Capability Boehm: Challenges and Opportunities for Garbage Collection with Capability Hardware

Dejice Jacob
dejice.jacob@glasgow.ac.uk
University of Glasgow
Glasgow, UK

Jeremy Singer
jeremy.singer@glasgow.ac.uk
University of Glasgow
Glasgow, UK

Abstract

The Boehm-Demers-Weiser Garbage Collector (BDWGC) is a widely used, production-quality memory management framework for C and C++ applications. In this work, we describe our experiences in adapting BDWGC for modern capability hardware, in particular the CHERI system, which provides guarantees about memory safety due to runtime enforcement of fine-grained pointer bounds and permissions. Although many libraries and applications have been ported to CHERI already, to the best of our knowledge this is the first analysis of the complexities of transferring a garbage collector to CHERI. We describe various challenges presented by the CHERI micro-architectural constraints, along with some significant opportunities for runtime optimization. Since we do not yet have access to capability hardware, we present a limited study of software event counts on emulated microbenchmarks. This experience report should be helpful to other systems implementors as they attempt to support the ongoing CHERI initiative.

CCS Concepts: • **Software and its engineering** → *Runtime environments*; **Garbage collection**.

Keywords: CHERI, early experience, memory management

ACM Reference Format:
Dejice Jacob and Jeremy Singer. 2022. Capability Boehm: Challenges and Opportunities for Garbage Collection with Capability Hardware. In *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '22), March 1, 2022, Virtual, Switzerland*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3516807.3516823

1 Introduction

Automatic memory management is practically ubiquitous for high-level languages that execute on managed runtime systems like JVMs and JavaScript engines. Although manual memory management with `malloc` and `free` remains the default for software developed in C/C++, it is possible for such applications to use automatic memory management—for example with the Boehm-Demers-Weiser garbage collector (BDWGC) [5] which is a drop-in library replacement for the system memory allocator. Boehm's seminal work in this area was cited as one of the reasons for his recent (2020) SIGPLAN Programming Languages Achievement award [1].

Implicit memory management 'just happens' from an application perspective, with no need for software developers to concentrate on memory allocation if it is not core application functionality. Naturally, there are two underlying assumptions for such clients of a garbage collection (GC) runtime service:

1. that the memory management is *correct*, i.e. no program runtime errors are induced by the GC; and,
2. that the memory management is *performant*, i.e. the code runs sufficiently fast and the GC does not incur noticeable overhead.

Many end-user applications depend on BDWGC for effective automatic memory management; these include the Inkscape vector graphics editor, the Guile interpreter for Scheme and the W3m text-based web browser.

In this paper, we investigate the redeployment of BDWGC to CHERIBSD on RISC-V and Morello platforms [16] which provide micro-architectural support for hardware capabilities. CHERI-style capabilities are a new hardware innovation, supporting fine-grained dynamic memory protection through direct hardware checking and enforcement of explicit pointer bounds and permissions.

A single capability value specifies a base address, an offset and an upper bound address limit, along with access permissions for read, write and execute behaviour. Capabilities include far richer metadata than traditional raw pointer values, which simply specify a single address, admit pointer arithmetic, and have access permissions set at the coarse granularity of pages. Once a user process acquires a capability value from the OS, the CHERI monotonicity property [9] ensures the capability's bounds cannot be widened and its permissions cannot be upgraded.

The principal motivation of the CHERI framework is to eliminate whole classes of memory-based system exploits; examples include out-of-bounds writes, out-of-bounds reads, and use-after-free accesses. Respectively these are ranked first, third, and seventh in the the 2021 Common Weakness Enumeration (CWE) top 25 most dangerous software weaknesses [8]. Although these problematic memory accesses result in undefined behaviour according to the C language specification, nevertheless such operations are tolerated by many current compilers and runtimes, with the potential to lead to security compromises. Many catalogued common vulnerabilities and exposures (CVE) instances are caused by such memory-based exploits. However, in CHERI-based systems this insecure behaviour immediately causes runtime errors in the form of untrappable signals, forcing a faulting application to be terminated at once.

Although CHERI systems provide robust security guarantees, our experience is that they require careful programming. This is particularly the case for low-level systems code, which needs significant adaptation including the use of dedicated CHERI intrinsics for managing capabilities, modifications to pointer-based algorithms and data structures, and a new application binary interface [6].

Retargeting a memory management subsystem for CHERI throws up a host of challenges for systems software implementation. In this paper we explore some of these challenges in pragmatic detail. We also demonstrate that a richer awareness of memory structure and hardware-supported metadata provide meaningful opportunities for optimization of memory management algorithms.

We report an evaluation of BDWGC on the CHERIBSD platform running on RISC-V and Morello processor architectures, emulated using QEMU. We use two representative allocation-intensive microbenchmarks integrated with BDWGC. We measure the number of GCs and system calls that occur during benchmark execution. We compare our results against a baseline non-CHERI AArch64 FreeBSD platform. In general, there is some runtime overhead associated with architectural capabilities but this is not prohibitive.

The key contributions of this work are:

- the first technical analysis of a port of an industrial-strength memory management tool to the CHERI platform;
- a discussion of the potential pitfalls and opportunities that become apparent due to the discrepancies between typical assumptions in memory management implementations and the novel CHERI feature set; and,
- a detailed empirical evaluation of the performance implications of our design decisions for BDWGC on the CHERI platform.

The target audience for this paper includes systems software developers who intend to target the CHERI platform, particularly when this involves transferring legacy code from traditional, non-capability platforms. However, we also expect the work will be of general interest to a broader community of programming language virtual machine researchers and implementors.

2 Background

2.1 Garbage Collection

Garbage collection has a long history [7]. It is now widely deployed in the majority of high-level language runtimes.

BDWGC [5] often known as Boehm GC, is an industrial strength automatic memory management framework for C and C++ applications. It is highly tuned for runtime performance, with a range of heuristics [3]. It is a *conservative* collector, since it can operate without runtime time information and infer probable pointer values from typeless bit-patterns. However, BDWGC can leverage available type information—for instance when it is deployed in a Java application context with GCJ.

In BDWGC, a block is a contiguous buffer of one or more pages. A block stores a single large object (at least half a page in size) or multiple small objects of the same size. There are size-segregated freelists for small objects. Per-block metadata is retained in a separate data structure. This metadata is associated with blocks via a hashing lookup structure where the hash value is derived from the block address, for efficient indexing. Blocks can be coalesced into larger blocks or split into smaller blocks according to runtime demand.

2.2 Capability Hardware

The CHERI initiative [13, 16] involves adding certain micro-architectural features to conventional RISC processors to support fat pointers, with hardware enforcement of bounds and permissions. This is a long-running project that continues to gather momentum.

At present, CHERI software is executed via emulation with a well-supported package that uses QEMU, with RISC-V (64-bit) and Morello targets. Morello [2] is an AArch64 processor currently under development by Arm, which instantiates the CHERI principles. Physical boards should be available for evaluation in 2022, as part of the Digital Security by Design programme [11].

A capability-aware variant of the FreeBSD OS, known as CHERIBSD, runs on the emulated systems. There is also excellent LLVM toolchain support [14]. However, to the best of our knowledge, no automatic memory management frameworks have been ported to CHERI prior to our work.

3 Motivation

We work in the *pure-capability* (purecap) context, rather than a hybrid execution mode. This means addresses (in OS kernel interactions, the memory management framework and the client code) are consistently represented as capability values with explicit bounds and permissions. BDWGC and client applications are compiled using pure-capability C and executed in a CheriABI process environment. This gives the typical CHERI assurances of spatial, referential, and (non-stack) temporal memory safety [6].

We incorporate fine-grained *capability support in BDWGC* library code directly. Although this GC framework has been operational since the 1980s, we believe it has no formal correctness proofs. In that sense, it is a best-effort, heuristic-based, conservative GC that is still under active development [4]. Capabilities can serve to make the codebase more robust and resilient in terms of memory safety. In particular, the deployment of capabilities will add a layer of protection to GC metadata which is often the target of code exploits.

We support *capabilities in client code* to provide the typical CHERI set of memory safety guarantees. If BDWGC is to be used as a drop-in replacement for the CHERIBSD platform `malloc` then we must have behavioural equivalence for client applications, particularly with respect to memory accesses.

In this work, we assume the standard CHERI operating threat model [13]. This involves attacks on code in execution, usually to subvert legitimate code or inject malicious code. Exploits are generally attempted using improper memory accesses that violate language semantics but are tolerated in compiler toolchains and runtimes.

4 Challenges

In this section, we explore a number of challenges we encountered while adapting the BDWGC codebase for the CHERI architecture. Typically, each challenge arises from a tacit assumption about the underlying platform: such assumptions hold for all previous 'traditional' BDWGC targets but the new CHERI ecosystem presents radical differences. We highlight these differences and explain the workarounds employed to ensure the code executes properly.

Although this the first detailed presentation of CHERI constraints in an automatic memory management context, the problems we identify have been previously noted in earlier CHERI literature. We use the CheriABI classification [6] in this section.

4.1 Word Size

From the early days of systems programming when BCPL had a solitary word datatype [10] there has been a convention that machine word size is equal to pointer size. Contemporary systems implementation languages maintain this constraint: for example, consider the `usize` type in Rust. However all pointers are 'fat' in CHERI, so systems with 64-bit machine words like Morello actually embed 64-bit addresses within 128-bit capability values.

BDWGC conflates the machine word and pointer size. This is the CheriABI *pointer shape* (PS) issue. To overcome this issue, we set the existing compile-time constant `CPP_WORDSZ` on CHERI platforms to 128 bits, then define a supplementary `INTEGER_WORDSZ` constant which is 64 bits. Further, we ensure all capability values are 16-byte aligned in memory. This means that when we are scanning memory buffers to identify capabilities during GC, we will scan in word-sized units with appropriate alignment.

The word size affects the minimum allocatable unit in BDWGC, known as the *granule* size. We set this to one (128-bit) word on CHERI platforms. This minimum size is required for freelist management, since any unused granule needs to become a cell in a linked list, containing a pointer to the next cell.

We might expect a relative degradation in BDWGC performance on CHERI, given the increase in memory usage and the corresponding drop in cache occupancy. One mitigating factor might be that the original BDWGC configuration specifies the granule size as *two* words, as a compromise between fragmentation overhead and space usage for mark bits, so our CHERI granule size is same as on traditional 64-bit platforms.

4.2 Pointer Synthesis

C programmers often convert implicitly between integer word values and pointers. This might include casts from **`int`** to **`int`**`*`, or complex pointer arithmetic. Unfortunately, these operations do not preserve the validity of capability metadata. This is the CheriABI *integer provenance* (IP) issue, when pointers are cast via integer types other than **`uintptr_t`**.

An instance of this problem crops up when BDWGC traverses all blocks it manages, in the `GC_apply_to_all_blocks (f, data)` higher-order function, which maps `f` over all blocks. There is a hash-based data structure to associate each block address with its corresponding metadata: BDWGC maintains a metadata header object for each block.

The current algorithm uses pointer arithmetic to generate addresses of successive blocks to be traversed. Bitfields (known as *bottom indices*) from these synthetic addresses are used as lookup values in the hashing structure, to fetch the metadata header. Subsequently, BDWGC uses the synthesized address to operate on the block, applying the mapping function as appropriate. The *problem* is that the synthesized address is not a valid capability so cannot be used in a memory access. The *solution* is straightforward: the out-of-band metadata header stores the appropriate block capability value as a field (`hb_block`) that can be accessed directly from the header pointer. Although this memory load will be slower than the integer arithmetic it replaces, it ensures we use a valid capability to access the block.

An alternative workaround involves the use of a *supercapability*: this would be a capability with sufficient bounds and permissions to access any part of memory owned by BDWGC. This means we could then derive valid capabilities as offsets from the supercapability base address. However, we decided *against* this approach since the notion of an 'access-all-areas' supercapability reduces the effectiveness of CHERI in securing systems software like BDWGC, for example if the supercapability is leaked to user code or if a buggy GC memory access accidentally trashes some metadata.

4.3 Block Coalescing

BDWGC handles memory in conceptual units known as *blocks*. Each block is sized at an integer multiple of the page size, with the minimum block size being one page. For small objects (less than 0.5 pages) each block stores a set of objects with the same size. For large objects (at least 0.5 pages) each block stores a single large object. Blocks are requested on-demand by BDWGC during execution, via `mmap` calls.

The process of *block coalescing* merges empty contiguous blocks into a single larger block. This simplifies BDWGC metadata management and allows for larger objects to be accommodated in the existing heap rather than requiring fresh `mmap` calls. It is possible subsequently to decompose a large block into smaller blocks, if necessary.

This coalescing operation is common in memory management systems in general. Unfortunately, the tight bounds enforced by CHERI capabilities means such coalescing is not possible using capabilities derived from OS calls to `mmap`. A capability cannot 'grow' its bounds, only reduce them. Therefore, merging contiguous blocks into a single block,with a single capability spanning that block, is not supported. This relates to the CheriABI issue known as *Monotonicity* (M). The problem becomes apparent when executing code attempts to reach outside capability bounds or increase memory access permissions.

In our CHERI adaption of BDWGC, bounds enforcement means coalescing is not possible unless we are coalescing something that had previously been split, but was originally part of a single capability returned by one `mmap` call. We add CHERI bounds checks on block capabilities and only perform coalescing where the capability of the lower block would span the full range of the coalesced pair of blocks.

We note that a supercapability (as already discussed in Section 4.2 above) would mitigate this problem but it brings other disadvantages.

4.4 Bounds Precision

Strictly speaking, this is not a legacy C problem; rather this is a challenge intrinsic to all CHERI platforms. A capability value encodes a base pointer, a limit and an offset. Each of these three fields is a distinct pointer, in theory requiring a machine word of storage. There are additional bits required for metadata including access permissions. An uncompressed CHERI capability conceptually occupies 256 bits.

CHERI Concentrate [15] is an elegant capability compression scheme that reduces the footprint of a capability value to 128 bits. However this shrinkage comes at the cost of some loss of precision in terms of representable base addresses and bounds. For capabilities with small bounds, there is no precision problem. However the problem becomes apparent as capability bounds grow.

```
if (cheri_gettag(limit) == 0) {
  limit -= ALIGNMENT;
} else { // ...
```

Figure 1. Testing for capability tag during a memory scanning loop in the BDWGC mark phase

The most pragmatic way to handle this issue is to over-approximate capability bounds so they become precisely representable. A memory allocator would pad buffers with unused memory to maintain capability integrity. While we have not encountered this issue in our small-scale testing of BDWGC, we remain aware of the problem and are writing code, including runtime checks, in a precision-aware manner.

5 Opportunities

After reading Section 4, one might be forgiven for thinking memory manager implementation on CHERI is fraught with difficulty and has many negative performance implications. On the other hand, in this section we sketch out two optimization opportunities afforded by CHERI. We demonstrate that capabilities not only make memory management more secure but they also have the potential for improved efficiency.

5.1 Tagged Capabilities

CHERI architectures have out-of-band metadata, with a single bit per 16-byte aligned memory location used to indicate whether a valid capability is stored at the corresponding location. In principle, checking such a capability tag bit should be a fast operation.

Since BDWGC is a conservative collector, in the absence of runtime type information it needs to scan all values in user memory with the assumption that any bit-pattern could represent an address. In practice, BDWGC uses sophisticated heuristics to eliminate the majority of non-pointer values. However, when BDWGC is deployed in contexts with rich runtime type information (e.g. GCJ) then it can take advantage of this information to reduce scanning overhead during the performance-critical mark phase of GC.

Similarly on CHERI, we can use the capability tags to direct the scanning of data to exclude non-capability values, with a consequent reduction in the marking phase overhead. The code in Figure 1 illustrates how this efficient scanning loop jumps over non-capability values in a memory region.

While it is possible to derive capabilities from integer offsets to other capabilities, this is a highly controlled operation in CHERI and the derived capability cannot exceed the bounds of the original capability. This means that we *never* need to trace non-capability values in BDWGC on CHERI, since any reachable data will always have at least one valid capability value in user memory space.

More generally, CHERI's explicit tagging for pointers will enable significant GC optimizations for C code. This might include support for a copying collector: where pointers can always be identified, it should be possible to safely update them when moving objects.

5.2 Unmapped Areas

BDWGC identifies unused blocks of memory with the `GC_unmap` call, for instance for a page that is part of the runtime heap but does not contain any live data. On BSD systems, `GC_unmap` is a `mmap` system call applying `PROT_NONE` to the relevant page(s) which makes them subsequently inaccessible.

If this unmapped block is required in future, perhaps if the live data size increases, the `GC_remap` call undoes the unmapping. On BSD systems, `GC_remap` is a `mprotect` system call increasing the permissions of the relevant page(s) so they become accessible again.

Unmapping serves two purposes:

- to prevent wild memory accesses to unused blocks in the heap—effectively the unmapped pages are *guard pages*; and,
- to hint to the OS that such pages may be swapped out since they are not in active use.

In our capability-aware BDWGC, wild memory accesses are no longer possible due to the strict bounds enforcement of CHERI. Therefore we decided to eliminate the `mmap` and `mprotect` system calls, effectively making `GC_unmap` and `GC_remap` into null operations. This reduction in system call overhead may improve performance. Further, the `mprotect` call which increases permissions may not be permitted under CHERI monotonicity constraints unless a supercapability value is available.

Note the Emscripten port of BDWGC for WebAssembly also eliminates the same pair of system calls since this OS-level memory management is not available in the WebAssembly runtime. Unmapping is intended to provide hints to the OS about page residence requirements, however modern OSs have robust heuristics to handle memory over-commitment.

6 Evaluation

This section gives a preliminary empirical evaluation of our BDWGC adaption for CHERI platforms. After reviewing our methodology (Section 6.1), we consider the runtime performance of memory management with capabilities (Section 6.2), the developer effort involved in the port (Section 6.3) and a brief discussion of threats to validity (Section 6.4).

6.1 Methodology

Platforms. Of necessity, our performance evaluation is only indicative since we do not yet have access to physical hardware for full-system experimentation. Instead we rely on running software emulations of CHERIBSD sessions in QEMU; these are configured for the 64-bit RISC-V and Morello platforms, which use 128-bit capability values. QEMU is a functional full-system emulator. As such, it does not have a cycle-accurate processor model so we cannot obtain end-to-end timing results.

As a baseline comparison, we also use an Amazon Web Services (AWS) t4g.micro Graviton2 instance, running FreeBSD 12. This platform features an AArch64 Neoverse N1 CPU, which shares the same base Arm processor specification as the Morello system. Since the Graviton2 system does not support capabilities, we use raw pointers in the benchmarks. We also report Graviton2 *padded* results, where each 64-bit pointer field is padded to 128 bits—this is intended to achieve a fairer comparison in terms of data structure size and heap occupancy.

Benchmarks. We select two simple C benchmarks as representative memory-intensive workloads:

- the *binarytrees* program from the Computer Language Benchmarks Game (CLBG), which allocates one long-lived binary tree then a number of short-lived trees of various depths in a tight loop (we set the `max_depth` parameter to 6); and,
- a *random mixed allocation* program, which allocates 32,000 short-lived objects in a tight loop, where each object has a pseudo-randomly chosen size, uniformly distributed between 16 bytes and 16KB.

Each benchmark is compiled and executed with our fork of BDWGC configured for Debug mode, with all of GCJ, parallel mark, multi-threading and dynamic loading all switched off.

Metrics. Since our evaluation platforms are either emulated (QEMU) or virtualized (AWS), we do not report any wall-clock time results. Instead we measure runtime event counts, such as number of GCs or system calls during benchmark execution. In principle, these should be relatively platform-independent metrics that provide indications of GC performance.

6.2 Performance Impact

In a first evaluation study, we profile the number of GC events for each fixed workload on each platform. This will inform us about the impact of heap fragmentation on CHERI platforms (RISC-V and Morello) due to the lack of block coalescing. Note that coalescing is enabled on the Graviton platform. We are also interested in the memory overhead of using 128-bit capabilities (RISC-V, Morello and Graviton-padded) as opposed to raw 64-bit pointers (Graviton).

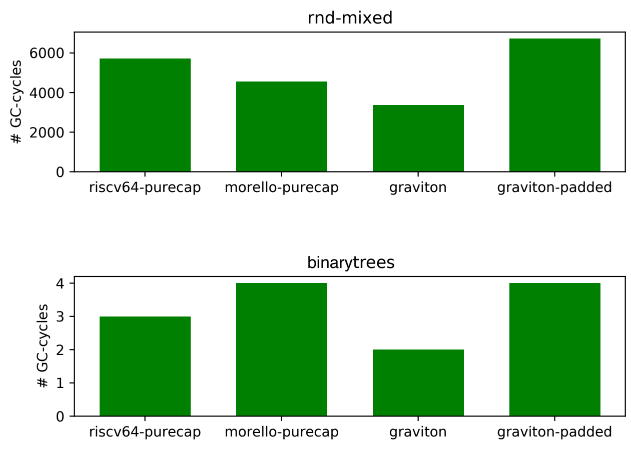

Figure 2. Number of GCs during micro-benchmark execution on various platforms

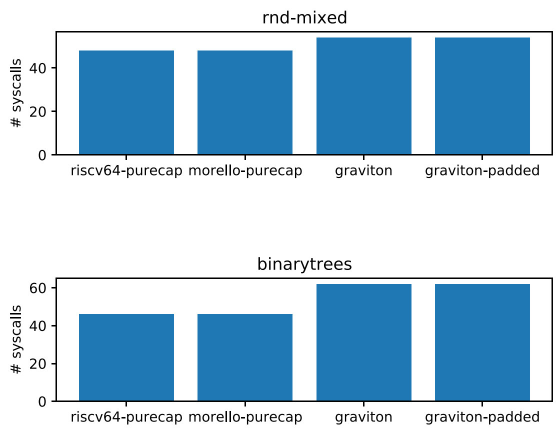

Figure 3. Number of `mmap` and `mprotect` system calls during micro-benchmark execution on various platforms

The results in Figure 2 indicate that the larger capability size does have a significant impact on GC overhead for allocation-intensive workloads; the Graviton results are noticeably lower in both benchmarks. However we do not observe significant inefficiency due to lack of coalescing; the Graviton-padded results are not lower than the RISC-V or CHERI results. We realise that more complex benchmarks with irregular allocation patterns might be more likely to exercise the coalescing behaviour; we reserve this for future work.

In a second study, we use the *truss* tool to count the total number of `mmap` and `mprotect` system calls during each benchmark run on each platform. Recall from Section 5.2 that we have eliminated system calls for BDWGC unmapping operations, which may reduce runtime overhead.

The results in Figure 3 indicate a slight reduction in system call counts on the two CHERI platforms, because of the increased efficiency in the GC block mapping behaviour.

6.3 Developer Effort

Our port of BDWGC to CHERI is ongoing. So far we have spent around *four person months* of full-time developer effort, although most of this initial work involved understanding the sophisticated internals of BDWGC and the complexities of CHERI. We are now at the stage where we can run simple benchmarks and integration tests.

To date we have only modified around *100 lines of code* (LoC) in a total codebase size of 60k LoC. We expect to make further changes, but our work should be commensurate with the 0.026% LoC change rate in approximately 6 million lines of C and C++ code that introduces CHERI memory safety for a Unix desktop system [12].

6.4 Threats to Validity

We have only considered a simplistic tracing GC algorithm, which is stop-the-world, single-threaded and non-moving. However the issues identified in this paper will arise in more complex memory managers on CHERI systems. Our performance evaluation is high-level, involving only software event counts on emulated platforms running basic allocation workloads. When Morello physical systems become available, we intend to conduct full stack performance evaluations on a wider range of workloads.

7 Conclusions

We have presented an in-depth technical discussion of the challenges and opportunities arising when transferring a legacy C automatic memory management framework to the CHERI ecosystem. We have indicated pragmatic workarounds for the challenges we faced in the porting endeavour.

Our initial empirical evaluation shows that there may be some performance hit to be expected with capability-based memory management, but this does not appear to be excessive. We eagerly await the physical Morello system in order to carry out a larger-scale, in-depth performance evaluation. In particular, we are presently unable to assess the micro-architectural impact of our changes in terms of caching behaviour.

In this research project, we encountered many similar issues to those identified by the CheriABI team. We hope that our anecdotal experience of memory management for CHERI will be valuable to other systems implementors as they attempt to support the ongoing CHERI initiative with a wide variety of software ports.

Acknowledgments

This work was supported by the Engineering and Physical Sciences Research Council grant number EP/V000349/1 as part of the Digital Security by Design (DSbD) programme.

References

[1] ACM SIGPLAN. 2020. Programming Languages Achievement Award. https://www.sigplan.org/Awards/Achievement/

[2] Arm. 2021. Arm Architecture Reference Manual Supplement — Morello for A-profile Architecture. https://developer.arm.com/documentation/ddi0606/aj/?lang=en.

[3] Hans-Juergen Boehm. 1993. Space Efficient Conservative Garbage Collection. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*. 197–206. https://doi.org/10.1145/155090.155109

[4] Hans-J Boehm et al. 2021. Boehm-Demers-Weiser Garbage Collector. https://github.com/ivmai/bdwgc/

[5] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage collection in an uncooperative environment. *Software: Practice and Experience* 18, 9 (1988), 807–820. https://doi.org/10.1002/spe.4380180902

[6] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. 2019. CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-Time Environment. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 379–393. https://doi.org/10.1145/3297858.3304042

[7] Richard Jones, Antony Hosking, and Eliot Moss. 2016. *The garbage collection handbook: the art of automatic memory management*. CRC Press.

[8] MITRE Corporation. 2021. 2021 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html

[9] Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. 2020. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1003–1020. https://doi.org/10.1109/SP40000.2020.00055

[10] Martin Richards. 1971. The portability of the BCPL compiler. *Software: Practice and Experience* 1, 2 (1971), 135–146. https://doi.org/10.1002/spe.4380010204

[11] UKRI. [n. d.]. Digital Security by Design: Securing The Future of the Digital Economy. https://www.dsbd.tech.

[12] Robert N. M. Watson, Ben Laurie, and Alex Richardson. 2021. Assessing the Viability of an Open-Source CHERI Desktop Software Ecosystem. https://www.capabilitieslimited.co.uk/pdfs/20210917-capltd-cheri-desktop-report-version1-FINAL.pdf.

[13] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Simon W. Moore, Steven J. Murdoch, and Michael Roe. 2014. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-set architecture*. Technical Report UCAM-CL-TR-850. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-850

[14] Robert N. M. Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W. Moore, Edward Napierala, Peter Sewell, and Peter G. Neumann. 2020. *CHERI C/C++ Programming Guide*. Technical Report UCAM-CL-TR-947. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-947

[15] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert M. Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W. Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. 2019. CHERI Concentrate: Practical Compressed Capabilities. *IEEE Trans. Comput.* 68, 10 (2019), 1455–1469. https://doi.org/10.1109/TC.2019.2914037

[16] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 457–468. https://doi.org/10.1109/ISCA.2014.6853201