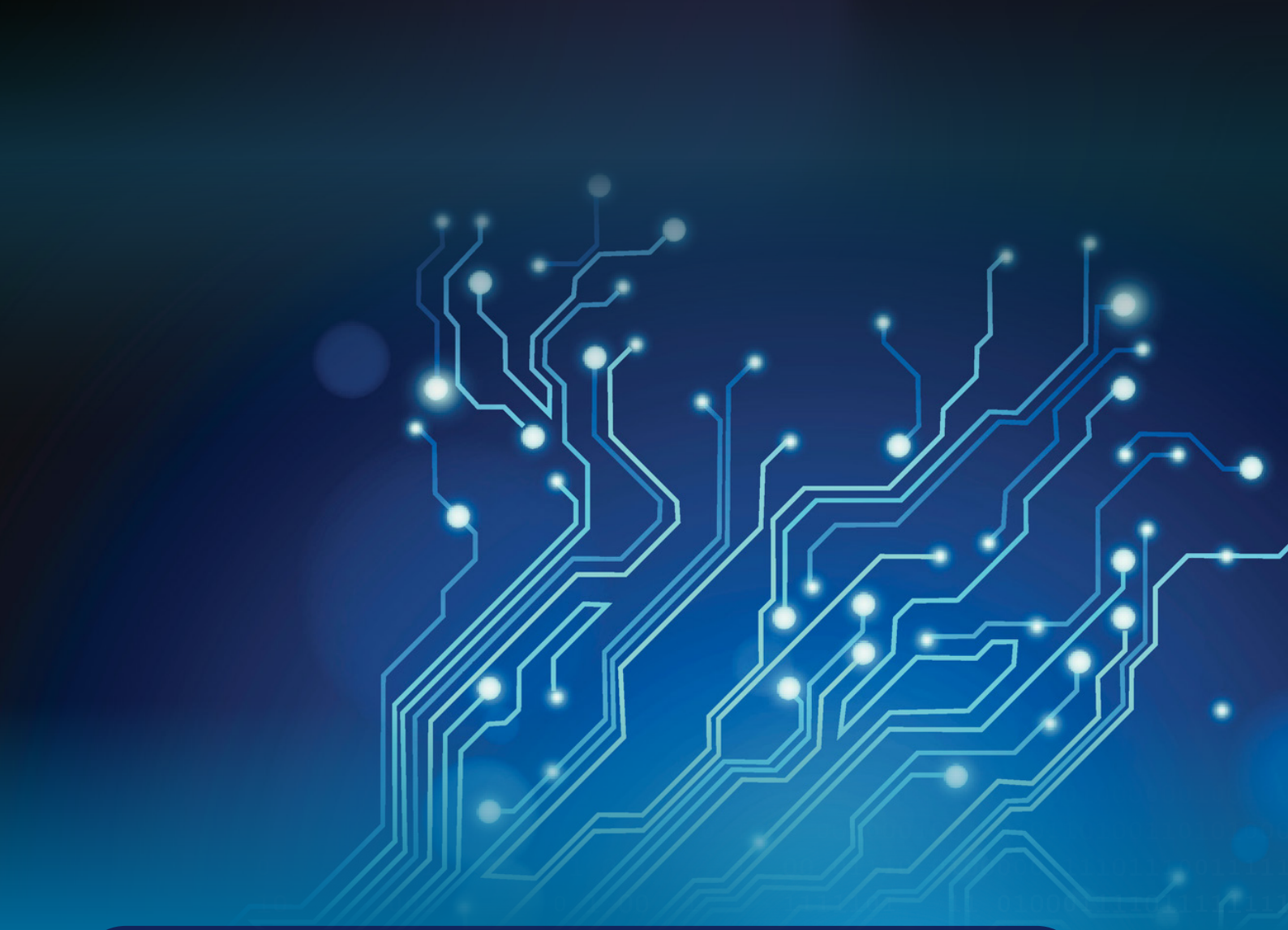

AdaCore | Build Software that Matters

Tech Paper

Secure by Design Principles – Fuzzing in a Memory Safe Verification Environment

CORENTIN MACHU[1], PAUL BUTCHER[2], DANIEL KING[3], KYRIAKOS GEORGIOU[4]

[1] AdaCore SAS, Paris, ˆIle-de-France, 75009, France
[2,3,4] AdaCore Ltd, Bristol, BS1 6PU, United-Kingdom

adacore.com

Abstract

In today's interconnected world, the security of software systems is paramount. The increasing frequency and severity of cyberattacks underscore the need for a fundamental shift in software development [31]. The "Secure by Design" paradigm is a proactive strategy that advocates integrating security considerations throughout the software development life cycle rather than treating them as an afterthought. This paper introduces an innovative approach that leverages memory-safe hardware, a processor enhanced with Capability Hardware Enhanced RISC Instructions (CHERI) [37], and fuzz testing. The approach adheres to the Secure by Design principles by enhancing security verification in embedded real-time systems at early stages of the development life cycle. Experimental results show that the solution is beneficial for detecting software memory issues in both memory-safe and unsafe programming languages. We further demonstrate that the approach detects classes of memory errors that are not reliably captured by conventional fuzzing combined with address sanitizers, by converting otherwise silent memory violations into deterministic hardware-detected faults.

1. Introduction

In 2019, Microsoft conducted a critical evaluation of its software vulnerability mitigation landscape over the past 10 years, concluding that memory safety issues have been the dominant source of Common Vulnerabilities and Exposures (CVEs) [24]. Their four most common reported vulnerabilities were heap out-of-bounds reads, use-after-free, type confusion, and uninitialized use.

Six years later, and the 2025 Common Weakness Enumeration (CWE) Top 25 Most Dangerous Software Weaknesses list still includes some of the same memory vulnerabilities that Microsoft identified in its 2019 study, with out-of-bounds read and use-after-free ranking in the top 10 in terms of severity and impact [26]. Furthermore, the same list includes other memory-related vulnerabilities such as out-of-bounds writes and null pointer dereferences. For reference, the Top 25 CWE list is compiled annually by the MITRE Corporation from a well-established, community-developed catalog of software and hardware weakness types that can compromise the resilience, reliability, and integrity of software. This makes a strong case that memory-related vulnerabilities are still the Trojan horse of today's software security.

The core source of memory-related vulnerabilities is the use of memory-unsafe programming languages. Historically, C and C++ have been some of the most popular programming languages, mainly due to their performance benefits and large ecosystems. Some of the most widely used software applications are written in these languages, including the Windows and Linux kernels, which are primarily written in C. What makes such languages memory-unsafe is their memory handling paradigm. In a nutshell, memory locations are designated by memory addresses, which are represented as integer values at the programming level. Any piece of code running in a process that can create an integer value that is a valid memory address within its process's address space can access that memory location. This memory handling mechanism constitutes the most exploitable attack vector for software.

2. This work is funded by the Safety Critical Harsh Environment Micro-processing Evolution (SCHEME) project. The £37.5m investment program is co-funded by the ATI Programme, which funds civil aerospace research in the UK and which is delivered in partnership by the Aerospace Technology Institute, the Department for Business & Trade and Innovate UK. Rolls-Royce is joined by AdaCore, TT Electronics, Volant Autonomy, Rapita Systems, The Manufacturing Technology Centre, Queen's University Belfast, University of Bristol, University of Sheffield, and University of York.

This work has been published at: https://hal.science/ERTS2026/hal-05513731v1

One might argue that the solution is to port existing C and C++ codebases to memory-safe languages like Rust and Ada/SPARK. Indeed, this is the trend for many large organizations in recent years. Google's Android security team reports that moving new development from C/C++ to memory-safe languages such as Rust reduced the percentage of Android vulnerabilities caused by memory safety issues from 76%to 24% over six years [19]. Major cloud providers such as AWS have also standardized on Rust for infrastructure software, such as the Firecracker micro-VM that underpins AWS Lambda, citing Rust's memory-safety guarantees as a key factor in their design [9, 10]. In 2018, NVIDIA evaluated the use of the SPARK memory-safety language and formal veri-ication toolkit to improve software assurance, primarily for automotive applications [3, 27]. Despite the initial investment in training and tool adoption, the assessment concluded that the resulting gains in security and verification efficiency justified the transition. NVIDIA reported significantly improved robustness in SPARK-based implementations compared to earlier versions developed in memory-unsafe languages [2].

However, despite the recent increase in success stories related to the adoption of memory-safe languages hitting the headlines, these cases represent only a small fraction of the global software landscape. The reality is that migrating the vast number of codebases written in memory-unsafe languages over decades remains prohibitively costly and time-consuming, and retraining engineering teams to adopt new memory-safe paradigms is itself a substantial organizational challenge. Moreover, even when memory-safe languages are adopted, developers may still introduce memory-safety vulnerabilities through memory-unsafe language constructs, such as Ada's *Unchecked Conversion* [1], which bypass the language's safety guarantees. As a result, memory-safe languages alone cannot eliminate the need for complementary verification mechanisms. Thus, software testing techniques, such as fuzz testing, are still required as a preemptive measure to detect and fix CVEs before they are exploited.

While testing is vital to software development and continuous integration, several classes of memory-safety violations can be triggered during testing, but remain undetected because they do not immediately cause an observable program failure; they silently overwrite adjacent memory, corrupt metadata, or cause delayed crashes far from the root cause. A typical example of this is a silent buffer overflow, where the program writes data past the bounds of the intended memory buffer, without triggering a visible crash, assertion failure, or any other observable misbehavior. However, the undetected memory issue constitutes a potentially exploitable security vulnerability.

Address sanitizers, such as LLVM's *AddressSanitizer* (ASan) [20], instrument a program at compile time to detect classes of memory errors that often remain undetected during standard testing, even when they are triggered. They achieve this by surrounding objects in memory with specially marked guard regions and replacing standard memory allocation routines with checked variants. As a result, issues such as out-of-bounds accesses and use-after-free are more likely to cause immediate, deterministic failures rather than silent memory corruption; however, typical address sanitizers cannot guarantee detection in all cases (see example in Section 4). Grey-box fuzzers, such as AFL++ [17] and libFuzzer [21], complement address sanitizers by generating large numbers of mutated inputs and using code coverage feedback to explore program paths and reveal hard-to-reach corner cases. The combination of sanitisation and gray-box fuzzing is therefore one of the most effective dynamic verification strategies for exposing otherwise silent memory-safety vulnerabilities, substantially improving the observability of faults during testing.

One promising solution to memory-safety issues of memory-unsafe languages, such as C and C++, is the CHERI project [35]. The solution revisits the traditional hardware/software stack from the ground up to improve Security by Design (SbD). It introduces intra-process memory isolation capabilities, a fundamental change to most mainstream architectures. The idea is to replace

the use of integer addresses for accessing intra-process memory with an architecture-based capability system. Such a system uses the notion of capabilities to control access to system resources; in our case, the resource is the processor's memory, and the capabilities enforce various checks and controls on memory usage. Therefore, in a pure-capability CHERI-enhanced architecture, load, store, and instruction-fetch operations use capabilities to access memory locations rather than integer-based memory addresses. Thus, CHERI can be perceived as a built-in architecture address sanitizer that ensures no memory violations go undetected. CHERI also supports a hybrid mode in which CHERI capabilities can be selectively introduced, e.g., for specific modules, libraries, or data structures, while the rest of the program continues to use traditional pointers. In this paper, we always refer to the CHERI pure capability mode, where all pointers in the system are capabilities, including stack, heap, global, and function pointers.

This paper explores the potential of combining memory-safe architectures, such as CHERI, with fuzzing techniques. The argument is that fuzzing in a memory-safe hardware environment can be a more powerful technique in detecting memory-safe vulnerabilities at early stages of the software development lifecycle than existing state-of-the-art techniques, such as combining address sanitizers with fuzzers. Furthermore, combining CHERI with fuzzing could also be beneficial for programs written in memory-safe languages. This paper tries to validate these arguments through the following approach:

- Firstly, common memory corruption vulnerabilities, such as stack and heap buffer overflows, double frees, and use-after-free errors, have been identified and successfully re-produced in both C- and Ada-implemented functions.
- Then, fuzzing campaigns were conducted on these erroneous functions using a novel, cross-platform, in-house developed fuzzer named *Adaptive Engine*.
- All tests were compiled with GCC 15.2.1, the latest release available at the time of writing, and fuzzed with the following configurations:
  - x86 architecture, without ASan
  - x86 architecture, with ASan enabled
  - Morello CheriBSD in pure-capability mode (see Section 2.2)

The results indicate that, under specific circumstances, CHERI can detect vulnerabilities that are not readily found even with memory sanitizers. More specifically, ASan demonstrated a blind spot for a particular class of buffer overflow issues, which we classify as the *Red Zone Jump* pattern. *Red Zones* are memory regions allocated around heap and stack memory objects by ASan to detect whether memory operations cross their permitted boundaries. Any overflow that leaps over ASan's poisoned *red zones* and lands in valid neighboring allocations is not captured by ASan. These are not ASan implementation shortcomings but inherent tradeoffs in its red-zone mechanism design. Using the Adaptive Engine and CHERI, we were able to reliably capture this class of memory issues that ASan failed to do by turning previously silent spatial violations into deterministic CHERI-capability faults.

The paper's results show that CHERI capability outperforms existing ASan solutions in detecting memory-related vulnerabilities through fuzzing. Furthermore, the experimental results validated that CHERI and fuzzing could enhance the security of programs written in memory-safety languages by capturing all potential classes of issues related to the use of those languages' unsafe features.

While the CHERI architecture is still at a very early stage of adoption, with limited hardware availability, the results are encouraging and demonstrate that combining the CHERI capability with fuzzing can be a powerful tool for enabling Secure by Design principles and, therefore, significantly enhance software robustness and limit the attack surface.

The remainder of this paper is structured as follows. Section 2 introduces the Secure by Design principles, provides background on the CHERI architecture and fuzz testing, and presents the Adaptive Engine fuzzing framework. Section 3 describes the experimental methodology and evaluates the effectiveness of the proposed approach across multiple classes of memory-safety vulnerabilities and execution environments. Section 4 discusses the results, highlighting the strengths and limitations of hardware- and software-based memory-safety mechanisms. Finally, Section 5 concludes the paper and outlines directions for future work.

2. Background and Related Work

2.1. Introduction to Secure by Design principles and the concept of Secure by Verification

Secure by Design (SbD) is a systems development paradigm born out of the ever-increasing global demand for more advanced and interconnected technology and therefore ever-expanding attack surface [30]. SbD is not a prescriptive standard. Instead, it aims to bring security considerations to the forefront, ensuring they are considered at the earliest stages of the development lifecycle, throughout the lifecycle, and beyond, into deployment and decommission. The once much-relied-on legacy approach of identify, patch, redeploy, and repeat, coupled with perimeter defense, is now widely considered an unsustainable and insufficient approach against modern, adaptive threat landscapes. This fundamental vulnerability gap has driven governmental bodies and industry toward the rigorous adoption of the SbD mindset [34]. This section will cover the core tenets of SbD and their relationship with a Security by Verification (SbV) approach, which aims to prove security properties at the outset through rigorous techniques, such as refutation testing, as described in EURO-CAE ED-203A, [16], and its technically equivalent standard, RTCA DO-356A, [29].

In the context of software development, a key goal of SbD is to shift the focus of security to earlier stages of the software development lifecycle and make it a non-optional component [12].

While there is no de facto SbD standard, multiple frameworks and widespread government publications emphasize that the most common exploitable vulnerabilities that can cause significant harm are often associated with unsafe memory instruction calls (e.g., buffer overflows and underflows). A goal of SbD is to ensure that design strategies are employed to identify all potential vulnerabilities during development and to apply applicable security countermeasures in the deployed application. Promoted Mechanisms for memory safety vulnerability detection (pre- and post-deployment) include the early adoption of memory-safe programming languages (ref: https://www.cisa.gov/case-memory-safe-roadmaps) [40]. Ada, Rust, and SPARK are examples of memory-safe programming languages, and CHERI is an example of a memory-safe hardware architecture (See Section 2.2).

One prominent example of a response to a call for SbD is from the Office of the National Cyber Director (ONCD) of the United States government. The strategy promoted through "Back to the Building Blocks: A Path toward Secure and Measurable Software" [36] advocates for three primary decisions to reduce memory safety vulnerabilities: using memory-safe programming languages, employing memory-safe hardware architectures, and utilizing formal methods as complementary approaches to achieve similar outcomes. Other reports such as "The Case for Memory Safe Roadmaps", published collaboratively by agencies like the U.S. Cybersecurity and Infrastructure Security Agency (CISA), the National Security Agency (NSA), the Federal Bureau of Investigation (FBI), and their counterparts from Australia, Canada, New Zealand, and the United Kingdom, argue a unified transglobal approach on enforcing security via memory-safety [11].

While static and dynamic checks within memory-safe programming languages can answer many questions about memory safety, SbD principles are also being satisfied through advances in microprocessor architectures. The CHERI architecture provides a dynamic hardware framework

for detecting memory safety issues through the augmentation of common ISAs, with metadata added to pointers to enforce fine-grained compartmentalization [39]. CHERI is a dynamic analysis technique that uses hardware-enforced fine-grained memory protection for any programming language compiler that can implement CHERI pure capability memory allocators. Therefore, CHERI can provide security enforcement for legacy C/C++ codebases or other languages typically considered memory-unsafe. The effort required to port existing code bases to CHERI depends on the reliance on the hardware architecture; for example, a C application may make assumptions about pointer sizes that need to be changed to run with CHERI capabilities [39].

The UK Government is another example of a primary state sponsor of the research-to-commercialization pathway of CHERI adoption through the now-concluded Digital Security by Design (DSbD) programme [14]. The considerable investment made through DSbD underscores the importance of bringing SbD principles to the masses.

In addition, critical domains, such as civil aviation, consider security in terms of safety; security incidents that can lead to safety hazards must be treated with the appropriate level of rigor associated with the potentially triggering hazard. To enforce SbD principles within this specific domain, the industry set up the Airworthiness Security Process [15, 28]. One mechanism for implementing a security verification technique is refutation testing, which can be achieved via various forms of static or dynamic analysis. For a dynamic analysis approach, the critical factor to ensure the maximum number of vulnerabilities can be detected is the throughput of potential vulnerability triggering scenarios and the capability of anomaly detection. Within this paper, we argue that a Secure by Verification approach can satisfy this core dependency via fuzz-testing on a CHERI architecture.

2.2. Introduction to CHERI

The CHERI project was initiated in 2010 by SRI International [32] and the University of Cambridge in response to a DARPA funding call to rethink the traditional hardware/software stack from scratch to improve security [13]. Since then, the project has evolved significantly, with contributions from companies such as ARM, Google, and Microsoft, and further support from government funding programs, such as the UK's ISCF Digital Security by Design program [33]. This section provides an overview of CHERI.

CHERI is an extension of RISC instruction set architectures that enables fine-grained memory protection and compartmentalization, enforced at the hardware level. CHERI extensions have been defined for MIPS, Armv8-A, and 32-bit and 64-bit RISC-V architectures. Morello is Arm's CHERI prototype platform for Armv8-A.

CHERI mitigates common memory-safety vulnerabilities, such as buffer overflows and dangling pointers, by introducing architectural capabilities. CHERI capabilities extend conventional pointers with additional metadata to control how memory can be accessed:

- **a 1-bit tag:** determines the validity of the capability. Only valid capabilities can be dereferenced.
- **bounds:** limit the address range that can be accessed via the capability.
- **permissions:** prevent unintended use of capabilities, such as preventing code execution from a data-only buffer.
- **an object type field:** used for sealing, which makes a capability immutable and non-dereferencable.

This additional metadata increases the size of pointers on CHERI architectures. For example, on a 64-bit RISC-V platform, each capability is 129 bits. The 129th bit is the tag bit, stored separately to prevent it from being written directly by software.

All instructions that access memory (load, store, and branch instructions) require a valid capability with the correct bounds and permissions for the requested operation. If the capability does not permit the requested operation, or an invalid capability is provided, a hardware exception is raised to allow software to deterministically handle the violation.

Furthermore, the CHERI ISA limits how capabilities can be created and manipulated. In particular, bounds and permissions can only be narrowed, not broadened (attempting to do so results in an invalid capability), and valid capabilities can only be derived from other valid capabilities. This ensures that software can access only the memory it has been explicitly granted access to, following the principle of least privilege.

CheriBSD is an Operating System (OS) developed by the University of Cambridge and SRI International that extends FreeBSD to take advantage of Morello and CHERI RISC-V platforms, implementing fine-grained CHERI memory protection and software compartmentalization features. This paper uses CheriBSD on ARM's Morello as the execution platform for our experiments.

2.3. Introduction to Fuzzing

Fuzz-testing, also known as fuzzing, is a software testing technique developed in the late 1980s [25]. The technique involves providing an executable with random, invalid, unexpected, or malformed inputs to find potential security vulnerabilities, crashes, or other unexpected behaviors. The primary goal of fuzzing is to improve the security and reliability of the software by identifying and fixing any weaknesses that attackers could exploit. Over the last decade, fuzzing has become more effective due to advancements in tools and hardware, increased automation, better input generation, more sophisticated feedback mechanisms, integration into development workflows, such as Continuous Integration (CI) pipelines, community collaboration, and increased levels of cybersecurity threats. As a result, it has become an essential tool for improving software security and robustness, gaining widespread adoption across multiple industries and from an increasing number of organizations.

This paper focuses on gray-box fuzzers, such as AFL++ [17], libfuzzer [21] and our Adaptive Engine (introduced in Section 2.4), that leverage code-coverage feedback to explore program paths effectively. We will use AFL++ as an example to explain how a gray-box fuzzer typically functions.

A high-level overview of a typical gray-box fuzzing session, demonstrating the basic elements and phases involved, is shown in Figure 1. In the case of the GCC compiler, a GCC plugin, enabled via the *afl-gcc-fast* compiler driver directive, generates efficient code instrumentation that enables AFL++ to monitor the execution of GCC-compiled code. Once the system under test is compiled

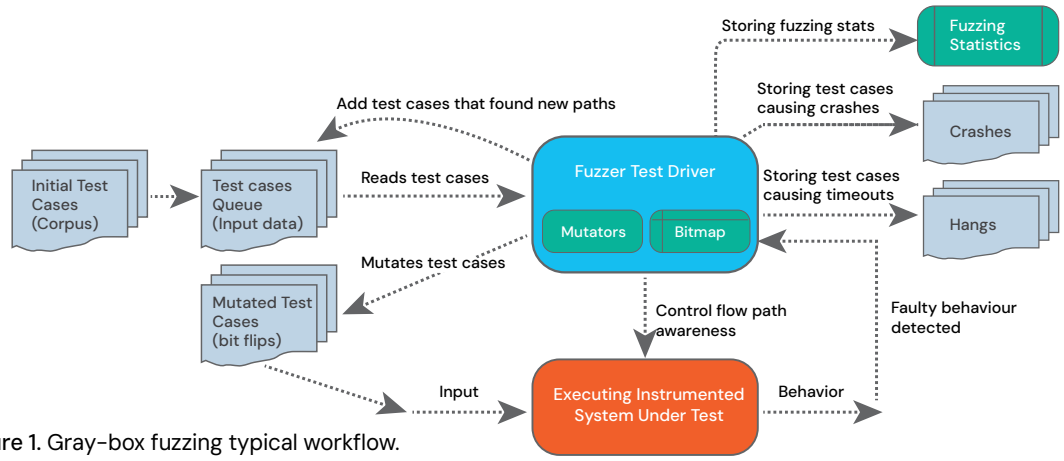

Figure 1. Gray-box fuzzing typical workflow.

with the AFL++ GCC plugin, the *afl-fuzz* fuzzing driver can be used to launch a fuzzing session. Note that AFL++ is also supported for LLVM-based compilers. Before starting a fuzzing session, the user must provide initial test cases, called the seed corpus, each in a separate file. Starting a fuzzing session moves the seed corpus to the test cases *queue*, a directory containing a list of files, each representing a test case. AFL++ adds tests to the *queue* but never removes any.

The details are complex, but in a nutshell, for each test case in the *queue*, AFL++ uses one or more internal or external mutators to create multiple new test cases (it can generate hundreds to tens of thousands of new tests from each test in the *queue*, each of which it executes on the system under test). The fuzzer monitors each execution for any faulty behavior and moves any test cases that trigger such behavior to a directory called *crashes*. AFL++ provides a configuration parameter that sets a timeout for each test. If the test exceeds the timeout, its execution is killed, and the relevant test case is stored in the session's *hangs* directory. Saving malfunctioning tests allows them to be replayed and investigated.

While running each test, AFL++ collects all instrumentation data and updates its *bitmap*, a global session array whose in-dices correspond to instrumentation points added to the system under test during its compilation by the GCC AFL++ plugin. Whenever an instrumentation point is executed, its corresponding index in the *bitmap* is incremented. In AFL-style coverage, *bitmap* indices are derived from instrumentation identifiers, often combined with lightweight context such as the previously executed location to approximate edge coverage. Therefore, the *bitmap* enables the fuzzing session to be path-coverage aware. If a mutated test case executes a new path, it is considered a good candidate for increasing code coverage and is added to the *queue* so that AFL++ will generate further mutations of that test. For multi-threaded programs, AFL++ provides a thread-safe instrumentation mode that ensures atomic updates to the shared coverage *bitmap*, improving coverage tracking stability without altering the underlying coverage model. The fuzzing session cycle counter is incremented when the fuzzer has finished processing all test cases in the *queue* and loops back to the beginning. The user can manually interrupt a fuzzing session when they consider the testing to be satisfactory.

2.4. The Adaptive Engine

To research the benefits, limitations, and overlaps of fuzz testing in memory-safe verification environments, such as CHERI-based hardware architectures and software-based sanitizer strategies, we developed the 'Adaptive Engine' fuzzer. This fuzzing engine was designed to be lightweight, generic, and highly portable for use in resource-constrained embedded systems. This design goal was critical because, while established tools like AFL++ and libfuzzer are popular, at the time of this investigation, they either lacked stable CHERI ports or were not designed for the bare-metal environment we are targeting for future research.

Similar to AFL++, the Adaptive engine is also a gray-box fuzzer and thus operates according to the same principles described in Section 2.3. Furthermore, the Adaptive Engine extends and specializes its capabilities beyond what out-of-the-box fuzzers, such as AFL++, offer. The goal of this specialization is two-fold. First, to automate many of the manual steps needed for setting up, monitoring, and stopping a fuzzing cam-paign, and to add extra capabilities, such as source code coverage collection. Second, to make it a lightweight and highly architecturally portable solution. The following subsections provide a brief overview of the most critical aspects of the Adaptive Engine and how they differ from existing gray-box libfuzzers.

2.4.1. Automation

The Adaptive Engine is integrated within GNATfuzz. GNATfuzz is a fuzz-testing library that automates and simplifies the manual, non-intuitive steps of using fuzzers effectively. In a

nutshell, GNATfuzz abstracts away the complexity of building, executing, and terminating fuzzing campaigns and enables unit-level fuzz testing. In more detail, the library offers the following automation and enhancements:

- Automatic detection of a project's fuzzable subprograms.
- Automatic creation of the test harnesses.
- Offers a good set of predefined fuzz testing configurations and mechanisms to provide a custom fuzzing session con-figuration easily.
- Automatic starting-corpus creation and minimization.
- Allowing for the specification of criteria for stopping a fuzzing session and monitoring how those criteria are being met during the session.
- Provides live source-code-level coverage feedback.
- Automatic support of complementary to fuzzing engines bug-finding capabilities, such as symbolic execution via SymCC, and state-to-input correspondence via CMPLOG. These capabilities are needed to generate test cases that access tight branches, something fuzzers are typically weak at.

The library currently supports all the above features for programs written in Ada and SPARK, and for C and C++ programs via Ada bindings. Work is underway to soon target C and C++ programs without the need for Ada bindings.

Furthermore, the library is designed to accommodate multiple fuzz-testing engines. Currently, AFL++, libFuzzer, and, recently, the Adaptive engine have been integrated into GNATfuzz. When the fuzzers run on the same platform, GNATfuzz enables them to collaborate by executing them in parallel and exchanging the test cases they generate, thereby increasing source-code coverage. Note that GNATfuzz ensures that test cases generated by one tool are compatible with all the GNATfuzz-supported tools. This test case communication could allow fuzzers to discover new test cases that further increase code coverage. Similarly, fuzzers can leverage test cases generated by tools such as SymCC and CMPLOG.

As the Adaptive Engine is integrated into GNATfuzz, it can leverage the majority of GNATfuzz's automation and advanced capabilities, with some customizations that will be explained in the following sections. However, for this work, other engines, such as libfuzzer, and tools, such as SymCC and CMPLOG, are not currently supported on CheriBSD. While we cannot directly benefit from these tools on that platform, we can still use their generated test cases for the same subprograms under test from fuzzing sessions executed on other supported platforms. For example, the same subprograms we target for our experiments on CheriBSD can be tested on an X86 platform with AFL++, libfuzzer, SymCC, and CMPLOG, and their generated test cases can be passed as a starting corpus to the fuzzing session on CheriBSD with the Adaptive Engine. Thus, by utilising a starting corpus that has already successfully explored the program's under test Control Flow Graph (CFG), the Adaptive Engine can be more efficient at discovering subtle code issues, such as buffer overflows.

2.4.2. Portability

The Adaptive Engine is an autonomous fuzzer implemented in Ada and has no dependencies other than the Ada standard library. It leverages the widespread availability of the Ada runtime across numerous platforms, thus enabling its compilation on all systems that support the full Ada runtime. The user's code and the fuzzing engine implementation are integrated into a single executable, resulting in a monolithic binary that can be deployed onto the intended target. Ongoing work also aims to support the Adaptive Engine with the Ada embedded runtime, enabling fuzzing on bare-board platforms.

In contrast, existing fuzz-testing engines, such as AFL++, are typically not designed to support many platforms due to their reliance on OS features, such as the Unix fork system call.

As explained in Section 2.3, typically, gray-box fuzzers need to use some form of code coverage to guide the fuzzer in exploring the CFG of the program under test. Such code coverage depends on code instrumentation compiled into the fuzz-tested executable and on the instrumentation being collected and evaluated at runtime. Most available gray-box fuzzers rely on their own code coverage implementation, which typically targets Path coverage. For example, in the case of AFL++, a custom bitmap implementation is used to track the Path coverage and guide the fuzzer, as described in Section 2.3. While a custom coverage implementation can yield a significantly optimized solution by minimizing instrumentation overhead and thus its impact on the execution time of the fuzz-tested executable, it limits the types of code coverage that can be used.

The Adaptive Engine takes a different approach to coverage-guided fuzzing than existing gray-box fuzzers. Instead of using a custom instrumentation solution, it uses an existing solution provided by the GNATcoverage tool [8]. GNATcoverage is an industrial-grade coverage tool for Ada and C/C++, supporting statement, decision, and Modified Condition/Decision Coverage (MC/DC) across embedded, real-time, and general-purpose environments. Thus, Adaptive Engine can leverage all available GNATcoverage supported coverage types to guide its fuzzing.

The workflow of how the Adaptive Engine utilizes GNATcoverage is as follows:

- The program under test is compiled with GNATcoverage instrumentation.
- During the fuzzing campaign, the gnatcoverage runtime is invoked to acquire code coverage information. This runtime furnishes the engine with the count of satisfied coverage obligations at a given point in time. Subsequently, the engine compares the number of coverage obligations before and after the execution of the user function. Should the user function not crash and the number of fulfilled coverage obligations increase, the mutation is deemed "of interest" because it has triggered the execution of an uncovered portion of the code.
- Any test case that is deemed to increase the code coverage in the previous step is being prioritized by the Adaptive Engine for further mutation and testing.

In this paper, we focus on executed statements and explored branches for coverage, as they are easier to achieve than MC/DC but still capable of exploring our benchmarks' CFGs and discovering their memory vulnerabilities.

2.5. Anomaly Detection

In standard fuzzing solutions such as AFL++ [17] and LibFuzzer [21], there is a heavy reliance on the OS to detect un-expected termination signals, such as SIGSEGV or SIGILL. A SIGSEGV is an indication of a memory access error; an attempt was made to execute an unsafe memory instruction. A classic example of how an application could trigger such a signal is via a buffer overflow. However, it is not always the case that a buffer overflow will result in a SIGSEGV. Instead, it depends entirely on the specific address associated with the unsafe memory instruction and where the overflow is actually trying to write data. If the write operation attempts to corrupt memory outside of the process's allowed address space, the OS will react to a hardware-generated fault notification by signaling a SIGSEGV. Specifically, the OS will use a CPU's Memory Management Unit (MMU) to protect memory pages. When a process attempts to write to an unmapped page or a page explicitly marked as protected, the hardware triggers a page fault, which the OS kernel catches and handles by sending the process a SIGSEGV signal. However, suppose the buffer overflow results in a memory write instruction within the process's allowed address space, for example, overflowing into a neighboring variable within the same process. In that case, the OS will not

trigger a SIGSEGV. Instead, the fuzzer must rely on other forms of anomaly detection, such as ad-dress sanitizers like AddressSanitizer (ASan) [20] and UndefinedBehaviorSanitizer (UBSAN) [22]. Other forms include programming-language runtime constraint checking, as found in the GNAT Pro Ada runtime [7], and CHERI pure capability mode, which leverages hardware-detected memory misuse that the OS can also detect.

The highly portable design of the Adaptive Engine advances this paradigm by enabling fuzzing sessions to occur on any target that supports the full GNAT Pro Ada runtime. This includes AdaCore's GNAT Pro for the CHERI compiler toolchain aimed at the Arm Morello platform and targeting the CheriBSD OS [4]. As discussed in the "Security by Default - CHERI ISA Extensions coupled with a security-enhanced Ada runtime" paper [5], the GNAT Pro for CHERI Ada runtime includes CHERI pure-capability memory allocators and the ability to catch and trap CHERI hardware capability faults within enclosed Ada exception handlers. Within the Adaptive engine, a top-level exception handler can detect and respond to CHERI capability faults, storing the scenario (i.e., the test case values) that triggered the anomaly. When coupled with the standard GNAT Pro Ada runtime constraint checks, we have found that the result is an anomaly detection mechanism that is superior to standard address sanitizers, and more fine-grained than relying on a CPU's MMU, which cannot detect unsafe memory instructions within the same process address space (see Section 3.1).

3. Experimental Process and Results

We identified several prevalent and often elusive bugs in C. This section details each of these vulnerabilities. Some of our test cases incorporate conditional statements to emulate the behavior of real-world applications, thereby increasing the complexity of the fuzzing campaign.

Each function present in our benchmarks reproduces common memory bugs. Each erroneous function is then tested on the following platforms and configurations:

- Adaptive engine, x86 without ASan
- Adaptive engine, x86 with ASan
- Adaptive engine, Morello

To ensure reproducibility, a fixed random seed of 1 was employed for all runs. The fuzzer was executed for a maximum duration of five minutes per function and per platform, and testing ceased immediately upon the first detected crash.

To determine the key differences, overlaps, and limitations in vulnerability detection between CHERI and ASan, we developed an extensive test suite of vulnerabilities shown in Table 1 and [6]. We tested the bug-capturing capabilities of var-ious approaches (including ASan and CHERI) across multiple platforms (see section 3.1). Additionally, to help uncover the limitations

Bug type	Description
Stack buffer over-flow	A stack buffer overflow occurs when a program writes more data than a stack-allocated buffer can hold.
Heap buffer over-flow	A heap buffer overflow happens when writes extend past the bounds of a dynamically allocated buffer.
Double free	Occurs when a program calls free() (or equivalent) more than once on the same memory pointer.
Use after free	Occurs when a program continues to use or reference memory after it has been freed or deallocated.
Array in struct	Occurs when performing an out-of-bounds access on an array declared within a struct.
Out-of-bounds pointer arithmetic	A vulnerability where out-of-bounds pointer arithmetic causes a pointer to jump illegally from one buffer into an adjacent buffer.
ASan overflow evading	A specific heap buffer overflow that exploits AddressSanitizer's architectural limitations by overflowing a buffer such that ASan poisoned red zone memory is avoided.

Table 1. Classes of bugs we consider.

of the methods, we developed a specialized automated tool to systematically explore memory corruption scenarios and identify cases where ASan fails to detect buffer overflows [6]. We then tested the same scenarios on a CHERI hardware-enforced capability system.

3.1. Benchmark results and discussion

3.1.1. Stack overflows

A stack buffer overflow occurs when a program attempts to read or write to memory beyond the boundaries of a stack-allocated buffer. This is a common vulnerability that can allow an attacker to overwrite a function's return address and execute arbitrary malicious code. From a purely software perspective, buffer overflows can also corrupt the values of other stack variables, leading to an invalid program state. On a traditional system, an access attempt within the allocated stack area is generally permitted; however, on a CHERI-based architecture, the validity of the access is contingent upon the metadata associated with the pointer used.

We evaluated various types of buffer overflows, including those accessing data significantly beyond the buffer size limits and others exceeding the buffer end by only a few bytes.

Setup			Crash	Remarks
Architecture	Language	ASan		
x86	C	Yes	Yes	Only large overflows that caused a segmentation fault were detected.
x86	C	No	No	
x86	Ada	Yes	Yes	Only large overflows that caused a segmentation fault were detected.
x86	Ada	No	No	
Morello	C	-	Yes	CHERI capability fault
Morello	Ada	-	Yes	CHERI capability fault

Table 2. Stack overflow results

Experimental results indicate that ASan and Morello reliably detected these issues, while plain x86 systems only exhibited crashes with larger stack buffer overflows, attributable to attempts to access unallocated memory addresses.

3.1.2. Heap buffer overflows

All tests focus on detecting both small and large heap buffer overflow accesses.

Setup			Crash	Remarks
Architecture	Language	ASan		
x86	C	Yes	Yes	
x86	C	No	No	
x86	Ada	Yes	Yes	Ada runtime exception
x86	Ada	No	Yes	Ada runtime exception
Morello	C	-	Yes	CHERI capability fault
Morello	Ada	-	Yes	CHERI capability fault

Table 3. Heap buffer overflow

Similarly, both ASan and Morello detected all out-of-bounds accesses arising from these heap buffer overflows. On x86 systems, small overflows often go unnoticed because adjacent memory is typically valid; in contrast, ASan and Morello detected these violations through bounds checking.

3.1.3. Double free

Initially, both ASan and Morello failed to detect double-free calls. Nevertheless, we observed that the Linux allocator occasionally detected the double free, leading to program termination. Conversely, the CheriBSD allocator silently ignored invalid free function calls. After further investigation, we realised that the *runtime_revocation_every_free_default* CheriBSD feature, which automatically revokes all pointer capabilities when they are passed to free function calls, was disabled. Enabling it makes CheriBSD detect double-free calls.

Setup			Crash	Remarks
Architecture	Language	ASan		
x86	C	Yes	Yes	
x86	C	No	No	Allocator Assertion
x86	Ada	Yes	Yes	
x86	Ada	No	Yes	Allocator Assertion
Morello	C	-	Yes	With OS feature
Morello	Ada	-	Yes	With OS feature

Table 4. Double free

3.1.4. Use after free

Setup			Crash	Remarks
Architecture	Language	ASan		
x86	C	Yes	Yes	
x86	C	No	No	
x86	Ada	Yes	Yes	Heap corruption
x86	Ada	No	Yes	Heap corruption
Morello	C	-	Yes	With OS feature
Morello	Ada	-	Yes	With OS feature

Table 5. Double free

All instances of “use after free” were successfully detected by ASan. A crash was observed during the execution of the Ada tests; however, this outcome was unexpected as the fuzzing engine itself was failing due to heap corruption.

CheriBSD distinguishes between a *trap* (an immediate error) and *safety* (prevention of exploitation). It ensures that a use-after-free is never dangerous as accessing a reallocated block via a dangling pointer by managing memory through three distinct phases:

- **Quarantine:** Upon *free*(), memory is moved to a quarantine pool. While dangling capabilities remain valid and can access the block, the operation is *benign* because the memory has not yet been repurposed for a new object.
- **Revocation:** A background *revoker* scans memory for capabilities pointing to quarantined blocks and clears their tag bit. This renders the pointers permanently non-dereferenceable.
- **Reallocation:** Once revocation is complete and no valid capabilities remain, the memory is safely released to the allocator. Any subsequent access attempt will trigger a *deterministic hardware trap*.

While this behavior provides robust memory safety, it complicates the use of CHERI-based architectures for fuzzing-driven bug detection. Because the double-free occurs before the capabilities have been revoked, the hardware does not initially trigger a trap. However, by “forcing” the revocation process either on-demand via *malloc_revoke()* or immediately following a *free()* as described in 3.1.3 these errors are successfully detected on the Morello platform.

Setup			Crash	Remarks
Architecture	Language	ASan		
x86	C	Yes	No	
x86	C	No	No	
x86	Ada	Yes	Yes	Ada runtime exception
x86	Ada	No	Yes	Ada runtime exception
Morello	C	-	No	
Morello	Ada	-	Yes	Ada runtime exception

Table 6. Array in struct

3.1.5. Array in struct

Overflowing an array declared within a structure is not detected by ASan unless access extends beyond the structure's boundaries. Conversely, the Ada runtime successfully detects such out-of-bounds accesses in these arrays due to its built-in out-of-bounds detection mechanism. It is also worth nothing that there's some ongoing work in the morello-llvm project to improve the compiler generated capabilities in this situation and making this error visible to the Morello hardware [23].

Unfortunately, we were not able to use this feature yet as it resulted in a compiler bug for us. We expect that Morello will be able to detect this error in the future when the relevant compiler feature is more mature.

3.1.6. Out of bounds pointer arithmetic

Setup			Crash	Remarks
Architecture	Language	ASan		
x86	C	Yes	No	
x86	C	No	No	
Morello	C	-	Yes	CHERI capability fault
Morello	C	-	Yes	CHERI capability fault

Table 7. Out-of-bounds pointer arithmetic

This test involves a stack buffer overflow that overwrites data in an adjacent buffer on the stack. ASan is unable to detect this specific vulnerability as it constitutes a legal memory access by design. However, CHERI-based platforms can detect this issue because the bounds are explicitly encoded within the pointer capabilities, making the access illegal from the hardware point of view.

3.1.7. ASan overflow evading

This evaluation mirrors the previous test but uses heap-allocated buffers. Although the allocator does not guarantee adjacent placement, the small allocation size resulted in the buffers being placed next to each other in most cases.

As seen in section 3.1.6, these errors bypass ASan detection because they represent semantically invalid accesses that are nonetheless permitted by the system's memory mapping.

4. Discussion

In Section 3, we evaluated two prominent solutions for capturing software memory safety issues: a software-based heuristic approach based on ASan, and a hardware-based architectural modification approach based on CHERI. Software-based approaches, exemplified by ASan, add compiler instrumentation and specialized runtime libraries to introduce runtime checking of unsafe memory accesses. In contrast, architectural approaches, such as the CHERI examined in

this paper, embed memory-safety enforcement directly into the CPU and memory system. This section provides a more detailed look at our experimental results to evaluate the strengths and weaknesses of each approach in terms of their memory issue-capturing capabilities. However, as performance is also an essential factor in fuzz testing, an understanding of the overheads that each technique introduces is needed.

Setup			Crash	Remarks
Architecture	Language	ASan		
x86	C	Yes	No	
x86	C	No	No	
Morello	C	N/A	Yes	CHERI capability fault
Morello	C	N/A	Yes	CHERI capability fault

Table 8. ASan overflow evading

While ASan offers enhanced capabilities in detecting many common bugs, including use-after-free, double-free, and most forms of buffer overflows, this software-level enforcement introduces significant overhead, typically resulting in a performance slowdown ranging from 2 to 3 times (on average) and substantial memory consumption due to the requirement for shadow memory and *red zones* [18], when compared to the same program executed without ASan instrumentation. In comparison, pure-capability CHERI systems, both in their hardware architecture and software runtime implementations, typically incur a minimal performance penalty [38]. Bench-marking the performance of the two approaches against each other falls outside the scope of this study, which primarily focused on anomaly detection capability.

Our findings revealed a fundamental architectural limitation in ASan's *red zone* approach that allows specific overflow patterns to evade detection. In all cases, CHERI detected these specific overflow patterns and flagged hardware capability faults. We extracted this finding as a special test case called "ASan overflow evading" 3.1.7.

ASan instrumentation includes *red zones* (poisoned memory regions) placed around memory allocations. For small heap allocations, the red zones are typically 16-32 bytes in size. When the instrumented code accesses memory, ASan checks whether the address falls within a *red zone*; any attempt to access memory in a red zone results in ASan reporting a buffer overflow. As *red zones* have a finite size, any overflow that exceeds the red zone boundary and lands in the subsequent valid allocation is undetectable because the overflowing write never touches poisoned (*red zone*) memory. Furthermore, if the target location is addressable (i.e., a valid memory allocation within the process's virtual address space), and because ASan has no mechanism to track which pointer accesses which allocation, it will see the memory access as valid and allow it. This is not a bug in ASan, but rather a fundamental trade-off in its design. Our automated tool also detected similar problematic cases for stack and data segment (global variable) allocations.

In all our test scenarios, the Adaptive Engine on Morello flagged the exploits and detected the vulnerability, while ASan failed. CHERI's hardware-enforced bounds check on every capability access eliminates this vulnerability class.

5. Conclusion

This paper investigated the effectiveness of fuzz testing in a memory-safe verification environment by combining hardware-enforced memory safety, via the CHERI architecture, with coverage-guided fuzzing. We experimentally evaluate this approach across both memory-safe and memory-unsafe programming languages and compare it against the state-of-the-art practice of fuzzing instrumented with ASan.

Our results demonstrate that while ASan remains effective at detecting many common classes of memory errors, it exhibits inherent limitations stemming from its red-zone–based design. In particular, we identified a class of spatial memory violations, referred to as *Red Zone Jump* patterns, that systematically evade detection despite being semantically invalid. In contrast, CHERI's capability-enforced bounds checking reliably detected these violations by converting otherwise silent memory corruption into deterministic hardware faults.

Beyond legacy memory-unsafe code, our findings also show that hardware-assisted fuzzing strengthens the verification of memory-safe languages by detecting misuse of unsafe language features that bypass static guarantees. This highlights an important distinction between language-level memory safety and system-level verification completeness: even memory-safe languages benefit from hardware-enforced runtime checking when subjected to adversarial testing techniques such as fuzzing.

The Adaptive Engine played a key role in enabling these experiments by providing a lightweight and portable coverage-guided fuzzing framework capable of operating across architecturally distinct execution environments, including conventional x86 systems and CHERI-based pure-capability platforms. Its integration with existing coverage and runtime-checking mechanisms enabled consistent fuzzing across architectures while maintaining fine-grained anomaly detection, and its integration within GNATfuzz automated much of the effort required to configure, execute, and monitor fuzzing campaigns. These characteristics make the Adaptive Engine well-suited for early-stage verification workflows, particularly in embedded and real-time contexts where traditional fuzzing tools are difficult to deploy. In this setting, the combination of a portable fuzzing framework and memory-safe hardware provides practical support for Secure by Design and Secure by Verification approaches.

Although CHERI hardware remains in the early stages of adoption, the results presented here highlight its potential as a powerful complement to existing dynamic analysis techniques. As memory-safe architectures mature and become more widely available, hardware-assisted fuzzing offers a promising path to systematically eliminate entire classes of memory-safety vulnerabilities and reduce the long-term attack surface of critical embedded and real-time systems.

References

[1] Ada Reference Manual. Unchecked Type Conversions. http://www.ada-auth.org standards/12 rm/html/RM-13-9.html, 2012. Section 13.9, Accessed: 2025-12-15.

[2] AdaCore. When formal verification with SPARK is the strongest link. https://www.adacore.com/blog/when-formal-verification-with-spark-is-the-strongest-link, 2021. AdaCore blog post, Accessed: 2025-12-15.

[3] AdaCore. NVIDIA security team: What if we just stopped using C? https://www.adacore.com/blog/nvidia-security-team-what-if-we-just-stopped-using-c, 2021. AdaCore blog post, Accessed: 2025-12-15.

[4] AdaCore. Elevate Security Confidence with Memory Safe Hardware and Software. https://www.ad acore.com/papers/elevate-security-confidence-with-memory-safe-hardware-and-software, 2024. AdaCore technical paper, Accessed: 2025-12-15.

[5] AdaCore. Security by Default – CHERI ISA Exten-sions Coupled with a Security-Enhanced Ada Runtime. https://www.adacore.com/papers/sec urity-by-default-cheri-isa-extensions-coupled-with-a-security-enhanced-ada-runtime, 2024. Technical paper, Accessed: 2025-12-15.

[6] AdaCore. ERTS 2026: Memory Safety Research for Embedded Real-Time Systems - Research Artifacts. https://github.com/AdaCore/ERTS_2026, 2025. GitHub repository, research artifacts related to the 2026 Embedded Real-Time Systems Conference, Accessed: 2025-12-15.

[7] AdaCore. GNAT Pro Run-Times. https://docs.a dacore.com/gnat_ugx-docs/html/gnat_ugx/gnat_ugx/gnat_runtimes.html, 2025. GNAT User's Guide documentation, Accessed: 2025-12-15.

[8] AdaCore. GNATcoverage User's Guide. https: //docs.adacore.com/gnatcoverage-docs/html/gnatcov/gnatcov_part.html, 2025. GNATcoverage documentation, Accessed: 2025-12-15.

[9] AWS Open Source Blog. Why aws loves rust and how we'd like to help. https://aws.amazon.com/blogs/opensource/why-aws-loves-rust-and-how-wed-like-to-help/, 2020. Accessed: 2025-12-15.

[10] AWS Open Source Blog. Sustainability with rust. https://aws.amazon.com/blogs/opensource/sustainability-with-rust/, 2022. Accessed: 2025-12-15.

[11] Cybersecurity and Infrastructure Security Agency. Case for Memory Safe Roadmaps. https://www. cisa.gov/case-memory-safe-roadmaps, 2024. Accessed: 2025-12-15.

[12] Cybersecurity and Infrastructure Security Agency. Secure by Design. https://www.cisa.gov/securebydesign, 2024. Accessed: 2025-12-15.

[13] Defense Advanced Research Projects Agency. Clean-Slate Design of Resilient, Adaptive, Secure Hosts (CRASH). https://www.darpa.mil/research/programs/clean-slate-design-of-resilient-adaptive-secure-hosts, 2012. DARPA research program, Accessed: 2025-12-15.

[14] Department for Science, Innovation and Technology and Feryal Clark MP. CHERI technology for cyber security. https://www.gov.uk/government/publications/cheri-technology-for-cyber-security, 2025. Policy paper, Published 7 May 2025, Accessed: 2025-12-15.

[15] EUROCAE. ED-202B: Airworthiness Security Process Specification. https://www.eurocae.net/product/ed-202b-airworthiness-security-process-specification/, 2023. EURO-CAE standard, Revision B.

[16] EUROCAE. Ed-203a: Airworthiness security methods and considerations. https://www.eurocae. net/product/ed-203a-airworthiness-security-methods-and-considerations/, 2023. Accessed: 2025-12-15.

[17] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[18] Google. AddressSanitizer performance numbers. https://github.com/google/sanitizers/wiki/AddressSanitizerPerformanceNumbers, 2015. GitHub wiki page, Accessed: 2025-12-15.

[19] Google Security Blog. Eliminating memory safety vulnerabilities in android. https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html, 2024. Accessed: 2025-12-15.

[20] LLVM Project. Addresssanitizer. https://clang. llvm.org/docs/AddressSanitizer.html, 2024. Accessed: 2025-12-15.

[21] LLVM Project. LibFuzzer — a Library for Coverage-Guided Fuzz Testing. https://llvm.org/docs/LibFuzzer.html, 2024. Accessed: 2025-12-15.

[22] LLVM Project. UndefinedBehaviorSanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2025. Clang/LLVM documentation, Accessed: 2025-12-15.

[23] Paul Metzger, Jonathan Woodruff, Robert N. M. Wat-son, Brooks Davis, Wes Filardo, Jessica Clarke, and John Baldwin. Explore subobject bounds - SOSP 2023 CHERI tutorial exercises. Online, October 2023. URL https://www.cl.cam.ac.uk/~pffm2/sos p2023_cheri_tutorial/exercises/4_sub object-bounds/README.html. Presented at the 29th ACM Symposium on Operating Systems Principles (SOSP 2023), October 23, 2023.

[24] Microsoft. Trends and challenges in the vulnerability mitigation landscape, 2019. URL https://github.com/microsoft/MSRC-Security-Researc h/blob/master/presentations/2019_02_ BlueHatIL/2019_01%20-%20BlueHatIL%20 -%20Trends%2C%20challenge%2C%20and%2 0shifts%20in%20software%20vulnerability%20mitigation.pdf.

[25] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. Com-mun. ACM, 33(12):32–44, December 1990. ISSN 0001-0782. doi: 10.1145/96267.96279. URL https://doi.org/10.1145/96267.96279.

[26] MITRE Corporation. 2025 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2025/2025_cwe_top25. html, 2025. Accessed: 2025-12-15.

[27] NVIDIA. NVIDIA ISO-26262 SPARK Process. https://nvidia.github.io/spark-process/process/, 2025. NVIDIA SPARK Process documentation, Accessed: 2025-12-15.

[28] RTCA. DO-326B: Airworthiness Security Process Specification. https://store.accuristech.com/standards/rtca-do-326b?product_id=2939833, 2024. RTCA standard, Revision B (Airworthiness Security Process).

[29] RTCA. Aviation cybersecurity - rtca do-356a. https: //www.rtca.org/security/, 2024. Accessed: 2025-12-15.

[30] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975. doi: 10.1109/PROC.1 975.9939.

[31] Saman Rizvi and Eleanor Fordham. Cyber security breaches survey 2025, April 2025. URL https://www.gov.uk/government/statistics/cyber-security-breaches-survey-2025/cyber-security-breaches-survey-2025.

[32] SRI International. SRI International: Independent Non-profit Research Institute. https://www.sri.com/. Accessed: 2025-12-15.

[33] UK Government. Iscf digital security by design: Software ecosystem development. https://apply-for-innovation-funding.service.gov.uk/competition/1020/overview, 2021. Innovation funding competition overview, Accessed: 2025-12-15.

[34] UK Government. Secure by design: Principles. https://www.security.gov.uk/policy-and-guidance/secure-by-design/principles/, 2024. Accessed: 2025-12-15.

[35] University of Cambridge Computer Laboratory. Cheri: Capability hardware enhanced risc instructions. https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/, 2024. Accessed: 2025-12-15.

[36] U.S. Office of the National Cyber Director (ONCD). Back to the building blocks: a path toward secure and measureale software. https://bidenwhitehouse.archives.gov/wp-content/uploads/2 024/02/Final-ONCD-Technical-Report. pdf, feb 2024. Accessed: 2025-12-15.

[37] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Nathaniel Wesley Filardo, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability hardware enhanced risc instructions: Cheri instruction-set architecture (version 7). Technical Report UCAM-CL-TR-927, University of Cambridge, Computer Laboratory, June 2019. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.pdf.

[38] Robert N. M. Watson, Jessica Clarke, Peter Sewell, Jonathan Woodruff, Simon W. Moore, Graeme Barnes, Richard Grisenthwaite, Kathryn Stacer, Silviu Baranga, and Alexander Richardson. Early performance results from the prototype Morello microarchitecture. Technical Report UCAM-CL-TR-986, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500, September 2023.

[39] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37, 2015. doi: 10.1109/SP.2015.9.

[40] America's Cyber Defense Agency. The Case for Memory Safe Roadmaps. https://www.cisa.gov/case-memory-safe-roadmaps, Accessed: 2026-03-13.