AdaCore TECH PAPER

Security by Default - CHERI ISA Extensions coupled with a security-enhanced Ada runtime

P. Butcher | D. King | J. Kliemann

Security by Default - CHERI ISA Extensions Coupled with a Security-Enhanced Ada Runtime

Abstract

In an age where security breaches and cyberattacks have become increasingly prevalent, the need for robust and comprehensive security mechanisms within embedded real-time systems is paramount. We propose a novel solution to enforce fault-detection and increase security assurance: "Security by Default", specifically combining Capability Hardware Enhanced RISC Instructions (CHERI) ISA microprocessor extensions with a CHERI pure-capability compliant Ada runtime. We present case studies showing how combining memory-safe hardware with memory-safe software results in a mutualistic layered approach to security and increases assurance of embedded real-time systems. We argue that this satisfies regulatory security verification objectives outlined in standards like the "Airworthiness Security Process Specification" (DO-326A/ED-202A[1] [2]).

Keywords: Cyber-security, CHERI, Ada, Airworthiness Security, Memory-safety, embedded real-time systems

1. Introduction

As the UK's National Cyber Security Centre (NCSC) aptly states, "Secure by Default" is defined as "technology which has the best security it can without you even knowing it's there or having to turn it on"[3]. This principle served as the guiding philosophy of our research as we set out to evaluate the security assurance claims being made over the adoption of a CHERI compliant microprocessor and a CHERI pure capability runtime environment that understands how best to benefit from the CHERI extended instruction set architecture (ISA). More specifically, this paper describes the development steps and subsequent evaluation of a security-hardened Ada runtime executing on Arm's Morello CHERI extended ISA microprocessor[4]. The goal of the research was to demonstrate and evaluate a layered approach to security that avoids common failure modes and provides security with significantly reduced effort.

Whilst at first glance it may seem unnecessary to implement a CHERI pure-capability compliant runtime for a memory-safe programming language[5], our research shows that the two technologies are complementary, and although there are over-laps in memory safety checks, the limitations of one approach are overcome by the feature set of the other. This paper details why combining a memory-safe programming language and runtime with a memory-safe microprocessor results in a security framework upon which developed embedded real-time systems are resilient to attack and capable of attack recovery.

This paper assesses security claims about CHERI regarding the benefits of fine-grained memory protection, and the enabling of granularity in memory access controls that, while not entirely novel (other historical solutions have come and gone), are claimed to elevate security assurance levels of software executing on CHERI hardware. CHERI offers execution security through dynamic fine-grained memory protection checks, which offer a different approach than other microprocessor security features like trust zones and secure boot[6]. This paper documents our research findings around the security assurance impact when systems utilize CHERI to precisely define which portions of memory are accessible and which are off-limits. This evaluation includes

measuring the potential reduction of a system's attack surface, CHERI's overall ability to minimize memory-based vulnerabilities (for example, buffer overflows and data injection attacks), and the impact reduction following unauthorized electronic interaction.

Moreover, this paper quantifies the reduction in risk of privilege escalation and unauthorized data access. CHERI's object capabilities grant programs the ability to manage and control access to their data structures and resources with precision. This level of control can be utilized by systems to enact robust isolation between components.

Our research shows that CHERI adoption can not only provide a security layer to applications written in a memory-unsafe language like C but also provide a benefit to the adoption of applications written in a memory-safe language like Ada or Rust[5]. Thus CHERI can enhance critical systems, offering high resilience against security threats. This paper explores the integration of CHERI ISA extensions into high-integrity embedded systems and answers the question: Can CHERI bring extended memory protection and isolation, thereby enhancing the security posture of high-integrity, real-time embedded systems?

Ada, as a high-integrity programming language, is focused on supporting high-assurance, safety-critical, and embedded real-time systems. Over the years, Ada has demonstrated its value in various domains, including aerospace[7], defense[8], rail[9], space[10] and multiple other safety-critical applications. Its success is attributable to several key attributes, including its robust type system, rigorous runtime checking, a history of reliability and the availability of qualifiable freely licensed open source tooling. Historically, these characteristics have made Ada a valuable language for critical systems where maintainability and safety are paramount.

This paper introduces and evaluates a security-enhanced Ada runtime that extends the freely licensed open source GNAT Pro Ada runtime environment[11] with a tailored set of security features that align with the CHERI architecture. To assess the security benefits of coupling CHERI hardware with the Ada programming language, we have developed spatially safe and CHERI pure-capability compliant memory allocators within the GNAT Pro Ada runtime. Spatial safety ensures that out-of-bounds memory accesses beyond the bounds of the allocated memory are detected. In addition, by leveraging Ada's runtime exception handling, we have implemented a mechanism to propagate CHERI-hardware-detected memory vulnerabilities into software exception handlers. We have also assessed the different approaches the Ada language and CHERI have taken to bounds checking, and we argue that joint adoption provides a defense-in-depth approach.

1.1 Introduction to CHERI

CHERI is an extension of the RISC (Reduced Instruction Set Computer) architecture to enhance memory safety and security in computing systems. CHERI is a joint research project of SRI International and the University of Cambridge[12], CHERI introduces new instructions and architectural features to enable fine-grained memory protection and mitigate common security vulnerabilities.

CHERI aims to improve memory safety by providing fine-grained protection mechanisms, reducing the risk of memory-related vulnerabilities such as buffer overflows and dangling pointers. CHERI is a microprocessor ISA hardware security toolkit for developing high-assurance software runtime environments to secure application execution.

The CHERI instruction set architecture introduces security extensions to standard memory addresses (i.e. pointers) via the concept of capabilities. More specifically, CHERI capabilities extend standard pointers with: a capability tag used to define the validity of the capability, a specification of the bounds of the accessible memory region within the address space, a set of permissible actions related to the memory and an object type field used for sealing capabilities (making them immutable and non-dereferenceable).

CHERI enforces access control policies that could be missed at the software level by directly

incorporating processing checks of capabilities into the hardware architecture, reducing the attack surface for malicious exploits. In addition, CHERI was designed to be compatible with existing software, allowing for gradual adoption and integration into existing computing systems without requiring significant changes to software development practices[12].

The CHERI Tag extension is a 1-bit value that defines the validity of the memory address. Any attempt to dereference invalid capabilities will result in a CHERI processor exception.

The hardware will raise a capability bounds fault hardware exception if an instruction attempts to dereference a capability when the associated virtual address is outside the configured capability bounds. In addition, the capability tag is automatically cleared by the CPU architecture when instructions attempt to change the configuration of a capability in an invalid way (for example, when trying to increase a capability's bounds). In addition, CHERI supports the concept of fine-grained memory protection by enforcing capability inheritance. Capability inheritance ensures that capabilities cannot be created out of thin air; instead, capabilities are derived from other valid capabilities and inherit the parent capability's bounds and permissions. Capability inheritance is also monotonic; capabilities cannot increase their bounds or permissions beyond those inherited from the parent capability. This policy allowed us to build an Ada runtime fine-grained memory protection model that adheres to the principle of least privilege. ***Figure 1*** shows the memory layout and some of the enforcement policies made available via capabilities.

CHERI introduces new instructions for working with capabilities, such as loading and storing capabilities in memory, restricting the bounds and permissions of capabilities, and sealing and unsealing capabilities. These instructions allow for the creation, manipulation, and enforcement of capabilities within the hardware architecture. Additionally, CHERI includes instructions for performing capability-based memory accesses, bounds checking, and permission checks to ensure secure and authorized access to memory regions.

Furthermore, CHERI introduces new registers

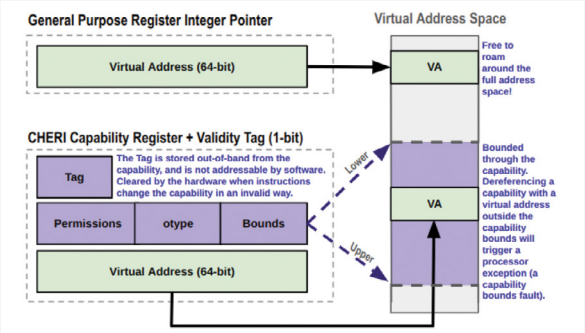

Figure 1: General Purpose Register Integer Pointer vs CHERI Capability

dedicated to storing capabilities, such as the capability register file. These registers hold capabilities that represent specific memory regions and include metadata such as base address, bounds, and permissions. The capability register file provides a means for managing and manipulating capabilities within the hardware, enabling efficient enforcement of memory protection policies and access control mechanisms.

1.2 Introduction to the Ada Runtime

The Ada language provides a significant set of features including multitasking, exception handling and memory management. To provide these features it requires a runtime library. The runtime library is similar but not identical to a standard library seen in other languages. It provides both interfaces for the compiler and the programmer to use. Some parts of the library are intended to be used by programmers, such as I/O interfaces while others are used indirectly like the multitasking interface.

The runtime library and the compiler work together to provide the complete set of Ada features to the programmer. Features like tasking or returning variable sized objects from functions are defined in the language itself but require runtime support. If one of these features is used, the source code itself will not contain any direct reference to the runtime. Instead the compiler will generate the required code to call the runtime library transparently to the user. The following paragraphs will go through the major features in more detail.

The Ada language and its runtime, support a feature called elaboration. It is used to ensure the initialization of all global objects in the proper order. It also takes care of initializing the runtime itself before starting the program execution. This feature is implemented as an additional step in the compilation process. After compilation, the compiler will call the binder. The binder evaluates the dependencies of both the program and the runtime features it uses and creates the proper initialization code. It will also detect dependency issues in the initialization such as circular dependencies. After the code generation the compilation process will continue similar to other languages.

The runtime also handles a part of the memory management that is typically not present in other languages. Ada supports returning variable sized objects from functions without requiring a heap. In a compiled language the compiler in part takes care of managing the stack. While code can use the stack for objects whose size is known only at runtime it can only do so on the top of the stack. When a function is called the memory for its return value is stored on the stack before the new stack frame is created. This requires knowledge about the size of the returned data. Otherwise the return value may overflow its allocated memory. Ada solves that by having a secondary stack. This secondary stack is not used for stack frames and therefore can be used for dynamically sized objects at runtime. If a function returns a variable sized object the compiler will emit code that uses the secondary stack to allocate memory for this object when returning the object. The stack implementation itself is part of the runtime library as this allows it to be adapted to the underlying system.

Tasking, or threading, in Ada is supported at the language level with the language providing a specific syntax and semantics for easy use by the programmer. It also allows an abstraction away from the underlying platform. Furthermore the language allows different tasking profiles providing different feature sets depending on the application. The compiler will generate expanded code specific to the chosen tasking profile and feature set which will call the runtime library providing the implementation specific to the underlying platform.

While the list of features highlighted here is not exhaustive it provides an overview of the specific features the Ada runtime library provides to the language, different from a typical standard library. The runtime provides the interface between the expanded code and the underlying system.

GNAT Pro provides a diverse set of runtime libraries for different targets and use cases. These range from the native runtime on native targets, which supports a wide selection of features, such as networking and file handling, to bare metal runtimes for resource-limited targets, which still en-able the use of features like tasking and the secondary stack without the need for an underlying operating system. Bare metal runtimes are typically deployed on embedded real-time systems.

1.3 Introduction to the Edge Avionics Project

The research described in this paper is funded by the Rapid Capabilities Office (RCO) of the UK Royal Air Force (UK RAF) via the Edge Avionics project ('Edge Avionics'). Edge Avionics is a consortium led on behalf of the RCO by the Defence Science and Technology Laboratory (Dstl, an executive agency of the UK Ministry of Defence (MOD)[13]) and delivered by GE Aerospace[14] (the prime), Wind River[15] and AdaCore[16]. The project aims to demonstrate a network of secure units running a distributed application at scale and capable of demonstrating resilience at the network level. A Dstl-owned and modified air platform mission system will be used to check the impact of the new security controls. Through Edge Avionics, the Edge Avionics consortium can substantiate CHERI security claims within a defense environment whilst investigating legacy software rework overheads.

2. GNAT Pro for Morello Ada: Security by Default for Embedded Real Time Systems

This paper presents the architectural details of multiple toolchains for GCC[17] and LLVM-targeted[18] bare-metal Morello systems. In all cases, we have focused on Ada cross-compiler configurations, basing them on the freely licensed open source GNAT Pro for the bare-metal product line, augmented to support an Ada application benefiting from CHERI features.

The toolchain focuses on using CHERI to enforce spatial memory safety between objects in memory. For each object in memory (whether on the stack, heap or statically allocated), a capability is created at runtime whose bounds are restricted to the size of the allocated object. Restricting the bounds prevents out-of-bounds access via a pointer to one object from being able to access another object. The compiler and runtime work together to ensure that all memory allocations are correctly bound and with the correct permissions. The compiler manages the allocation of objects on the primary stack, and the runtime manages heap and secondary stack allocations.

The compiler is also responsible for making sure that the bounds and permissions, obtained when an object is allocated, are subsequently used for all accesses made to the object or parts of it. That is a relatively easy task in Ada because the language maintains a strict separation between addresses (of objects) and offsets that may be added to these addresses during regular operations. In cases where this separation is broken, which can happen only by using very low-level devices, the compiler gives an explicit warning.

For objects on the primary stack, the compiler sets up a capability for each object on entry to the stack frame. Each capability is set up to point to the portion of the stack frame that is allocated for the object, with the bounds set to the allocated region. These capabilities are inherited from the capability stack pointer (CSP) which prevents out-of-bounds access in case of a stack overflow.

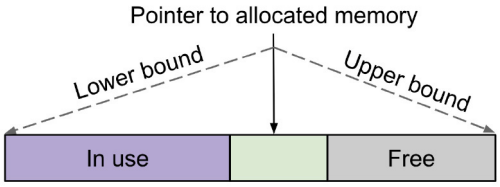

Figure 2: Memory allocation without spatial safety. The allocated pointer (capability) inherits the bounds to the entire heap/stack and can access other allocated objects.

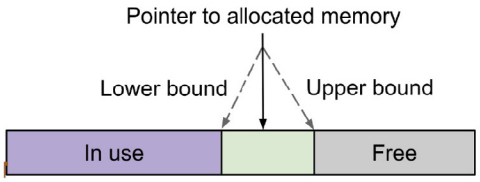

Figure 3: Memory allocation with spatial safety. The allocated pointer's (capability) bounds are limited to the size of the allocated block. The pointer cannot be used to access other allocated objects.

The heap and secondary stack memory allocators in the run-time have been augmented to take advantage of CHERI and provide spatial safety between memory allocations. Each allocator has a capability to the entire block of memory assigned to the allocator, which is used to derive capabilities to fulfil allocation requests. For each allocation, the allocator returns a pointer to the allocated memory with the bounds limited to the size of the allocation to enforce spatial safety. The difference between the enforced bounding of the memory allocation is shown in ***figures 2 and 3***.

For runtimes built with exception propagation enabled, the runtime implements a mechanism to catch CHERI processor exceptions and convert them into Ada exceptions that can be propagated, caught, and handled by user code. This is discussed further in ***section 3.2.1***.

3. Security Assessment Evaluation

Fine-grained memory allocation is a foundational element of the developed security-enhanced Ada runtime. By implementing tightly bounded memory allocation, stringent memory safety checks can be performed by the CHERI hardware. Our research shows that fine-grained memory allocation prevents common vulnerabilities related to memory manipulation, such as buffer overflows and data corruption. These checks ensure that memory accesses remain within predefined bounds, minimizing the potential attack surface and acting as countermeasures to attacks that exploit memory vulnerabilities. Whilst a regular Ada runtime comes with protection over the safe usage of the Ada language, the security-enhanced Ada runtime also protects the parts of the language that are considered unsafe, like memory overlays.

3.1 CHERI Limitations

Overall, CHERI provides the means to increase security assurance. However, it does not guarantee it, furthermore, the benefits can only be realized with the correct usage of the instructions and registers. More specifically, CHERI is meaningless without a CHERI-compliant software runtime. However, by integrating capabilities directly into the hardware architecture CHERI provides a solid foundation for developing demonstrably more secure software runtimes for building more secure and resilient computing systems.

In addition, the protection offered by CHERI is highly dependent on compiler configuration, and multiple factors must be considered when assessing the security assurance offered by adopting a CHERI-based CPU[19]. One major factor is pure-capability vs hybrid mode. The code uses CHERI capabilities for all pointers when configured in pure-capability mode. In comparison, hybrid mode allows a mix of standard RISC pointers and CHERI capabilities (typically configured via source-code annotation). Whilst hybrid mode has the benefit of integrating legacy systems with greenfield CHERI-enabled development, the downside is that we can no longer guarantee all pointers are capabilities, making security arguments harder to write.

Furthermore, our research has found that Ada code often requires low effort to port to CHERI, and in many cases, we have found that the codebases worked with no changes. Therefore, to focus our work on the highest levels of security assurance we developed bare-metal Ada CHERI compilers that only support pure-capability mode and enforce that all pointers (programmer or compiler generated) benefit from CHERI capability protections. As discussed later in this paper, this required extensive changes to the GNAT Pro Ada runtime[11].

By only supporting pure-capability CHERI, we enforce that all pointers are represented in memory as CHERI capabilities and that manipulation of a capability, and therefore the associated Ada object, can only be performed through capability instructions. Furthermore, the developed runtime must adhere to the strict CHERI rules around pointer integrity and monotonicity, specifically that valid capabilities cannot be created out of thin air; they must be derived from other, valid capabilities and the inherited bounds and permissions cannot be broadened. This results in the following list of unsupported Ada features which would require the creation of pointers (capabilities) to arbitrary memory addresses at runtime:

- `Ada.Tags.Internal_Tag` (used for Ada tagged types to convert an external tag - a string representation of an address - to an Ada tag, which is implemented as a pointer in GNAT)
- `Ada.Tags.Descendant_Tag` depends on `Ada.Tags.Internal_Tag`)
- Reading Ada tagged objects from streams (via S'Class'Input, as this depends on `Ada.Tags.Descendant_Tag`)
- Reading addresses/pointers from streams (more specifically, Ada access types can be streamed to media, but we can not support streaming the data back into an access type)

This prerequisite would unlikely pose a significant issue for developing an Ada application on the presented solution. Furthermore, a high-integrity system, as often found in an embedded real-time system, would likely have been developed under strict guidelines prohibiting using these Ada features. In addition, for the Ada.Tags.Internal_Tag limitation, this is not a limitation of the Ada language, but rather reflects the way GNAT currently implements this feature. There are alternative implementations of Ada.Tags.Internal_Tag that would not need to create capabilities out of thin air.

3.2 Regular Ada software checks Vs CHERI runtime hardware checks

Ada provides various language-defined run-time checks to protect against detectable bugs such as out-of-bounds array accesses, range violations, integer overflow, and null pointer dereferences. Run-time check failures raise exceptions, which can be caught and handled by user-defined exception handlers to gracefully recover from the error whilst ensuring unsafe instructions are not executed. During the work to port the GNAT Pro to bare-metal Morello architecture, we evaluated the possibility of replacing these software run-time checks with the CHERI hardware run-time checks to reduce the overhead of the checks at run-time. We identified two kinds of run-time checks for consideration: null pointer checks and array bounds checks.

CHERI performs more robust pointer validity checks than Ada. CHERI verifies the validity of a pointer by inspecting the "tag bit" and ensures the control flow cannot dereference pointers with an invalid tag. By contrast, Ada's checks can only detect null pointers; non-null pointers that refer to an invalid memory location can still be dereferenced.

We evaluated using CHERI's bounds checking to implement Ada's semantics for array index checking. However, we found two issues that prevented CHERI from being able to implement the semantics required by Ada's language-defined checks[20]. The rules of the Ada programming language require raising a `Constraint_Error` exception before accessing an array with an index value that is not within the bounds of the array index type. This requires the bounds check to be precise, even for very large bounds. The first issue is that Morello architecture uses a compressed bounds format, ensuring the bounds are precise for objects up to 4 KB[21]. For objects larger than 4 KB, however, the bounds must be aligned to increasingly more significant powers of 2 address boundaries. This prevents the bounds for large objects from being represented precisely and thus requires padding in the memory allocation to align the capability bounds, meaning that CHERI will not detect minor accesses past the end of the array in the padding area. The second issue is that array types in Ada can be unconstrained where the array bounds are not known at compile time. This requires the array's bounds to be stored and passed alongside the array object at runtime, and this additional data must be within the bounds of the underlying CHERI capability. These two issues mean the CHERI bounds can be larger than the bounds checks enforced by the Ada language.

While Ada's compile-time and run-time checks ensure the correct usage of many parts of the language, some "unsafe" parts do not have associated run-time checks and instead rely on the programmer to ensure correct usage. The term "unchecked" generally indicates Ada language features not covered by language checks, such as Unchecked_Conversion, which is used to cast between unrelated types. One particular feature considered unsafe is memory overlays, where an object is allocated (overlaid) at the same address as another object, introducing aliasing. There are no Ada language-defined checks associated with memory overlays, so it is up to the programmer to ensure that the two objects have compatible sizes and alignments and to avoid causing invalid data representations.

CHERI's hardware run-time checks cover this gap and provide memory safety for these unsafe parts of the language. For example, in the case of a memory overlay, the overlaid object inherits the capability of the target object. This ensures that any attempt to access beyond the bounds of the target object will be prevented at run-time.

3.2.1 Beyond Language-defined Run-time Checks

CHERI's hardware-level run-time checking provides additional memory safety assurances beyond language-defined run-time checks. The hardware checks apply to all code, including compiler-generated code that is not otherwise subject to run-time checks. This can reveal errors and defects in parts of the code that would otherwise go unnoticed and could potentially lead to exploitable security vulnerabilities.

While porting the GNAT Pro Ada compilers to Morello, we discovered a regression introduced in an unreleased development version of the compiler front-end that led to an out-of-bounds memory access. The regression occurred in a specific case where a function returns a variable-sized object whose size is known only at run-time. In this case, the compiler allocates memory on the secondary stack to store the returned object. The error was that the compiler used the wrong object size in the call to the secondary stack allocator, resulting in the allocation being too small to store the object. Subsequent memory accesses to the returned object could then access memory beyond the end of the allocation, potentially accessing other adjacent objects on the secondary stack which could have led to an exploitable vulnerability.

This regression was found only by running our existing test suites on Morello with our spatially safe secondary stack allocator. Running the same test suites on conventional architectures did not detect the regression. We also ran the test suites with Valgrind[22] and AddressSanitizer[23] which were also unable to detect the regression as all memory accesses were still within the bounds of the underlying buffer used for the secondary stack.

3.3 Enhanced Security over Language Bindings

Static checks completed by the compiler ensure a significant part of code correctness. While the programmer must define types and function signatures properly, the compiler can check for violations of these constructs. In C, this includes the `const` modifier. If used in a function signature, the caller can be sure that the passed value will remain unmodified throughout the call. While there are ways to circumvent this check, it has to be done explicitly. Without this circumvention, the compiler verifies the value remains unmodified inside the function. Utilising the `const` modifier is a design decision that improves the understandability of the code and reduces the risk of wrong assumptions.

Ada employs a similar mechanism where function arguments can be specified with the modifiers `in`, `out`, and `in out`. Parameter mode `in` is the default and enforces that a value is only passed into the function and must not be modified. It is similar to the const modifier in C. Parameter mode `out` requires the callee to set the value, allowing the caller to assume initialization after the call. The modifier `in out` specifies that the caller must pass an initialized value, which the callee may modify. Additionally, when passing arrays into a function, the array value contains information about its bounds, allowing the compiler to insert runtime checks if required.

While the compiler can do many checks and prevent many problems, its scope is limited to the constructs of the language. However, the compiler only creates machine code from source code and does not connect the built parts of code, typically object files. The linker does this task. The linker, having only access to object files and symbols, cannot check types or even function signatures when linking object files into an application. It resolves symbols in object files with addresses, ensuring the correct place in the final binary is executed when the corresponding function is called.

This limitation is acceptable if the compiler can check types and signatures. Still, it stops working when multiple languages are used in the same project and foreign function interfaces are invoked for calls between languages. A foreign function interface is a feature that allows the programmer to tell the compiler that a particular function is imported and not defined in the same language. Imported means that the compiler will keep references to the imported function unresolved and instead expect the linker to resolve them. For these inter-faces to be used, the programmer must declare the imported function to have the same arguments, argument

types, argument modes and a calling convention as defined in the other language. It cannot check whether the imported declaration in one language conforms to the exported definition in another.

Consider the following C and Ada code:

```
void print_string (const char *s, size_t len ){
    for(size_t i = 0; i < len; i ++){
        putchar(s[ i ]);
    }
}
```

This C function prints a string by iterating over its characters and printing each of them. The input string `S` is passed by pointer and is declared `const`, forbidding the function from modifying it. The following Ada program will import the function and use it to print a string:

```
with Interfaces .C; use Interfaces.C

procedure Main is
   procedure Print_String (S : char_array; Len : size_t)  with
      Import,
      Convention => C,
      External_Name => “print_string”;
    Some_String : constant char_array := “Hello World!”;
    Backup_String : constant char_array := Some_String;
begin
     Print_String (Some_String, Some_String’Length);
     pragma Assert (Some_String = Backup_String);
end;
```

The Ada program imports the C function using a matching function signature. By telling the compiler `Convention => C`, it will know that the string is passed by a pointer and will prepare the arguments accordingly. To ensure that both Ada and C use compatible function signatures the Ada code uses C types defined in the `Interfaces.C` package. The assertion checks that both strings are equal after the call (i.e., that they have the same contents). The assertion check should never equate to false, as both strings are equal on creation and constant.

But what happens when `Print_String` is modified to change the string it writes to the console? If it was implemented in Ada the new declaration would be:

```
procedure Print_String (S
:  in out char_array; Len : size_t );
```

With this change, the compiler will complain when `Some_String` is passed as it cannot be modified due to being a constant. However, this function is implemented in C, likely in a separate library. Its new implementation looks as follows

```
void print_string (char *s, size_t len ){
    for(size_t i = 0; i < len, i ++){
        putchar(s[ i ]);
    }
}
```

If the change is not identified during development, the Ada compiler will continue to assume that the string is not modified and base its checks on that assumption. Identification of the C code change is manual and error-prone; the Ada compiler cannot distinguish between the different C code bases as it does not parse and incorporate the C code. By design, the linker, responsible for connecting the compiled Ada and C code, does not understand types and calling conventions and will, therefore, also miss this inconsistency. The result is a program that does not behave according to the programmer’s intention, even though it should be according to the code and compiler. Furthermore, if the implementation is erroneous, the C function may overflow the buffer provided, corrupting the caller’s stack.

While CHERI does not solve the problem statically, it can introduce runtime checks beyond what the compiler can typically do. The caller of a function creates capabilities matching the permissions required by the arguments the caller is ex-pecting to call the callee with. In this example, the string is passed as a capability using the bounds of the string without write permission. The modified or erroneous C implementation may still try to violate these permissions. However, this now results in a CHERI exception. This approach can be employed for all data passed by reference. It does not cause the erroneous program to fail to compile, but it allows the problematic condition to be detected early through verification testing. Furthermore, it will enable the program to abort in a defined state or even recover from the error. As with many other improvements CHERI enables, this feature requires thoroughly applying the principle of least privilege in all pointers/capabilities passed to functions.

CHERI improves the situation; however, it does not solve the problem of foreign function interfaces in general. Calling conventions, type sizes, valid values, or even the number of arguments are

still unchecked by the compiler. It does, however, apply to the bounds and permissions of data passed by reference, avoiding hard-to-debug memory corruptions or broken assumptions about the immutability of passed data.

3.4 Hardware detected Capability Fault Propagation and Recovery

One feature of the Morello bare-metal Ada runtime is its ability to convert CHERI hardware exceptions into Ada exceptions that can be propagated, caught, and handled by application exception handlers. In traditional bare-metal runtimes, hardware exceptions are handled by a top-level trap handler which typically aborts the entire program. By contrast, the Ada Morello bare-metal runtime implements a trap handler that first determines which kind of CHERI exception has occurred, then returns control back to the call stack that triggered the trap but with the return address altered to call a subprogram that raises an Ada exception. This effectively causes an Ada exception to be raised from the point in the call stack that triggered the CHERI exception. When the Ada exception is raised, the runtime unwinds the call stack until it finds a handler for the exception, at which point control is passed to the handler. This is illustrated in ***Figure 4*** which shows the conversion and propagation of a CHERI exception across function calls.

This mechanism allows the software application to use conventional Ada exception handlers to detect, isolate, and recover from any CHERI exceptions that occur within that code block. In a multitasking environment, this also isolates the exception to the individual task that triggered it, allowing other tasks to continue execution unaffected. The ability to actively detect and respond to memory safety breaches allows the system to isolate compromised elements and initiate recovery procedures, enhancing fault-resilience. The affected system can fail in a "secure but degraded" manner, resulting in unaffected areas of the program being allowed to continue.

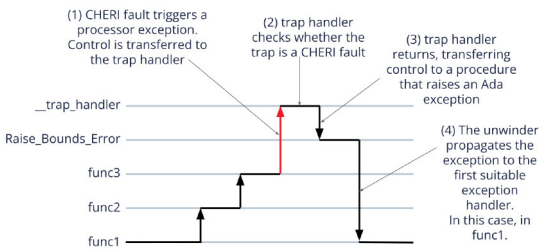

Figure 4: Converting a CHERI exception (raised in func3) to anAda exception and propagating it up the call stack to the handler in func1.

3.5 Reduction in Exploitability

CHERI significantly improves application memory safety by checking for violations of memory boundaries. Many exploit techniques require such a violation to work, be it by inserting code directly into the attacked application or modifying the application's state. These techniques typically require a good knowledge of the application's internal memory, especially the used address ranges.

In order to counter memory corruption-based attacks, many mitigation mechanisms have been created both on the system and the application level. In applications, memory protection is often improved by inserting runtime checks to detect over-flows and abort execution in case of detection. While this improves the application's security and often allows recovery from the error, it also incurs a performance penalty. Additionally, the protection is limited to the code written in that language. Other system parts, sometimes even the runtime required to execute the application, are not protected.

At the system level, memory protection consists of mechanisms that increase the difficulty of successfully executing an attack. This approach may not prevent memory corruption; however, it makes it harder to take control and manipulate the application behavior. Address Space Layout Randomization (ASLR) is a technique that assigns random parts of the address space to the application. While an attack may still overflow a buffer, it is much harder to guess the correct address for the overflow. For example, triggering the application to jump to a specific address requires knowledge of the address space.

Another approach is to restrict the privileges of different memory regions. More specifically, the Write XOR Execute (WˆX) principle. The assumption is that code is never modified by the program in its regular state and that modifiable memory, such as the stack and heap, are never used to execute code. While this also does not prevent memory corruption, it prevents placing and executing the attack payload. Even if the payload can be placed through memory corruption, it will likely be placed on the stack or in the heap where it cannot be executed. It also cannot be moved into executable memory as this region is not writable.

A technique to defeat this restriction is Return Oriented Programming. It uses the fact that some relevant metadata for the program execution is still placed on the stack and, therefore, in modifiable memory. This includes the return address, which controls the program's execution when the current function returns. An exploit using this technique will overwrite the stack, especially the return addresses, with values that cause the CPU to jump to parts of the code that contain the functionality needed by the attacker. These parts, often called gadgets, can be only parts of functions. They can be chained together by writing multiple return addresses into the stack, and after a jump into a gadget has happened, the next jump will be executed once the gadget tries to return.

We have analyzed CHERI's resistance against Return Oriented Programming. While this approach still often includes an initial memory corruption, we assume that the attacker is able to manipulate the stack once at the beginning of the attack. While CHERI would typically prevent this initial condition from happening we wanted to know whether Return Oriented Programming could leverage an initial vulnerability to further exploit the system.

At first, we validated the WˆX property of CHERI by creating examples that violate this principle while otherwise keeping all capabilities valid. Our tests have been executed and validated on an unprotected AArch64 platform. CHERI successfully rejected executing the stack by raising a capability permission fault. It also detected the code modification and raised a capability-sealed fault. A capability sealed fault is raised if a sealed capability is either dereferenced or has been modified before use. Sealed capabilities are used for capabilities referencing code. They cannot be modified or dereferenced but can be used as a jump target for the program counter.

Furthermore, we created two test cases to modify jump targets. The first test executed a common approach in ROP by overflowing a buffer on the stack and thereby overwriting the return address of the calling function. As all examples are written in Ada, the language caught that attack with a runtime check. Disabling runtime checks made the attack work on an AArch64 target. As expected on CHERI, this attack caused a capability-bound fault as soon as the program tried to write outside the allocated buffer on the stack.

The second test tried to manipulate the control flow more directly. It consists of a routine that takes a function pointer, adds an offset and calls the resulting function pointer. Adding an offset of zero should yield the same result as calling it without an offset while adding an offset greater than zero will cause the call to jump somewhere into the middle of the function or behind it. We generated the base function pointer from an existing function to ensure we start from a valid capability. On a non-CHERI target, this worked even with runtime checks enabled. Our CHERI solution detected the violation and raised a capability tag fault. While the created capability was valid, modifying it with an offset, even if it was zero, invalidated it. The reason is that CHERI uses sealed capabilities to represent function pointers, and sealed capabilities are immutable.

For an attacker that has access to all the information about the program but cannot modify its code directly, we conclude that a successful Return Oriented Programming attack is very unlikely, if not impossible. Even if many assumptions about CHERI, such as the bounds checks for capabilities, are invalidated, it still has additional layers of defense that prevent the unintended execution of the program's code. Even with the ability to manipulate the stack without triggering an exception, an attacker must replace the return address with a valid sealed capability.

Generally we notice that CHERI provides more than just resistance against memory corruptions. It also restricts the control flow into a narrow path, preventing deviations from the programmer's intended functionality. Even if some of CHERI's fundamental properties are violated, the remaining constraints still prevent or at least increase the difficulty of an effective attack.

3.6 Performance

The Edge Avionics project aims to produce a demonstrator Avionics system that showcases the security benefits of CHERI. The final prototype is not intended for flight. Therefore, Arm's Morello development board is a good choice for the final target demonstrator platform. The Morello board microprocessor is a CHERI-enabled prototype CPU based on Arm's Neoverse N1, as found in the N1SDP evaluation board. As stated by Arm, "This is a high-performance superscalar, out-of-order pipeline design. The existing 64-bit Armv8.2-A support in the CPU was retained and support for the new Morello architecture was added"[24]. Performance metrics are essential to understanding the feasibility and impact of adopting a CHERI microprocessor architecture, particularly for high-integrity and safety/security-critical avionics. Our research through the Edge Avionics programme will contribute towards the final demonstrator platform and should the work be taken into commercial production, the Morello development board would not be selected as the final target hardware. An extensive performance analysis is only needed for the final target platform intended for flight, which is outside of the project's scope. CHERI microprocessors have yet to achieve airworthiness certification at the time of writing. However, the University of Cambridge Computer Laboratory has already completed extensive performance studies around the prototype Morello microarchitecture, and the results provide a strong argument that future, commercially available and fit-for-flight CHERI microprocessor solutions will be able to cope with modern-day demands of defence-related avionics. Cambridge writes, "results to date give us strong confidence that CHERI support can be tightly and cleanly integrated into future Arm architectures"[25]. The report stipulates that "the dynamic performance aims for Morello were to create a hardware design able to enable the evaluation of the usage of capabilities within rich established software ecosystems and to demonstrate their practical viability and security benefit[25]". Therefore, whilst the performance was a factor in the design of the Morello CPU, it is expected there is room for significant optimizations and that second and third-generation CHERI microprocessor architectures and subsequent hard-ware implementations will be higher performing in terms of execution speed, memory footprint and energy consumption. Cambridge backs up this claim by stating: "It is reasonable to project that the goal of 2%-3% overhead for deterministic spatial and referential memory safety is achievable with an optimized instruction-set architecture on a performance-optimized microarchitecture[25]]". The performance penalty predicted with future CHERI microprocessor architectures and subsequent compiler designs is acceptable, adding to the viability of the Edge Avionics project.

4. Airworthiness Security Methods and Considerations

Airworthiness is the discipline of ensuring air vehicles are safe. Airworthiness Security forms part of that same discipline, focusing on security aspects that, should they fail, would lead to safety hazards. More specifically, a security case is argued that claims system security risks do not lead to unacceptable safety risks. Regulatory organizations like the Federal Aviation Administration (FAA) in the U.S. and the European Union Aviation Safety Agency (EASA) have circulated advisories around the need to detect and prevent unauthorized electronic interactions within air vehicles, primarily to ensure existing and future air vehicles remain safe. Satisfying advisory circular requirements around unauthorized electronic interactions is required before the awarding of airworthiness certifications. At the time of writing, the FAA or EASA are not mandating a particular solution; however, industrial consortium working groups within the European Organisation for Civil Aviation Equipment (EUROCAE) and RTCA (previously known as the Radio Technical Commission for Aeronautics) have put considerable effort into two jointly developed sets of Airworthiness Security publications. Our research has focused on how the described "Security by Default" approach can help meet the objectives stated within the EUROCAE and RTCA "Airworthiness Security Methods and Considerations".

The European Organisation for Civil Aviation Equipment (EUROCAE) ED-203A "Airworthiness Security Methods and Considerations"[26] foreword states that ED-203A is technically identical to RTCA DO-356A [27], and this is also true of ED-202A[2] and RTCA DO-326A[1]. Furthermore, these standards and guidelines are equally applicable to the defense industry as well as the civilian aerospace industry. For example, first published in May 2023, the UK Ministry of Defence (MOD) announced new regulations for the Military Aviation Authority (MAA)[28]. The report includes reference to Regulatory Articles (RA) 5890 [29], which states: "The MAA recognises the risk assessment and mitigation process detailed in RTCA DO-326A / EUROCAE ED-202A and associated standards RTCA DO-356A / EUROCAE ED-203A as an acceptable means of compliance."

Our work has indicated three promising areas where our pro-posed solution to "Security by Default" can help satisfy Air-worthiness Security objectives: Security Measures, Vulnerability logging and Refutation testing.

4.1 DO-326A/ED-202A Security Measure

To produce a convincing argument over the safe management of avionics security risk, we must show evidence that identifies all threat conditions and scenarios that could lead to loss of privacy, integrity or availability of identified security assets. Second, all attack paths must be understood and addressed. Applying risk treatment to an attack path amounts to allocating and assessing the effectiveness of one or more security measures and their ability to satisfy allocated security requirements. Where an attack vector involves memory safety vulnerabilities like a buffer overflow, we can argue a CHERI microprocessor architecture is a security measure that can reduce or stop the damage caused by the attack. Suppose the attack intends to expose a security asset within the system, i.e., violate a security requirement regarding asset privacy. In that case, correctly using CHERI's finegrain memory protection will result in a high-assurance security measure; the attacker may be able to trigger an exploit, but the hardware trap will detect the violation and guard against unauthorized memory reads/writes. The same feature provides a security measure that enforces the security asset's integrity; by bounding a memory address we ensure neighboring data is not overwritten and corrupted. In addition, whilst the security measure must still detect the event even if the attack only intends to cause disruption or loss of availability (for example, a denial of service attack), it must also satisfy security requirements that minimize or eradicate the loss of service, for example, recovery, isolation, or damage limitation. Our proposition described within this paper argues that the

propagation of CHERI capability faults into Ada runtime exception handlers provides detection and countermeasure options to respond to the loss of service attack and, therefore, acts as an additional high assurance security measure.

4.2 Security Verification Objectives

The aim of refutation in the context of the Airworthiness Security Process is to refute the allegation of exploitable vulnerabilities[30]. The Airworthiness Security Process[2] [1] describes refutation as: "an independent set of assurance activities beyond analysis and requirements. As an alternative to exhaustive testing, refutation can be used to provide evidence that an unwanted behavior has been precluded to an acceptable level of confidence. NOTE: Refutation is also known as Security Evaluation in some contexts[26] [27]." The refutation activities aim to identify any unexpected situations where the system would unexpectedly transition into a non-secure state (or, more generally, violate a minimal invariant guaranteeing the system's security)[30]. The difficulty with refutation testing is in the consistent and repeatable detection of the transition. Consistency and repeatability make it feasible to argue for an elevation in security assurance and, for Airworthiness Security, this must be adequately described within the Plan for Security Aspects of Certification (PsecAC). The PsecAC is the initial phase within the Airworthiness Security Process, and it is here that we set our security goals and how we intend to security test our application. Much like a "Plan for Safety Aspects of Certification" within the parent process "Software Considerations in Airborne Systems and Equipment Certification" (ED-12C and DO-178C [31]), integrators need to ensure regulatory authority accepts the plan before commencing with development and test phases.

Our approach enforces anomaly detection through regular Ada runtime constraint checks[20] and through the developed CHERI-hardware-enforced pure-capability Ada runtime. This dynamic verification feature captures unsafe memory instructions that would otherwise result in memory violations, such as out-of-bounds reads/writes. Not only can we isolate security assets in deployed systems, but we greatly enhance security verification testing as more anomalies can be detected. To understand why this is important, consider the resultant behavior of a standard (non-CHERI) CPU executing an application not using Ada runtime constraint checks. When a triggered software bug results in an out-of-bounds memory read or write instruction, the system could exhibit behavior that can be detected, for example, a segmentation fault may get signaled; however, it could equally go undetected such that the system continues to operate but also transitions into a state where the security can no longer be guaranteed. The combination of an Ada pure-capability runtime executing on a CHERI microprocessor architecture eliminates this possibility; all out-of-bounds reads/writes will be captured by either the Ada runtime or the CHERI hardware. In both cases, the transition into a non-secure state will be visible to the verification suite so that the bug can be identified, logged and mitigated at a higher level in the safety plan or fixed and retested. It is also important to recognise the symbiosis of the pure-capability Ada runtime and the CHERI hardware capability checks; without the combination the guaranteed detection is lost and the quality of the refutation degraded.

The Airworthiness Security Process Guidelines in ED-203A and DO-326A[26] [1] state that refutation encompasses multiple disciplines, including "Dynamic Code Analysis". This specific refutation testing technique analyzes the system's behavior whilst the system is executing. An example of dynamic code analysis would be monitoring a non-safe or non-secure sequence of instruction calls made to the processor (i.e., detection of buffer overflows). Dynamic code analysis can be enforced within the semantics of programming languages via run time constraint checks or tools that detect memory corruption bugs via code instrumentation added during additional compilation passes[30]. Our research has shown that existing memory detection tools like Address Sanitizer[23] and Valgrind[22] can't detect the complete set of memory violations that our approach can ***(see section 3.4)***. In both cases,

anomalies can only be detected when a test case triggers the scenario that exhibits the unwanted behavior. However, our research has uncovered sequences where Address Sanitizer and Valgrind fail to detect the transitions that our solution captures. Vulnerability identification is a critical aspect of any security process. As it is widely recognised that non-safe memory instruction calls form the basis of the majority of exploitable software bugs, being able to dynamically and consistently detect and guard against memory violations provides a security safety net should all other measures fail.

4.3 Vulnerability Logging / Fault-Recovery / Fail-Secure

In addition, our "Security by Default" research argues that propagating CHERI hardware detected faults into Ada soft-ware handlers makes it feasible to isolate system components such that fail-secure-but-degraded is possible. Without this feature, CHERI-based systems can still protect security assets. However, fault-recovery is only possible through intervention from a third-party monitoring system, such as a hypervisor. However, the main difference between this approach and the one proposed is that by dynamically detecting the impending violation at the point just before the failure condition, the state of the system, the triggering conditions and any other relevant information can be recorded within a security log file. Regulatory Article 1202 describes a framework approach for In-Service Air System Cyber Compliance. It is noted that this method is based on the requirements of the US National Institute of Standards and Technology (NIST) Cybersecurity Framework, which advocates the phases of "Identify", "Protect", "Detect", "Respond" and "Recover", note that the National Cyber Security Centre (NCSC) also provides a Cyber Assessment Framework (CAF) that shares the principles of the NIST Framework. Two aspects of this methodology where the proposed solution plays a role are "Detect" and "Respond". "Detect" is described as being "introduced to enable timely detection of cyber security Incidents that may impact Air Safety, such as continuous monitoring and security log files."[29] Therefore, capturing and isolating attacks is essential to satisfying the Detection requirements. In addition, the "Respond" phase is described as "once a cyber incident affecting Air Safety has occurred, the level of response is key in supporting the ability to contain the impact." This requirement is aimed at "business continuity plans" and "associated response plans" and having the ability to detect, capture, isolate and report the attack directly within the affected system is clearly beneficial.

4.4 Software Supply Chain Security

Modern-day large-scale systems often require collaborative efforts spanning large geographical regions that exacerbate the complexity around software supply chain security. The software supply chain is made up of all aspects of software development across all phases of the software development lifecycle. This includes development tools that have direct access to the source code and pose a risk to security assurance. A compiler's primary responsibility is to translate source code into machine code. Assuring that this translation is correct amounts to traceability studies that include binary-to-source code analysis[32]. Our work included porting a developed CHERI pure capability Ada runtime with different compiler back-ends, namely GCC[17] and LLVM[18]. Having more than one compiler solution is beneficial as it allows for novel software supply chain security verification techniques, like differential testing. Here, we argue that the integrity of the development tool is maintained by comparing it to the behavior of the alternative simply by feeding the same inputs into both, verifying the output, and observing the state. Voting algorithms are frequently used in high-integrity systems to increase the assurance of processed data. For example, flight control systems may sample data from multiple sensors and use algorithms to check the consistency and decide which value to use. The same argument can be applied to the security assurance of software development tools. However, this approach

requires multiple independent solutions that, whilst the sampled data will be identical (i.e. the source code, in the case of a compiler), perform the same functionality and generate output that satisfies the translation requirements (the generated CPU instructions perform the functionality de-fined by the source code) with differing algorithmic designs. Developing both GCC[17] and LLVM[18] Ada Morello bare-metal compilers allows this argument to be made.

5. Further work

To further extend the approach described, we propose developing additional software runtime components that enhance the capabilities of CHERI hardware extensions. Security assurance can be further elevated by integrating features such as Temporal Memory Protection and Compartmentalization. Beyond spatial memory protection, software can extend CHERI's capabilities to include temporal memory protection. Temporal Memory Protection helps prevent vulnerabilities like use-after-free errors, which is achieved through careful memory management and the use of capabilities to track and control memory lifecycles. Examples include tools like CHERIvoke[33] and Cornucopia[34] that characterize pointer revocation using CHERI Capabilities for Temporal Memory Safety. Compartmentalization is concerned with adding protection around untrusted libraries such that separate heap allocations are used and compartmentalized code can only access code or data in another compartment through a well-defined interface. Examples include CHERIoT[35]. In addition, support could be added for Ada. Tags.Internal_Tag in Morello GNAT which would remove the limitation on streaming Ada tagged types. Finally, future work will focus on the latest microprocessor architectures supporting CHERI. While the research conducted within this paper used Arm's Morello platform, the next phase of work will likely be on a CHERI-RISC-V CPU.

6. Conclusions

The paper summarizes research and development into a "Security by Default" approach to real-time embedded systems by leveraging the Arm Morello CHERI ISA extensions and a bare-metal security-enhanced Ada runtime. More specifically, a layered approach to security is described that demonstrates the benefits of memory-safe programming languages executing on memory-safe microprocessors. This combination allows Ada developers to benefit from an enhanced security toolchain and execution environment for high-integrity real-time systems. In addition, the paper proposes a fault resilience approach to bare-metal software security design by propagating CHERI hardware capability bounds exceptions into bare-metal application code exception handlers. Furthermore, our experience with CHERI has shown that it is an excellent verification target due to the advanced anomaly detection features of hardware capabilities and that porting Ada code to CHERI is often no effort. In addition, had it not been for this work, a security vulnerability could have made its way into deployed software, and our continuous integration suite now benefits from executing our Ada runtime regression tests on CHERI. Our work included analyzing the benefits of a CHERI pure-capability runtime and a CHERI-compliant microprocessor to airworthiness certification. As described in ***section 4***, our developed solution could satisfy multiple security objectives; more specifically, it can be used as a deployed security measure guarding against high-security assurance level vulnerabilities and a dynamic analysis security verification tool for refutation testing. The results and insights presented in this research open additional avenues for strengthening the security of embedded real-time systems, ultimately contributing to safer, more reliable and more secure technology.

References

[1] RTCA, "DO-326A - Airworthiness Security Process Specification," 12 2014. for the U.S. Federal Aviation Administration FAA.

[2] EUROCAE, "ED-202A - Airworthiness Security Process Specification," 6 2014. for the European Union Aviation Safety Agency (EASA).

[3] National Cyber Security Centre, "Secure by Default," NCSC publications, 2018. https://www.ncsc.gov.uk/information/secure-default [Accessed: (31/10/2023)].

[4] Arm®, "Arm® Architecture Reference Manual Supplement Morello for A-profile Architecture," Arm® *Morello publications*, vol. A.k, 2022. https://de veloper.arm.com/documentation/ddi0606/latest [Accessed: (31/10/2023)].

[5] National Security Agency, "Cybersecurity Information Sheet - Software Memory Safety," *National Security Agency Publications*, 2022. https://media.de fense.gov/2022/Nov/10/2003112742/-1 /-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF [Accessed: (02/11/2023)].

[6] Z. Ling, H. Yan, X. Shao, J. Luo, Y. Xu, B. Pearson, and X. Fu, "Secure boot, trusted boot and remote attes-tation for ARM TrustZone-based IoT Nodes," *Journal of Systems Architecture*, vol. 119, p. 102240, 2021.

[7] Frédéric Pothon, Quentin Ochem, "AdaCore Technolo-gies for DO-178C / ED-12C," *AdaCore publications*, 2017. https://www.adacore.com/uploads/books/pdf/AdaCoreTechnologiesForDO178C-web.pdf [Accessed: (31/10/2023)].

[8] AdaCore, "White Paper - Disruptive Technology for Military-Grade Software," *AdaCore publications*, 2021. https://www.adacore.com/uploads/techPapers/Developing-Military-Grade-Software.pdf [Accessed: (31/10/2023)].

[9] Jean-Louis Boulanger, Quentin Ochem, "AdaCore Tech-nologies for CENELEC EN 50128:2011," *AdaCore publications*, 2018. https://www.adacore.com/books/cenelec-en-50128-2011f [Accessed:(31/10/2023)].

[10] Benjamin M. Brosgol, Jean-Paul Blanquart, "AdaCore Technologies for Space Systems Software Supporting Qualification for ECSS-E-ST-40C and ECSS-Q-ST-80C," *AdaCore publications*, 2021. https://www.adacore.com/uploads/books/pdf/Ad aCore-Tech-Space-Systems.pdf [Accessed:(31/10/2023)].

[11] AdaCore, "AdaCore / bb-runtimes (Public)," 2024. https://github.com/AdaCore/bb-runtimes [Accessed: (03/11/2024)].

[12] Robert N. M. Watson, Simon W. Moore, Peter Sewell, Peter G. Neumann, "An Introduction to CHERI," *UCAM-CL-TR-941, ISSN 1476-2986*, 2019. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf.

[13] Defence Science and Technology Laboratory, "About Us," 2024. https://www.gov.ukgovernmentorganisations/defencescience-and-technology-laboratory/about [Accessed: (03/11/2024)].

[14] GE Aerospace, "About Us," 2024. https://www.geaerospace.com/ [Accessed: (03/11/2024)].

[15] Wind River, "Wind River: About Us," 2024. https://www.windriver.com/contact [Accessed:(03/11/2024)].

[16] AdaCore, "About Us," 2024. https://www.adacore.com/companyabout [Accessed:(03/11/2024)].

[17] Richard M. Stallman et al., "Using the GNU Compiler Collection," 2012. https://https://gcc.gnu.org onlinedocs/gcc-4.4.2/gcc/

[18] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *CGO*, (San Jose, CA, USA), pp. 75–88, Mar 2004.

[19] Nicolas Joly, Saif ElSherei, Saar Amar – Microsoft Security Response Center (MSRC), "Security analysis of CHERI ISA," *MSRC-Security-Research/papers/2020*,2020. https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Security%20analysis%20of%20CHERI%20ISA.pdf.

[20] AdaCore, "Ada Conformity Assessment Authority: Ada Reference Manual," 2024. http://ada-auth.org/arm.html [Accessed: (03/11/2024)].

[21] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert Norton, Thomas Bauereiss, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W. Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, Simon W. Moore, "CHERI Concentrate: Practical Compressed Capabilities,"*IEEE Transactions on Computers 68(10),* 2019. https://doi.org/10.1109/TC.2019.2914037,https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/2019tc-chericoncentrate.pdf.

[22] "Valgrind," 2023. https://valgrind.org/ [Accessed: (03/20/2024)].

[23] Clang 19.0.0git documentation, "AddressSanitizer," 2023. https://clang.llvm.org/docs/AddressSanitizer.html [Accessed: (03/20/2024)].

[24] Arm, "Creating the Morello Technology Demonstrator,"2022. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/creating-the-morello-technology-demonstrator.

[25] 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," September 2023. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-986.pdf.

[26] EUROCAE, "ED-203A - Airworthiness Security Meth-ods and Considerations," 6 2018. for the European Union Aviation Safety Agency (EASA).

[27] RTCA, "DO-356A - Airworthiness Security Methods and Considerations," 9 2014. for the U.S. Federal Avia-tion Administration FAA.

[28] UK Ministry of Defence, "Cyber security for airworthi-ness: new MAA regulations," 2023. https://www.gov.uk/government/news/cyber-security-for-airworthiness-new-maa-regulations/ [Accessed: (03/20/2024)].

[29] UK Gov, "RA 5890 – Cyber Security for Airworthiness and Air Safety – Type Design and Changes / Repairs to Type Design," 2023. https://assets.publishing.service.gov.uk/media/656866f2cc1ec500138eef7b/RA5890_Issue_2.pdf [Accessed: (03/20/2024)].

[30] Paul Butcher, "Guidelines and Considerations Around ED-203A / DO-356A Security Refutation Objectives," AdaCore Papers, 2021. https://www.adacore.com/uploads/techPapers/Guidelines-a round-ED203A-and-DO356A-Security-Refutation-Objectives.pdf.

[31] RTCA/EUROCAE,"Software Considerations in Airborne Systems and Equipment Certification, DO178C/ED-12C," 2011.

[32] AdaCore, "Source Code to Object Code Traceability Study," 2016. https://www.adacore.com/up loads/books/pdf/traceability-sample. pdf.

[33] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richard-son, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, Timothy M. Jones, "CHERIvoke: Characterising Pointer Revocation using CHERI Capa-bilities for Temporal Memory Safety," in In Proceed-ings of the 52nd IEEE/ACM International Symposium on Microarchitecture (IEEE MICRO 2019), pp. 12–16, 2019. https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201910microcheritemporal-safety.pdf [Accessed:(31/10/2023)].

[34] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, Robert N. M. Watson, "Cornucopia: Temporal Safety for CHERI Heaps," in *In Proceedings of the 41st IEEE Symposium on Security and Privacy* (Oakland 2020), pp. 18–20, 2020. https://www.ncsc.gov.uk/information/secure-default [Accessed: (31/10/2023)].

[35] Amar, Saar and Chisnall, David and Chen, Tony and Wesley, Nathaniel Filardo and Laurie, Ben and Liu, Kunyan and Norton, Robert and Moore, Simon W. and Tao, Yucong and Watson, Robert N. M. and Xia, Hongyan, "CHERIoT: Complete Memory Safety for Embedded Devices," *in proceedings of the 56th IEEE/ACM International Symposium on Microarchitecture, Association, for Computing Machinery*, Oct. 2023.