redis 6.0 or newer, ACL configuration can be used to block the command. Using older versions, the `rename-command` configuration directive can be			by CHERI protections.

			used to rename the command to a random string unknown to users, rendering it inaccessible. Please note that this workaround may have an additional impact on users or operational systems that expect `CONFIG SET` to behave in certain ways			

Postgres

Version: 9.6
Primary source of security advisories: https://www.postgresql.org/support/security/9.6/

CVE	Date	Severity	Description	Threat model	CWE	Assessment
CVE-2021-32028	2021-05-04	Medium	A flaw was found in postgresql. Using an INSERT ... ON CONFLICT ... DO UPDATE command on a purpose-crafted table, an authenticated database user could read arbitrary bytes of server memory. The highest threat from this vulnerability is to data confidentiality.	Private data disclosure	CWE 200 (Exposure of Sensitive Information to an Unauthorized Actor)	Mitigated Reading of arbitrary server memory should be mitigated but more detailed investigation is required to assess any residual risk.
CVE-2021-32027	2021-05-13	High	A flaw was found in postgresql in versions before 13.3, before 12.7, before 11.12, before 10.17 and before 9.6.22. While modifying certain SQL array	Private data disclosure Denial of service	CWE-190 (Integer Overflow or Wraparound)	Mitigated Integer overflow allows out-of-bounds memory write. Writing of

			values, missing bounds checks let authenticated database users write arbitrary bytes to a wide area of server memory. The highest threat from this vulnerability is to data confidentiality and integrity as well as system availability.		CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer)	arbitrary server memory should be mitigated but more detailed investigation is required to assess any residual risk.
CVE-2020-25696	2020-11-12	High	A flaw was found in the psql interactive terminal of PostgreSQL in versions before 13.1, before 12.5, before 11.10, before 10.15, before 9.6.20 and before 9.5.24. If an interactive psql session uses \gset when querying a compromised server, the attacker can execute arbitrary code as the operating system account running psql. The highest threat from this vulnerability is to data confidentiality and integrity as well as system availability	Remote code execution	CWE-697 (Incorrect Comparison) CWE-270 (Privilege Context Switching Error) CWE-183 (Permissive List of Allowed Inputs)	Unmitigated
CVE-2020-25695	2020-11-04	High	A flaw was found in PostgreSQL versions before 13.1, before 12.5, before 11.10, before 10.15, before 9.6.20 and before 9.5.24. An attacker having permission to create non-temporary objects in at least one schema can	Remote code execution	CWE-89 (Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'))	Unmitigated

			execute arbitrary SQL functions under the identity of a superuser. The highest threat from this vulnerability is to data confidentiality and integrity as well as system availability.			
CVE-2020-1720	2020-02-13	Medium	A flaw was found in PostgreSQL's "ALTER ... DEPENDS ON EXTENSION", where sub-commands did not perform authorization checks. An authenticated attacker could use this flaw in certain configurations to perform drop objects such as function, triggers, et al., leading to database corruption. This issue affects PostgreSQL versions before 12.2, before 11.7, before 10.12 and before 9.6.17.	Private data disclosure	CWE-862 (Missing Authorization) CWE-285 (Improper Authorization)	Unmitigated
CVE-2019-10208	2019-08-08	High	A flaw was discovered in postgresql versions 9.4.x before 9.4.24, 9.5.x before 9.5.19, 9.6.x before 9.6.15, 10.x before 10.10 and 11.x before 11.5 where arbitrary SQL statements can be executed given a suitable SECURITY DEFINER function. An attacker, with EXECUTE permission on the function, can	Remote code execution	CWE-89 (Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'))	Unmitigated

			execute arbitrary SQL as the owner of the function.			
CVE-2019-10130	2019-09-05	Medium	A vulnerability was found in PostgreSQL versions 11.x up to excluding 11.3, 10.x up to excluding 10.8, 9.6.x up to, excluding 9.6.13, 9.5.x up to, excluding 9.5.17. PostgreSQL maintains column statistics for tables. Certain statistics, such as histograms and lists of most common values, contain values taken from the column. PostgreSQL does not evaluate row security policies before consulting those statistics during query planning; an attacker can exploit this to read the most common values of certain columns. Affected columns are those for which the attacker has SELECT privilege and for which, in an ordinary query, row-level security prunes the set of rows visible to the attacker.	Private data disclosure	CWE-284 (Improper Access Control)	Unmitigated
CVE-2018-10925	2018-08-09	High	It was discovered that PostgreSQL versions before 10.5, 9.6.10, 9.5.14, 9.4.19, and 9.3.24 failed to properly check authorization on certain statements involved with	Private data disclosure	CWE-863 (Incorrect Authorization)	Unmitigated

			"INSERT ... ON CONFLICT DO UPDATE". An attacker with "CREATE TABLE" privileges could exploit this to read arbitrary bytes server memory. If the attacker also had certain "INSERT" and limited "UPDATE" privileges to a particular table, they could exploit this to update other columns in the same table.			
CVE-2018-10915	2018-08-09	High	A vulnerability was found in libpq, the default PostgreSQL client library where libpq failed to properly reset its internal state between connections. If an affected version of libpq was used with "host" or "hostaddr" connection parameters from untrusted input, attackers could bypass client-side connection security features, obtain access to higher privileged connections or potentially cause other impact through SQL injection, by causing the PQescape() functions to malfunction. Postgresql versions before 10.5, 9.6.10, 9.5.14, 9.4.19, and 9.3.24 are affected	Private data disclosure	CWE-89 (Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')) CWE-200 (Exposure of Sensitive Information to an Unauthorized Actor) CWE-665 (Improper Initialization)	Unmitigated
CVE-2018-1115	2018-05-07	Critical	Postgresql before versions	Private data	CWE-732	Unmitigated

			10.4, 9.6.9 is vulnerable in the adminpack extension, the pg_catalog.pg_logfile_rotate() function doesn't follow the same ACLs than pg_rorate_logfile. If the adminpack is added to a database, an attacker able to connect to it could exploit this to force log rotation.	disclosure	(Incorrect Permission Assignment for Critical Resource)	
CVE-2018-1058	2018-03-01	High	A flaw was found in the way Postgresql allowed a user to modify the behavior of a query for other users. An attacker with a user account could use this flaw to execute code with the permissions of superuser in the database. Versions 9.3 through 10 are affected.	Private data disclosure	CWE-20 (Improper Input Validation)	Unmitigated
CVE-2017-15099	2017-11-09	Medium	INSERT ... ON CONFLICT DO UPDATE commands in PostgreSQL 10.x before 10.1, 9.6.x before 9.6.6, and 9.5.x before 9.5.10 disclose table contents that the invoker lacks privilege to read. These exploits affect only tables where the attacker lacks full read access but has both INSERT and UPDATE privileges. Exploits bypass row level security policies and lack	Private data disclosure	CWE 200 (Exposure of Sensitive Information to an Unauthorized Actor)	Unmitigated

			of SELECT privilege.			
CVE-2017-15098	2017-11-09	High	Invalid json_populate_recordset or jsonb_populate_recordset function calls in PostgreSQL 10.x before 10.1, 9.6.x before 9.6.6, 9.5.x before 9.5.10, 9.4.x before 9.4.15, and 9.3.x before 9.3.20 can crash the server or disclose a few bytes of server memory.	Denial of service Private data disclosure	CWE 200 (Exposure of Sensitive Information to an Unauthorized Actor)	Partially Mitigated Disclosure of sever memory likely mitigated (coerced to a deterministic crash) but further investigation required to assess any residual risk.
CVE-2017-7547	2017-08-10	High	PostgreSQL versions before 9.2.22, 9.3.18, 9.4.13, 9.5.8 and 9.6.4 are vulnerable to authorization flaw allowing remote authenticated attackers to retrieve passwords from the user mappings defined by the foreign server owners without actually having the privileges to do so.	Private data disclosure	CWE-522 (Insufficiently Protected Credentials)	Unmitigated
CVE-2017-7546	2017-08-10	Critical	PostgreSQL versions before 9.2.22, 9.3.18, 9.4.13, 9.5.8 and 9.6.4 are vulnerable to incorrect authentication flaw allowing remote attackers to gain access to database accounts with an empty password.	Private data disclosure	CWE-287(Improper Authentication)	Unmitigated
CVE-2017-7486	2017-05-11	High	PostgreSQL versions 8.4 - 9.6 are vulnerable to information	Private data disclosure	CWE 200 (Exposure of	Unmitigated

			leak in pg_user_mappings view which discloses foreign server passwords to any user having USAGE privilege on the associated foreign server.		Sensitive Information to an Unauthorized Actor) CWE-522 (Insufficiently Protected Credentials)	
CVE-2017-7484	2017-05-11	High	It was found that some selectivity estimation functions in PostgreSQL before 9.2.21, 9.3.x before 9.3.17, 9.4.x before 9.4.12, 9.5.x before 9.5.7, and 9.6.x before 9.6.3 did not check user privileges before providing information from pg_statistic, possibly leaking information. An unprivileged attacker could use this flaw to steal some information from tables they are otherwise not allowed to access.	Private data disclosure	CWE 200 (Exposure of Sensitive Information to an Unauthorized Actor) CWE-285 (Improper Authorization)	Unmitigated