# MSTest: A Property-Oriented, Comprehensive, and Cross-Platform Test Suite of Memory Safety

Ciyan Ouyang, Da Xie, Hao Ma, Wei Song, *Senior Member, IEEE*, Jiameng Ying, Sihao Shen, and Peng Liu, *Member, IEEE*

***Abstract*—The foundation of current software ecosystem is still unfortunately laid on memory unsafe languages, such as C/C++. Memory safety vulnerabilities remain as the primary source of bugs in the critical software stacks. Some of the advanced memory safety defenses are beginning to land on commercially available platforms, in the form of instruction-set architecture extensions, runtime enforcement by standard libraries and OSes, and compile-time checks. This tide of adoption of defenses brings us several questions: For a defense that is claimed supported on a platform, can it be actually deployed to directly benefit an application? For a defense claiming a certain level of protection regarding a type of memory safety on a platform, how solid is the protection? For two platforms implementing similar types of defenses, which one provides better guarantees?**

**Endeavor to answer these questions, a memory safety test suite, namely *MSTest*, is implemented. With its current 227 test cases, the test suite has already reached a wider coverage than all existing test suites and been ported to 19 platforms. To our best knowledge, MSTest is the first portable memory safety test suite conducting property-oriented testing, automatically resolving dependency between test cases, providing a comprehensive coverage on attack and defense capabilities, and capable of comparing memory safety cross platforms.**

***Index Terms*—memory safety, control-flow integrity, memory errors, attack primitives, cross-platform, test suite**

## I. Introduction

The foundation of current software ecosystem is still unfortunately laid on memory unsafe languages, such as C and C++. Due to the lack of intrinsic safety guarantees provided by these languages and the abundance of remaining coding errors, memory safety vulnerabilities are the primary source of bugs in the critical software stacks. Memory safety is a set of properties ensuring the execution of a program following its design. An execution violates memory safety if it accesses an object beyond its boundary, or accesses an object that has not been allocated or has already been deallocated. The former is known as spatial safety, while the latter is known as temporal safety [1]. By exploiting memory safety vulnerabilities, attackers could obtain access to arbitrary memory locations, manipulate the control-flow of a program, and eventually hijack a system.

The fight between attacks and defenses is constantly evolving as an eternal war in memory [2]. Some of the early defenses, such as stack canary, write $\oplus$ execute (W$\oplus$E) and address space layout randomization (ASLR) [3], have been widely adopted by all architectures, compilers and operating systems (OSes). They motivate attackers to develop sophisticated and evasive attack techniques, such as use-after-free (UAF), return-oriented programming (ROP) [4], jump-oriented programming (JOP) [5], counterfeit object-oriented programming (COOP) [6] and data-oriented programming (DOP) [7].

To battle with these new attacks, numerous advanced defense techniques have been proposed, such as address sanitizing (ASan) [8], bound checking [9], compartmentalization [10], memory tagging [11], control-flow integrity (CFI) [12], pointer authentication (PA) [13], code-pointer integrity (CPI) [14], data-flow integrity (DFI) [15], and dynamic information flow tracking (DIFT) [16]. Some of these advanced defenses are beginning to land on commercially available platforms, in the form of instruction-set architecture (ISA) extensions, runtime enforcement by standard libraries and OSes, and/or compile-time checks. For example, CFI has been implemented in LLVM and GCC [17] for most architectures. Intel memory protection extension (MPX) [18] was initially added to major compilers for runtime bound checking but later dropped due to its heavy performance overhead. Intel control-flow enforcement technology (CET) [19], a coarse-grained CFI implementation to thwart ROP and JOP, has been incorporated into the Intel Tiger Lake architecture. Arm has introduced its PA [20] in Armv8.3-A, and memory tagging extension (MTE) in Armv8.5-A [21]. Apple has begun its support for Arm PA from the A12 chip and Apple LLVM v8 [22]. In addition, Arm has also engaged in the development of Morello [23], an Arm's implementation of the capability hardware enhanced RISC instructions (CHERI) architecture [10] designed by the University of Cambridge. However, this tide of adoption of defenses brings us several questions urgently needing answers:

- *Q1: For a certain defense that is claimed supported on a platform, can it be actually deployed to directly benefit an application?* Counter-intuitively, there might not be an easy answer. Initially proposed in 2016, Intel CET was claimed supported in GCC version 8, the Intel Tiger Lake architecture (11th gen), and Linux kernel v5.18. However, nine years after its initial proposal, you may still find it

This work was supported in part by the National Natural Science Foundation of China under grant No. 61802402 and 62172406, and in part by the CAS Pioneer Hundred Talents Program. (*Corresponding author: Wei Song*)

Ciyan Ouyang, Da Xie, Hao Ma and Wei Song are with the State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS, Beijing 100085, China, and also with the School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China (e-mail: {ouyangciyan, xieda, mahao, songwei}@iie.ac.cn).

Jiameng Ying is with the Big Data Center of the Ministry of Public Security, Beijing 100176, China (e-mail: yingjiameng@foxmail.com).

Sihao Shen was with the Institute of Information Engineering, CAS, Beijing 100085, China (e-mail: shengsihao19@mails.ucas.edu.cn).

Peng Liu is with the Pennsylvania State University, Pennsylvania, PA 16801, USA (e-mail: pxl20@psu.edu)

TABLE I
COMPARISON WITH EXISTING TEST SUITES COVERING MEMORY SAFETY.

| | | Attack Capabilities | | | | | | | | Defense Capabilities | | | | | Variety of Enforcers | | | | **Portability** | | | **Test Capabilities** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | number of test cases | buffer overflow/underflow | arbitrary memory read/write | **UAF on heap and stack** | ROP | indirect call hijacking | JOP | **COOP** | DOP | backward CFI | forward CFI | compartmentalization | **CPI and PA** | data flow integrity | runtime enforcement | OS enforced defenses | sanitizer / model checker | security-related ISA extensions | **cross-compiler support** | **cross-architecture support** | **cross-OS support** | full-attack testing | **property-oriented testing** | **defense classification** | **dependency auto-resolve** |
| BASS [24] | 7 | ◐ | ◐ | ○ | ◐ | ◐ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ○ | ◐ | ○ | ◐ | ◐ | ○ | ✓ | × | × | × |
| ConFIRM [25] | 24 | ◐ | ◐ | ○ | ◐ | ◐ | ◐ | ◐ | ○ | ◐ | ◐ | ○ | ○ | ○ | ● | ◐ | ◐ | ◐ | ● | ○ | ● | ✓ | × | ✓ | × |
| CBench [26] | 23 | ◐ | ◐ | ◐ | ◐ | ◐ | ◐ | ◐ | ◐ | ◐ | ◐ | ◐ | ○ | ○ | ● | ○ | ◐ | ◐ | ○ | ○ | ○ | ✓ | × | ✓ | × |
| RIPE [27] | 850 | ● | ◐ | ○ | ● | ◐ | ◐ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ◐ | ● | ● | ◐ | ○ | ○ | ✓ | × | × | × |
| RecIPE [28] | 204 | ● | ◐ | ○ | ● | ◐ | ◐ | ○ | ○ | ◐ | ○ | ◐ | ○ | ◐ | ● | ● | ● | ◐ | ○ | ○ | ○ | ✓ | × | ✓ | × |
| *MSTest* | 227 | ● | ● | ◐ | ● | ● | ● | ● | ◐ | ● | ● | ◐ | ● | ◐ | ● | ● | ● | ● | ● | ● | ● | × | ✓ | ✓ | ✓ |

Attack and defense capabilities: ● sufficiently covered, ◐ partially covered, ○ barely covered.
Variety of enforcers: ● the desired use-case or known to be widely utilized, ◐ potentially usable, ○ cannot be used.
Portability: ● sufficiently portable, ◐ missing major compiler, architecture or OS, ○ not portable by design.

insufficiently supported by a latest Linux (v6.5) running on an Intel 12th gen processor.

- *Q2: For a defense claiming a certain level of protection regarding a type of memory safety on a platform, how solid is the protection? Is there any remaining vulnerability?* The same type of defenses may provide different levels of protection depending on the chosen compiler flags. Taking Arm Morello for an example, its boundary check can be set to one of five available levels. It is not easy to tell the (exact) differences between levels. When Morello is applied with the full-force, it is also unclear whether there are still loopholes in its protection.
- *Q3: For two platforms implementing similar types of defenses, which one provides better guarantees in memory safety?* This has gradually become a common question, since x86-64 and Arm are taking different ways in defending CFI. Even for the same Intel CET support, Linux and Windows are implemented differently. Platforms using different processors and OSes may have defenses covering the same types of memory safety vulnerabilities, but there is no automatic/systematic way to quantitatively compare them. Existing test suites are very limited in achieving this goal due to their lack of portability and insufficient coverage on both attack and defense capabilities.

Endeavor to answer these questions, we have gradually implemented a memory safety test suite, namely *MSTest*, in the previous six years. To answer *Q1* and *Q2*, MSTest has deliberately incorporated test cases checking the effectiveness of individual defense properties. It is then possible to verify whether a specific property claimed by a defense supported on a platform is actually enabled and effective in preventing the intended type of attacks. If any of these attacks is tested successful, the corresponding vulnerability remains exploitable and is identified. To answer *Q3*, MSTest provides a flexible and cross-platform test framework which is capable of running the same tests on different platforms. Consequently, multiple platforms can be properly compared by running the same set of test cases. With its current 227 test cases, the test suite has already reached a wider coverage than all existing test suites [24]–[28] and been ported to 19 platforms crossing x86-64/Armv8-A/RISC-V, Window/Linux/Darwin, and MSVC/GCC/LLVM. Our test results show that software sanitizers are effective in detecting violations in spatial and temporal safety, along with forward CFI. Apple's PA provides similar forward CFI enforcement, while CET-IBT and Arm-BTI is significantly weaker. CET-SHSTK provides stronger protection against ROP attacks than Arm PA. CHERI is indeed impressive in protecting spatial memory safety. To our best knowledge, MSTest is the first portable memory safety test suite conducting property-oriented testing, automatically resolving dependency between test cases, providing a comprehensive coverage on attack and defense capabilities, and capable of comparing memory safety cross platforms. MSTest is available at:

https://github.com/comparch-security/cpu-sec-bench/

## II. COMPARING WITH EXISTING TEST SUITES

There are several existing test suites [24]–[28] concentrating on the memory safety of a platform, but none of them is sufficient enough to provide a thorough evaluation of the memory safety or compare the safety between different platforms. To highlight the missing capabilities of existing test suites, Table I summarizes the existing test suites and MSTest according to five categories of capabilities described as follows:

*Attack capabilities*: A full attack depends on the successful exploitation of a sequence of memory vulnerabilities. Generic buffer overflow is normally utilized in the early stages of an attack to gain the capability of arbitrary memory read and write, while further specific memory corruption vulnerabilities are necessary for control hijacking. A test suite shall evaluate whether an attacker can exploit individual vulnerabilities on a certain platform.

*Defense capabilities*: Defenses may have different capabilities regarding the same memory vulnerability. For example,

CFI protects an indirect call by checking whether the call destination is valid, but it cannot stop attackers from replacing a function pointer with another one complying with control-flow graph. Code pointer integrity can be enforced by CPI or PA, while they both assume the original pointer is not corrupted. Data flow integrity might be used to verify the validity of a code pointer. Defenses have their bounded capabilities and remaining vulnerabilities, both of which should be evaluated.

Ability to accommodate a *variety of enforcers*: Defenses are enforced by different entities with the help of a capable compiler. Most CFI and UAF defenses are enforced at runtime, by compiler-inserted instrumentation and enhanced runtime libraries. Some of the widely adopted defenses, such as ASLR and W⊕E, are directly enforced by the OS. Modern compilers might complain or refuse to compile when a vulnerability is identified in the source code. Finally, the latest ISA extensions, such as Intel CET, Arm PA and MTE, are gradually fitted in the ecosystem. A test suite should be able to evaluate the defenses enforced by different entities.

*Portability*: Platform testing is very different from software testing. When software is being tested for memory corruption vulnerabilities, each test case is a specific input (e.g., an URL, a PDF document); in contrast, when a platform is being tested, each test case is a specific piece of code whose execution involves safety-violating memory accesses. A unique characteristic of platform testing is that a test case often contains *embedded assembly* as part of it, but this embedded assembly is normally platform-dependent. However, to compare the memory safety provided by different platforms, these platforms should be evaluated using the same test suite, which then requires the test suite to be portable between architectures, OSes and compilers.

*Test capabilities*: Existing test suites consist of only test cases using full attacks, which can be very limited: Even if a full attack fails, some vulnerabilities may have already been successfully exploited. A property-oriented test suite can evaluate a single attack or defense capability by a small self-sufficient program, which is more efficient in reaching wider coverage than full attacks. Such fine granularity also allows a test suite to identify the existence of a defense. A property-oriented test suite may further analyze the dependency between test cases.

BASS [24], ConFIRM [25] and CBench [26] are three representative test suites using full attacks to evaluate memory safety. Published in 2006, BASS is one of the earliest test suites. It contains 7 vulnerable programs and generates exploits targeting these programs on 32-bit x86 and 64-bit Alpha Linux platforms. The suite concentrates only on buffer overflow type of memory vulnerabilities and is ignorant to defenses by design. ConFIRM [25] is a recent test suite using 24 full-attack programs to evaluate the applicability of existing CFI defenses on x86-64 Linux and Window platforms. It extends the coverage to include control hijacking attacks, and begins to construct dedicated tests to differentiate the capabilities between defenses. CBench [26] further improves the coverage by adding attacks utilizing UAF on heap, out-of-bound code pointer reuse (DOP), and cross-module access (compartmentalization). However, in its current form, it requires LLVM and runs on x86-64 Linux only. Due to the small number of test cases, attack and defense capabilities are either partially covered or completely neglected.

RIPE [27] is the most utilized test suite for evaluating defenses against control hijacking attacks. It adopts a test generation framework where 850 attacks are generated from a single full-attack template using five variants. In this way, RIPE can achieve sufficient coverage of all types of buffer overflow and ROP attacks. A later research [28] finds out that RIPE presents high false positive rate. A significant number of test cases fail without a presenting defense as the generated code is malformed. Using a similar framework, RecIPE [28] effectively reduces false positive rate by generating and running dedicated exploitation code for each attack using pwntools [29] at compile and execution time. It also separates memory corruption from the later control hijacking, which allows RecIPE to differentiate data integrity, compartmentalization and control-flow integrity defenses. On the down side, the small number of attack templates utilized by RIPE and RecIPE concentrate on ROP-related attacks. Both of them suffer from a seriously limited coverage, although the numbers of generated test cases are large. Many types of control-flow hijacking attacks and temporal safety are not covered. In addition, the attack templates rely on embedded assembly for control hijacking, restricting platforms to x86 Linux only.

Instead of using full attacks, property-oriented testing programs have long been used to testing static model checking tools. One well-known example is the Juliet C/C++ and Java test suite [30], which provides 57099 (normal or error embedded) synthetic programs for model checking tools to verify their correctness and accuracy. It is found that a thorough coverage on a targeted property class can be achieved by a sufficient number of property-oriented test cases. Since Juliet has some coverage on UAF and control corruption errors while RIPE is limited to ROP-related attacks, researchers have even been borrowing test cases from Juliet to complement RIPE [31]. This is also a strong evidence showing that RIPE is insufficient.

*MSTest* is *a property-oriented, comprehensive, and cross-platform* test suite providing *a wide coverage on attack and defense capabilities*, and is capable of *identifying deployed defenses and comparing the memory safety between different platforms*. Existing memory safety test suites have unacceptable portability. They adopt the approach of embedding assembly for accurately hijacking control flow in numerous test cases [25]–[28], but this approach is clearly not portable. By abstracting platform-specific code into a share library, *MSTest is the first portable test suite capable of comparing memory safety cross 19 platforms covering well-known processors, OSes and compilers.*

In addition, existing memory safety test suites have unacceptable coverage. With 227 individual property-oriented test cases, *MSTest is the first to provide a comprehensive coverage on attack and defense capabilities*, and extra cases are constantly being added. Property-oriented testing is highly desirable when evaluating a platform's memory safety, since it allows a wider range of properties to be checked with a smaller

TABLE II
SUMMARY OF THE PLATFORMS UNDER TEST.

| Architecture | Processor | OS | Kernel | Compiler |
|---|---|---|---|---|
| Gen 8 x86-64 | i7-8550U | Ubuntu | Linux 6.8 | GCC 13.2<br>LLVM 18.0 |
| | | Windows | NT 23H2 | MSVC 19.3 |
| Gen 12 x86-64 | i7-12700 | Ubuntu | Linux 6.8 | GCC 13.2<br>LLVM 18.0 |
| | | OpenBSD | GENERIC#79 | GCC 11.4 |
| | i5-12400 | Windows | NT 23H2 | MSVC 19.3 |
| CHERI<br>Morello | QEMU<br>FVP | CheriBSD<br>CheriBSD | main-93<br>main-93 | LLVM 14.0<br>LLVM 13.0 |
| Armv8.4-A | Graviton 3 | Ubuntu | Linux 6.8 | GCC 13.2<br>LLVM 18.0 |
| Armv8.6-A | Appple M2 | Mac OS | Darwin 23 | LLVM 15.0 |
| Armv9.0-A | Tensor G3 | Android | Linux 6.1 | LLVM 17.0 |
| | Graviton 4 | Ubuntu | Linux 6.8 | GCC 13.2<br>LLVM 18.1 |
| | | OpenBSD | GENERIC#79 | GCC 11.4 |
| RV64GC | U740 | Freedom SDK | Linux 6.6 | GCC 13.2<br>LLVM 18.1 |
| | Spike | Buildroot | Linux 6.6 | GCC 13.2 |

number of test cases, compared with testing using full attacks. To the best of our knowledge, *MSTest is the first memory safety test suite conducting property-oriented testing.* Due to the relation between memory safety properties, resolving the dependency between property-oriented test cases is crucial for specifying a proper testing order and reducing testing time. *MSTest is the first to achieve dependency auto-resolving.*

## III. Threat Model

*Definition of a platform*: MSTest is designed to evaluate the provision of memory safety of a minimum running environment, namely a platform comprising a processor, an OS and a compiler. It is assumed that test cases are natively compiled using a GNU compatible make on the platform. To make the platforms under test focused, we currently limit them to the off-the-shelf ones listed in Table II. In addition, a small number of important but not yet available platforms are tested in an emulated way, including RISC-V on Spike [32], CHERI on QEMU [33] and Arm Morello [34]. We further assume that only necessary and pre-installed libraries are linked at runtime. No third-party defenses are applied to the runtime libraries or the OS. Note that these limitations do not indicate lack of portability. The test framework can easily support a new platform.

*Adversarial capability*: In the assumed attack scenario, unprivileged attackers can execute and control the inputs of user land programs running on a platform. The executed programs contain memory safety bugs in their source code. The platform adopts defenses preventing certain types of memory safety bugs from being maliciously exploited by attackers at runtime. Each defense is a mechanism enforced by parts of the platform, including the processor, the OS (runtime libraries) and the compiler. The types of memory safety vulnerabilities are limited to spatial and temporal safety. All attacks relying on micro-architectural vulnerabilities, such as transient execution attacks, resource contention in the core pipeline [35], cache side-channels and covert channels, are out of the scope. Since MSTest assumes user land attackers running on off-the-shelf platforms, attacks targeting memory safety vulnerabilities in kernel software, hypervisors, and TEEs are out of scope. In addition, memory safety vulnerabilities in the runtime for managed languages are not covered as well.

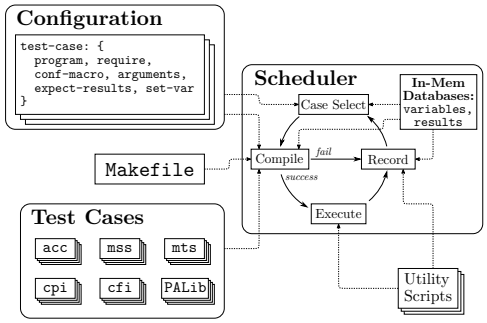



Fig. 1. The test framework of MSTest.

## IV. Test Framework

Depicted in Fig. 1, the test framework of MSTest is composed of the following components:

*Test cases*: A list of source code for individual test cases and a shared *platform abstraction library* (`PALib`). Each test case is a small self-sufficient program either exploiting a specific type of memory vulnerability or checking a detailed capability of a potential defense. Similar to RecIPE, templates are utilized to generate multiple test cases from the same source using different macros.

*Configuration*: A JSON file is used to record the parameters of each test case, such as locating the source code (`program`) for a test case, specifying dependency (`require`), defining template macros for compilation (`conf-macro`) and input arguments at runtime (`arguments`), result processing (`set-var`, `expect-results`), etc.

`Makefile` and *utility scripts*: Extra scripts used by the GNU Make for compilation (`Makefile`), and the scheduler during execution and result recording (*utility scripts*).

*Scheduler*: The central controller of MSTest. With its internal state machine, the scheduler dynamically chooses the order of testing, compiles and executes individual test cases, and handles the housekeeping work of analyzing results.

### A. Property-oriented test cases

Instead of using full attacks, MSTest chooses to use property-oriented test cases, where each case is a small self-sufficient program concentrating on evaluating a single (attack or defense related) property. Compared with full attacks, it is more likely to achieve the same coverage with less number of test cases.

To illustrate the inefficiency of a test suite consisting of full attacks, we take RIPE as an example. As shown Fig. 2, every test case in RIPE is a full attack exploiting a buffer overflow bug to alter control-flow. All test cases are generated using the same attack template with five configurable parameters: *buffer location*, *overflow technique*, *function abused*, *pointer type* and

*target type*. By choosing different combinations of these five parameters, RIPE uses in total of 850 test cases to cover only a portion of control-flow hijacking attacks (mostly ROP). This is obviously inefficient [28]. A successful ROP attack relies on the exploitation of three vulnerabilities: *buffer overflow*, *code-pointer corruption* and *control-flow hijacking*. There is dependency between these vulnerabilities but such dependency might be indiscriminate, e.g. code-pointer corruption depends on the possibility to exploit a buffer overflow but not selective on the exact type: overflow or underflow by spray or pointer. As long as there is a viable way to corrupt a specific return address (RA) using one type of buffer overflow, and this RA can indeed alter control flow, it is sufficient to conclude that the platform under test is vulnerable to ROP attacks. There is no need to re-examine the same ROP attack by exhausting all possible ways of corrupting this RA. As a result, RIPE's exhaustive way of testing leads to a large number of partially duplicated test cases.

MSTest provides a similar coverage by using a significantly smaller number of test cases. It evaluates the three vulnerabilities separately. For each vulnerability, MSTest has multiple test cases where each of them tests a specific way of exploitation or defense, which is considered as a *property* of the platform.[1] There is dependency between properties, e.g. returning to a ROP gadget relies on successfully revising a RA to the entry of this gadget. This dependency is recorded by parameter `require` in the configuration. As later discussed in Section IV-B, the scheduler relies on this dependency to specify the order of testing.

In order to ensure that the suite is not missing important vulnerability classes or defense properties, we have manually analyzed the RIPE, RecIPE, and Juliet at the early design stage of this benchmark. We have carefully ensured that the vulnerabilities covered in the analyzed benchmarks are also covered by MSTest. From the suite maintenance point of view, whenever a new attack emerges, the new vulnerabilities utilized in this attack need to be identified. Corresponding test cases should then be added to MSTest to cover them. Based on the manual analysis, MSTest currently evaluates six categories of properties[2]. Due to the limit space, a more detailed description of all test cases is provided in our code repository [36].

- *Access capability* (`acc`, 22 cases): Whether an attacker can read critical information from memory.
- *Memory spatial safety* (`mss`, 122 cases): Whether an attacker can read or write memory by a way violating memory spatial safety.

[1]When we test a specific way of exploitation, the corresponding safety (e.g., integrity) property tells whether the platform is immunized from the specific way of exploitation.

[2]There is currently no consensus on how to categorize the safety properties; e.g., code-pointer integrity might be considered as a part of control-flow integrity and many properties being tested under access capability may not be considered as vulnerabilities. The presented classification is a pragmatic one suiting the operation of the test suite as we consider an attack would typically target a key pointer (code-pointer integrity) and alter it using a memory spatial or temporal vulnerability selected according to the environment (access capability), before eventually hijacking the control flow.

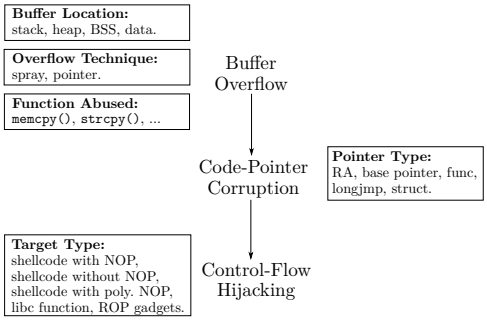



Fig. 2. The test template and parameters of RIPE.

- *Memory temporal safety* (`mts`, 16 cases): Whether an attacker can access data which is not allocated or already de-allocated.
- *Code-pointer integrity* (`cpi`, 10 cases): Whether an attacker can read or write critical code-related pointers.
- *Backwards control-flow integrity* (`cfi-b`, 17 cases): Whether an attacker can maliciously alter the backwards control flow.
- *Forwards control-flow integrity* (`cfi-f`, 40 cases): Whether an attacker can maliciously alter the forward control flow.

Most test cases are composed of a set-up of a vulnerable memory layout, a straight-forward and hard-coded violation of memory safety, and a result feedback. A simplified version of test case `return-to-wrong-call-site` is described in Listing 1. During the call of `helper()` on line 22, its RA is hijacked to `TARGET_LABEL`, which is the return site of `helper_useless()` instead of `helper()`. The definition of the two helper functions (line 5–13) and the calling sequence in `main()` (line 18–23) set up the vulnerable (code) memory layout, while the attack behavior (line 9,10) in `helper()` forcefully alters the RA on stack. The global variable `gvar` is used to track the progress of the test case. If the control-flow is successfully hijacked, 0 is returned by `exit()` on line 26.

This test case concentrates on whether the modified RA can hijack the control-flow. It is normally tested after other tests confirming that RA can be modified on stack. Therefore, the modification of RA is straightforward, either by a stack pointer (SP) indexed write using a macro `MOD_STACK_DAT(label, offset)` defined in `PALib` or a simple buffer overflow (line 10), depending on an input argument `sel` (line 18). The `offset` is also an input argument (line 19) specifying the distance between RA and SP/`buf`, while `label` is the address of the wrong return site (line 20).

The return value of a test case (by either `return` or `exit()`) identifies the exploitable status of a property. 0 denotes that the property is exploitable, while others are used to indicate the potential reason for the failed exploitation. Normally a test case can fail in four scenarios: *a compilation failure* as the compiler potentially identifies the vulnerability, *an expected exception or non-zero return value* as the exploitation is potentially prevented by a known defense, *an*

Listing 1
RETURN-TO-WRONG-CALL-SITE.CPP

```
#include "include/assembly.hpp"
long long offset;
int gvar;

void FORCE_NOINLINE helper(void *label, int sel) {
  gvar = 3;
  void * buf[2];
  COMPILER_BARRIER;
  if(sel) MOD_STACK_DAT(label, offset); // from PALib
  else    *(buf+offset) = label;
  for(auto b:buf) b = (void *)&gvar;    // enforce stack
      allocation
  gvar = 0;
}

void FORCE_NOINLINE helper_useless() { gvar = 4; }

int main(int argc, char* argv[]) {
  int sel = std::stoi(argv[1]);  // from configuration
  offset  = std::stoll(argv[2]); // from configuration
  void *ret_label = &&TARGET_LABEL;
  if(offset == -1) { goto *ret_label;} // confuse
     compiler
  helper(ret_label, sel); // actual attack
  helper_useless(); // never called
  COMPILER_BARRIER;
TARGET_LABEL:
  exit(gvar); // the wrong return site
}
```

Listing 2
CONFIGURATION EXAMPLES.

```
"write-cross-obj-index-overflow-stack": {
  "program": "write-cross-obj-index",
  "require": { "0":[["write-intra-array-index-overflow-
     stack"]]},
  "conf-macro": {"0": {"BUFFER_SIZE":"8", "REGION_KIND":
     "0","BUFFER_KIND":"1"} },
  "arguments": { "0": "0" },
  "expect-results": {"290": "CHERI enabled."}
},

"return-to-wrong-call-site": {
  "require": {
    "0": [["get-ra-offset"], ["write-by-stack-pointer"
     ]],
    "1": [["write-by-stack-pointer"]]
    "2": [["write-cross-obj-index-overflow-stack"]]},
  "arguments": {
    "0": "1 $stack-offset",
    "1": "1 [0:1:16]",
    "2": "0 [2:1:32]"},
  "set-var": {
    "0": [],
    "1": ["stack-offset"],
    "2": ["buf-offset"]},
  "retry-results": {
    "1"  : "failed to modify RA",
    "267": "segment error: partially written RA",
    "251": "Arm ASan enabled"},
  "expect-results": {"290": "CHERI enabled."}
}
```

*unexpected runtime error* as the memory might be messed up and the test fails randomly, and *an unexpected non-zero return value* as the exploitation fails silently. The first two scenarios are considered as strong evidence of effective defenses. MSTest adds its own exception handler to convert specific exception types into predefined return values, and use these return values to identify individual defenses. For the other two scenarios, the behavior of the test case is unexpected and is reported for further investigation.

### B. Test configuration

The parameters of each test case is stored in a JSON file. A rich set of parameters are supported by MSTest. We use Listing 2 as an example to show the important ones. It specifies a cross-object buffer overflow test (line 1–7) and the previous ROP test described in Listing 1.[3]

Let us first discuss the overflow test. If the source code does not share the name of the test case, it is located by parameter `program` (line 2). Parameter `require` (line 3) defines the dependency between test cases. A test case can be executed when one of the pre-defined scenarios is satisfied. The dependency in the example is simple. There is only one scenario (case 0) requiring that intra-object overflow has been confirmed exploitable. Similar to RecIPE, a template can be used to generate multiple cases using different macros at compile-time. In this case, `BUFFER_SIZE=8`, `REGION_KIND=0` and `BUFFER_KIND=1`, are defined by parameter `conf-macro` as a list. It is common that a test case needs runtime arguments, which are defined by parameter `arguments` (line 7). Parameter `expect-results` records all expected non-zero return values with their explanations; e.g., return value 290[4] confirms that this attack is failed by CHERI (Morello).

The configuration for `return-to-wrong-call-site` is more complicated. There are three scenarios defined by parameter `require`. For each scenario, the dependency is described as a product of maxterms; e.g., $[[A, B], D, [E, F]]$ represents a Boolean logic $(A + B) \cdot D \cdot (E + F)$, where each Boolean variable is a test case and counted true when the case returns 0 (exploitable). Scenarios are checked in the defined order. In this case, scenario 0 is chosen when the distance between RA and SP is obtainable (`get-ra-offset`) and SP-indexed write is possible on stack (`write-by-stack-pointer`). If test case `get-ra-offset` fails to locate RA on stack, the test case falls back to blind testing. It tries to modify RA by SP-indexed write (scenario 1) if test case `write-by-stack-pointer` still returns 0. Otherwise, generic buffer overflow (scenario 2) can be used as a fallback, if test case `write-cross-obj-index-overflow-stack` succeeds with 0. The selection of scenarios affects all steps in running the test cases, including `conf-macro`, `arguments` and `set-var`. As shown in line 14–17, arguments are individually defined for each scenario. In scenario 0, the 2nd argument is actually a variable (`$stack-offset`) stored in an in-memory variable database managed by the scheduler (as shown in Fig. 1). All variables in this database are set by individual test cases. In this case, variable `$stack-offset` should have been initialized by test case `get-ra-offset`. The scheduler is

[3]For simplicity, long names are reduced, and complex dependency conditions are simplified, while the core idea remains.

[4]Actually this is an exception caught by the test case and converted to this specific return value.

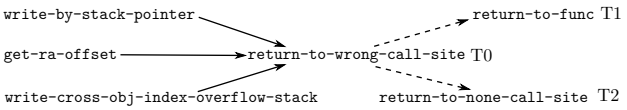



Fig. 3. Dependency related to `return-to-wrong-call-site`.

responsible for parsing the argument and replacing variables with their values stored in the database. In scenarios 1 and 2, without knowing the correct `offset`, the test case blindly runs multiple times by using a different `offset` in each trial. The range of this `offset` is defined directly in the argument and parsed by the scheduler; e.g., `[2:1:32]` denotes ranging from 2 to 32 using a step size of 1. If the test fails, it retries if the return value falls in the pre-defined list `retry-results`. When the test fails with an unknown return value or the range is exhausted, the test case finished with the return value of the last trial. When a test case succeeds (returning 0), it may set up specific variables in the in-memory database as defined by parameter `set-var`. For this test case, it sets up variable `stack-offset` or `buf-offset` depending on the chosen scenario.

### C. Relation Graph

Let us expand the dependency described in Listing 2 a little bit further by exposing both the prerequisites and the dependents (T1 and T2) of test case `return-to-wrong-call-site` (T0) using a directed acyclic graph depicted in Fig. 3.[5] The test suite considers two types of dependency: *utilization* and *relaxation*. Utilization describes the requirement of utilizing one property in the testing of another property. In Fig. 3, all prerequisites of test case `return-to-wrong-call-site` are utilization dependency and depicted in solid arrows. Rather than specifying the utilization relationships between two different properties, the relaxation type of dependency sets up an order for testing similar properties. In Fig. 3, the three ROP test cases (T0, T1 and T2) are similar, but T0 is the most evasive case. Hijacking RA to a none-call-site gadget (T1) or a function entry point (T2) is more likely to be detected by a coarse-grained CFI defense than hijacking RA to a valid call site (T0). Since T1 and T2 provide more freedom to attackers, they are considered as relaxed from T0. If T0 is tested failed, it is certain that T1 and T2 would not succeed and there is no need to test them. Therefore, evasive cases are tested before their relaxed versions. This dependency is also described in parameter `require` and depicted in a dashed arrow in Fig. 3.

Please note that the dependency defined in parameter `require` literally specifies the scenarios when a test case can be run. It is related to but different from the dependency utilized in real-world attacks. Dependency potentially utilized in a full attack may not be described in MSTest. For example, a ROP attack may corrupt RA using a heap pointer obtained by UAF; however, such dependency is not specified. MSTest is a property-oriented test suite which does not exhaustively

[5]This is for illustration only. Please see the actual dependency of case <173>defined in the document for more details.

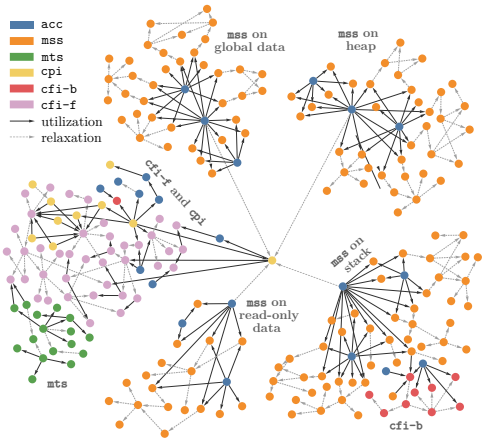



Fig. 4. Relation graph of dependency between test cases.

test all attack combinations as properties are tested individually. MSTest has dedicated test cases to check whether long distance write from heap to stack is possible (case <122,123> as listed in Section I of [36]). Instead of depending on all possible types of overflow, ROP-related test cases (`cfi-b`) depend on SP-indexed write on stack as it is one of the most evasive ways of corrupting RA. In the rare situation where SP-indexed write on RA is indeed prevented (e.g. LLVM's HWASan), a generic buffer overflow on stack is specified as a fallback. This is considered sufficient, as none of the platforms under test has successfully prevented both.

By connecting all test cases using the dependency specified arrows and clustering them according to connections, Fig. 4 presents the relation graph for all test cases. There are five visible clusters of test cases, where four of them are memory spatial safety cases (`mss`) targeting the four memory regions: stack, heap, read-only data and global data, respectively. As the ROP-related cases (`cfi-b`) need to corrupt RA on stack, they are naturally associated with `mss` on stack. The forward CFI-related cases (`cfi-f`) are associated with `CPI` cases (`cpi`), because hijacking forward control-flow normally needs to corrupt function pointers. Finally, memory temporal safety cases (`mts`) are connected to `cfi-f` because an attack may hijack a call to a virtual function of a released object. There are two interesting observations from the relation graph: Clusters are connected by memory access capability cases (`acc`) and `cpi` cases, which is understandable as most attacks depend on capabilities to access and corrupt critical data/pointers in memory. Almost all cross-category arrows are utilization dependency, while intra-category arrows are usually relaxation dependency.

MSTest is the first test suite to actively maintain this dependency as it brings two major advantages: *It automatically regulates the order of testing.* With the help of this dependency, running the test suite becomes a traverse problem of the relation graph. It always starts with test cases with zero indegree and finishes with test cases with zero outdegree. A test case is scheduled when all its preceding test cases are executed. Since the graph is acyclic, it is guaranteed that all test cases are executed once and only once. *It reduces the*

*time of testing*. When a test case can be scheduled as all its preceding test cases in the relation graph have been run, the execution of this case can be skipped if no scenario pre-defined in parameter `require` is satisfied. This case is then inferred failed and it is likely that all test cases depended on this one are inferred failed as well. According to our results, this can significantly reduce the time of testing by up to 40%.

### D. Test scheduler

Test scheduler is the central controller of MSTest. It runs MSTest in either a *fast-run* mode, where test cases are scheduled according to the relation graph and skipped if inferred failed, or an *exhausted-run* mode, where all test cases are tested.

Two in-memory databases are maintained by the scheduler: a *variable* and a *result* databases. As mentioned in Section IV-B, when a test case succeeds, it updates the variables defined by parameter `set-var`, which are maintained in the variable database and used by other test cases according to parameter `arguments`. The return values of all tested cases are recorded in the result database, which is used by the scheduler to check whether the pre-defined scenarios of a test case are satisfied. After running the entire test suite, this result database is analyzed and formated into a test report.

The scheduler is responsible for compiling (by launching GNU Make) and executing (by spawning a process) each test case. Note that the scheduler records a failed compilation as a defense. After a successful compilation, the test case may finish execution with a return value or throw an exception. The scheduler handles all catchable exceptions, translates known exceptions into expected return values, relaunches the test case if a retry is needed, and finally records the return value into the result database.

### E. Fight with compiler optimization

We choose to use self-attacks rather than the generated attacks adopted by RecIPE, as the latter rely on a separate tool (pwntools [29]) for analyzing the compiled binary and generating exploitation, which both limits the range of testable properties and reduces portability. However, self-attacks unavoidably expose the attack behavior to compilers. A compiler may obey the program and compile the attack as normal, or notice the attack and refuse to compile (as a way of defense), both of which are fine. However, modern compilers are capable of making aggressive optimizations which may silently disarm the embedded attack, which is undesirable for our use-case. For this reason, we have spent quite some effort to exam the test cases failing with unexpected return values, and carefully revise them to avoid undesirable compiler optimization.

Several strategies are utilized in MSTest. When a certain attack behavior is prevented by a compiler, it can be implemented using a snippet of embedded assembly, because most compiler would not try to analyze or optimize embedded assembly. It is common for a compiler to reorder operations when they are seemly independent. Since assembly implemented attack behaviors are hidden from compilers, the seemly safe reordering of operations may break an attack assumption. In this case, explicit compiler barriers are added to prevent such reordering, such as line 8 and 24 in Listing 1. Another common optimization utilized by modern compilers is constant propagation and result prediction. However, some attacks may revise the seemingly constant variables, especially when such revision occurs in embedded assembly. To stop compilers from predicting the values of attack related variables, the assignments of these variables are delayed to runtime using arguments, and some related class/function definitions are moved to shared libraries. Compilers are also found to be aggressive in removing dead code. For example, it may remove unreachable instructions according to its control-flow prediction, replace a (virtual) function call by a simple assignment if the function is deemed pure, or avoid allocating a stack frame for a non-leaf function if the operation on local variables is simple enough. All of these can unintentionally break attacks. In Listing 1, a fake jump is added on line 21 to guarantee label `TARGET_LABEL` is defined, and useless buffer operations are added on line 11 to ensure `buf` is always allocated on stack.

### F. Support for portability

Portability is one of the major design goals in the construction of MSTest. To fairly compare memory safety between platforms, MSTest must be ported to multiple platforms and run in a similar way. In addition, there is no doubt that new vulnerabilities will be discovered and new defenses will be adopted. MSTest shall allow new test cases to be easily added. To achieve this goal, MSTest separates test cases into platform-independent source code and a *platform abstraction library* (PALib). All platform specific (C/C++ or assembly) operations, calls to library functions, compiler intrinsics and exception handlers are packed into PALib, while the remaining source code in individual test cases is platform-independent and easily portable.

Assembly code is a major part of PALib. As mentioned in Section IV-E, embedded assembly is used to circumvent compiler optimizations. Some attack behaviors are also easier to implement in assembly compared to C/C++, such as revising a variable on stack using SP as the base pointer (line 9 in Listing 1). However, assembly code is obviously difficult to port between architectures. To resolve this problem, we carefully regulate the use of assembly and extract a small set of only 12 assembly macros and 1 assembly function (as described in Section III of [36]) as an interface shared by all test cases. All test cases include the same assembly header (`include/assembly.hpp`) as on line 1 in Listing 1. The correct assembly implementation is automatically chosen by the compiler using compiler pre-defined macros, as shown in Listing 3. A new implementation of these assembly code snippets is usually required when MSTest is ported to a new architecture, but no modification is needed for test cases utilizing these assembly snippets.

Modern compilers usually provide builtin functions and intrinsics for low-level functionalities. Whenever such functionalities are required, MSTest prefers to use them rather than make its own (assembly) implementation for better portability. One example is on line 20 in Listing 1, where the

Listing 3
CHOOSE THE RIGHT ASSEMBLY CODE.

```
1 // part of include/assembly.hpp
2 #if defined(__x86_64) || defined(_M_X64)
3   #if defined(_MSC_VER)
4     #include "x86_64/MSVC/assembly.hpp"
5   #else
6     #include "x86_64/POSIX/assembly.hpp"
7   #endif
8 #endif
```

address of label `TARGET_LABEL` is obtained using GCC's own `&&` operator (LLVM compatible). Macros are defined for common compiler directives as well, such as the macro `FORCE_NOINLINE` on line 5 and 15 in Listing 1 tells compiler to not inline the two helper functions.

Special attention is given to exception handlers. For a number of test cases with known defense-raised exceptions, MSTest implements a shared exception handling system that catches exceptions, verifies their correctness and translates known exceptions into pre-defined return values (defined by parameter `expect-results`). This is one of the major reasons why MSTest is capable of detecting and identifying the applied defenses.

In all ported platforms, x86-64+Windows+MSVC is the most difficult so far. MSVC's own NMake is incompatible with GNU Make. MSVC prohibits the use of embedded assembly for x86-64 and its support for POSIX is not strictly compatible. As a result, we have to re-implement the whole assembly part using MSVC's intrinsics, replace the POSIX exception handling system by Windows's vector exception handling, and require the installation of a GNU compatible make as a compromise. After all, our test suite is the first to achieve a fair comparison of memory safety between Windows and Linux/BSD platforms.

## V. EVALUATION

As described in Section III, the evaluation in this paper is focused on the commercially available platforms listed in Table II. Without further notice, each platform is tested with the default compiler that comes with the official OS distribution. Intel 8th and 12th gen processors running both Ubuntu 24 and Windows 11 are tested due to their different support of Intel CET. Multiple Arm processors from different vendors have been evaluated, including Apple, Amazon, and Google. Regarding RISC-V, we have tested Freedom U740 [37], along with Spike [32], the golden model used to standardize all new ISA extensions of RISC-V. We have also tested Morello [34], a security oriented architecture newly released from Arm based on the CHERI architecture [10] designed by the University of Cambridge. As the Morello dev board is unavailable outside the UK, the tests are performed by emulation (FVP [38] for Morello and QEMU for CHERI). Also in this section, the test cases related to important findings are labeled by case IDs (as listed in Section I and II of [36]) in angle brackets.

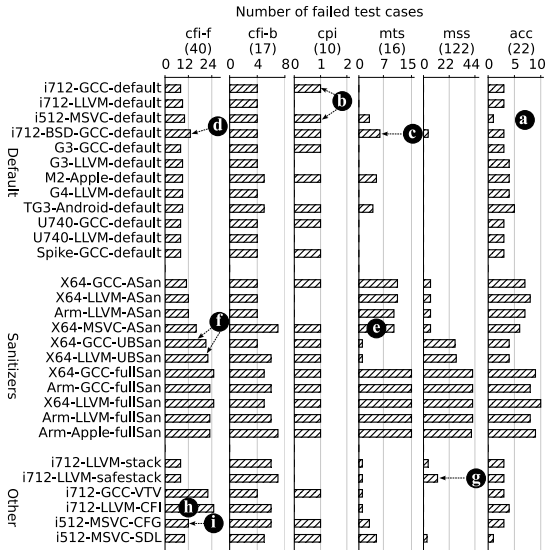



Fig. 5. The evaluation of software defenses on commercially available platforms. (The total number of test cases in each category is labeled in parentheses within the y-axis.)

### A. Software defenses available on commercially platforms

In this section, we evaluate the software implmented defenses that requires no special ISA extensions and are directly available with the default compilers by setting extra compiler flags. To provide a baseline, we would first measure the memory safety of various platforms using the default compilers with no security related compiler flags. The details of the platforms under test and the corresponding compiler flags are described in Table III with each platform labeled using an abbreviation listed in the first column for simplicity. The test results are shown in Fig. 5.

**Default:** In general, the protection of memory safety is weak on most platforms with the default compiler flags. As depicted in Fig. 5, wide-deployed protections, such as ASLR, W⊕E, RELRO (RELocation Read-Only) [3] and stack canary, contribute to only limited defense. Both code and stack segments are relocated by ALSR at the start of a new process on Linux but they are relocated only at boot time on Windows <14,15> ⓐ. Direct code injection is prevented by W⊕E on all platforms. GOT hijacking [39] can be stopped by full RELRO, as enabled by GCC and MSVC, but only partial RELRO is enforced by LLVM by default <21> ⓑ. Stack canary is added for functions operating strings on stack but disabled by default for others <17,18>. It should be noted that RISC-V platforms currently offer the worst level of protection, as they support only the protections mentioned above.[6]

As shown in Fig. 5, i512-MSVC-default, M2-Apple-default, i712-BSD-GCC-default and TG3-Android-default are safer than others thanks to their extra defenses enabled by default. Their enhanced memory allocators prevent attackers from

[6]RISC-V currently has the worst support for memory safety in all architectures tested in this paper. Sv48 and Sv57 are not supported by ASan [40]. GCC's VTV is not ported to RISC-V. Although pointer masking (PM), shadow stack (Zicfiss), and landing pad (Zicfilp) have been rectified, they are not fully supported by the Linux kernel at the time of writing of this paper [41]. There would be no further discussion of RISC-V in the rest of this paper.

TABLE III
THE CONFIGURATION OF SOFTWARE DEFENSES ON COMMERCIALLY AVAILABLE PLATFORMS.

| Abbr. | Arch. | Processor | Kernel | Compiler | Default/Extra compiler Flags |
|---|---|---|---|---|---|
| i712-GCC-default | x86-64 | i7-12700 | Linux 6.8 | GCC 13.2.0 | `-O2 -std=c++11 -Wall` |
| i712-LLVM-default | x86-64 | i7-12700 | Linux 6.8 | LLVM 18.0.0 | `-O2 -std=c++11 -Wall` |
| i512-MSVC-default | x86-64 | i5-12400 | Windows 23H2 | MSVC 19.38 | `/O2 /Gd /std:c++11 /W3 /WX- /EHsc` |
| i712-BSD-GCC-default | x86-64 | i7-12700 | GENERIC#79 | GCC 11.4.0 | `-O2 -std=c++11 -Wall` |
| G3-GCC-default | Armv8.4-A | Graviton 3 | Linux 6.8 | GCC 13.2.0 | `-O2 -std=c++11 -Wall` |
| G3-LLVM-default | Armv8.4-A | Graviton 3 | Linux 6.8 | LLVM 18.0.0 | `-O2 -std=c++11 -Wall` |
| M2-Apple-default | Armv8.6-A | Apple M2 | Darwin 23.2.0 | LLVM 15.0.0[a] | `-O2 -std=c++11 -Wall` |
| G4-LLVM-default | Armv9.0-A | Graviton 4 | Linux 6.8.0 | LLVM 18.1.3 | `-O2 -std=c++11 -Wall` |
| TG3-Android-default | Armv9.0-A | Tensor G3 | Android | LLVM 17.0.2[b] | `-O2 -std=c++11 -Wall` |
| U740-GCC-default | RV64GC | Free. U740 | Linux 6.6.21 | GCC 13.2.0 | `-O2 -std=c++11 -Wall` |
| U740-LLVM-default | RV64GC | Free. U740 | Linux 6.6.21 | LLVM 18.1.0 | `-O2 -std=c++11 -Wall` |
| Spike-GCC-default | RV64GC | Spike | Linux 6.6.2 | GCC 13.2.0 | `-O2 -std=c++11 -Wall` |
| X64-GCC-ASan | x86-64 | i7-12700 | Linux 6.8 | GCC 13.2.0 | `-fsanitize=address` |
| X64-LLVM-ASan | x86-64 | i7-12700 | Linux 6.8 | LLVM 18.0.0 | `-fsanitize=address` |
| X64-MSVC-ASan | x86-64 | i5-12400 | Windows 23H2 | MSVC 19.38 | `/fsanitize=address` |
| X64-GCC-UBSan | x86-64 | i7-12700 | Linux 6.8 | GCC 13.2.0 | `-fsanitize=undefined` |
| X64-LLVM-UBSan | x86-64 | i7-12700 | Linux 6.8 | LLVM 18.0.0 | `-fsanitize=undefined` |
| X64-GCC-fullSan | x86-64 | i7-12700 | Linux 6.8 | GCC 13.2.0 | `-fsanitize=address,pointer-compare,undefined,` `pointer-subtract,unsigned-integer-overflow,` `signed-integer-overflow` |
| Arm-GCC-fullSan | Armv8.4-A | Graviton 3 | Linux 6.8 | GCC 13.2.0 | |
| X64-LLVM-fullSan | x86-64 | i7-12700 | Linux 6.8 | LLVM 18.0.0 | |
| Arm-LLVM-fullSan | Armv8.4-A | Graviton 3 | Linux 6.8 | LLVM 18.0.0 | |
| Arm-Apple-fullSan | Armv8.6-A | Apple M2 | Darwin 23.2.0 | LLVM 15.0.0 | |
| i712-LLVM-stack | x86-64 | i7-12700 | Linux 6.8 | LLVM 18.0.0 | `-fstack-protector-all -fstack-clash-protection` |
| i712-LLVM-safestack | x86-64 | i7-12700 | Linux 6.8 | LLVM 18.0.0 | `-fsafestack` |
| i712-GCC-VTV | x86-64 | i7-12700 | Linux 6.8 | GCC 13.2.0 | `-fvtable-verify=std` |
| i712-LLVM-CFI | x86-64 | i7-12700 | Linux 6.8 | LLVM 18.0.0 | `-fsanitize=cfi -fvisibility=hidden -flto -static` |
| i512-MSVC-CFG | x86-64 | i5-12400 | Windows 23H2 | MSVC 19.38 | `/Od /guard:cf /link /RTCs` |
| i512-MSVC-SDL | x86-64 | i5-12400 | Windows 23H2 | MSVC 19.38 | `/sdl /GS` |

*a*. Apple's LLVM 15.0.0
*b*. Android LLVM cross compile toolchains.

easily reclaiming previous freed objects by double-free <144–146>. Both M2-Apple and TG3-Android fail extra UAF read in heap cases as they proactively clear objects after free <153,154,157>. M2-Apple enforces a strict scope check that prohibits direct jump from switch clause to outside <220>. It also endeavors to avoid spilling return addresses onto the stack when there are spare registers <181>, reducing the number of available ROP gadgets. OpenBSD has the strongest protection for temporal safety ⓒ. It proactively detect write-after-free on heap by rewriting objects when they are freed and checking whether they are intact in a postponed memory recycle <157,159>. Canaries are inserted not only on stack but also between heap objects <101,103,105>. Furthermore, OpenBSD enables IBT (part of Inte CET) in both kernel and userland by default, which reduces code gadgets available to attackers ⓓ.

**Sanitizer:** Sanitizers are powerful tools to detect existing memory errors in software. They provide strong safety enforcement, though usually considered as debugging or fuzzing tools rather than production-ready defense [42]. We use the 12th gen x86-64 platforms for comparing the individual sanitizers available in most compilers, while all sanitizers shipped with the default compilers are enabled when comparing between platforms.

Two sanitizers are widely available in most compilers: Address sanitizer [8] (ASan) and undefined behavior sanitizer (UBSan). ASan detects out-of-bound spray and overflow/underflow attacks by inserting redzones into stack, global data, and read-only data regions, but it is unable to safeguard against intra-object overflows <54–65>, pinpointed overflows bypassing redzones <38–45>, and overflows caused by type conversions <136–143>. Test results also show that ASan implements quarantines to detect UAF on heap <153,154> and uses shadow memory to prevent use-after-return (UAR) on stack <147,151> (unavailable in MSVC ASan ⓔ). UBSan consists of several small sanitizers. It stops all overflow accesses using out-of-bound indices, but using out-of-bound pointers remains possible <67–72>. It even stops all COOP test cases (`cfi-f`) except for those reusing VTables of sibling or parent classes <201,202> ⓕ. MSVC notably lacks the support for UBSan.

When all sanitizers shipped with the default compilers are enabled,[7] some differences between platforms are observed. The extra leak sanitizer enabled on x86-64 (X64-GCC-fullSan, X64-LLVM-fullSan) successfully prevents some forward control hijacking attacks from tampering stack-related registers <187,189,220>. The additional align check of Arm's UBSan makes extra trouble for attackers to fake stack frames <182>.

**Other software protection:** Modern compilers have adopted multiple software defenses in addition to sanitizers. Stack protection and safe stack are two of such defenses concentrating on protecting the stack. Compared with i712-LLVM-default, stack protection on i712-LLVM-stack stops 4 `mss` test cases issuing stack out-of-bound accesses <67,70,88,89> and 2 `cfi-b` test cases faking stack frames <181,182>. Safe stack stops 8 extra `mss` test cases related to pinpointed overflows on the stack <38–45> ⓖ. Compilers tend to have their own control-flow protection, such as GCC

[7]To enable the widest set of sanitizers provided by a certain compiler, we manually enabled all the flags described in the man page, and then removed the minority flags incompatible with others through multiple rounds of compiling and testing.

TABLE IV
THE CONFIGURATION OF HARDWARE DEFENSES ON COMMERCIALLY AVAILABLE PLATFORMS.

| Abbr. | Arch. | Processor | Kernel | Compiler | Extra Compiler Flags |
|---|---|---|---|---|---|
| i78-GCC-CET | x86-64 | i7-8550 | Linux 6.8 | GCC 13.2.0 | `-fcf-protection=full` |
| i78-MSVC-CET | x86-64 | i7-8550 | Windows 23H2 | MSVC 19.38 | `/link /CETCOMPAT` |
| i512-MSVC-CET | x86-64 | i5-12400 | Windows 23H2 | MSVC 19.38 | `/link /CETCOMPAT` |
| i712-GCC-CET[a] | x86-64 | i7-12700 | Linux 6.8 | GCC 13.2.0 | `-fcf-protection=full` |
| i712-BSD-GCC-CET | x86-64 | i7-12700 | OpenBSD#79 | GCC 11.4.0 | |
| G3-GCC-BTI | Armv8.6-A | Graviton 3 | Linux 6.8 | GCC 13.2.0 | `-mbranch-protection=bti` |
| M2-GCC-BTI | Armv8.6-A | Apple M2 | Darwin 23.2.0 | GCC 11.2.0 | `-mbranch-protection=bti` |
| G4-GCC-BTI | Armv9.0-A | Graviton 4 | Linux 6.8 | GCC 13.2.0 | `-mbranch-protection=bti` |
| G4-LLVM-BTI | Armv9.0-A | Graviton 4 | Linux 6.8 | LLVM 18.0.0 | `-mbranch-protection=bti` |
| G4-BSD-GCC-BTI | Armv9.0-A | Graviton 4 | OpenBSD#79 | GCC 11.4.0 | |
| TG3-LLVM-BTI | Armv9.0-A | Tensor G3 | Android | LLVM 17.0.2 | `-mbranch-protection=bti` |
| G3-GCC-PA | Armv8.4-A | Graviton 3 | Linux 6.8 | GCC 13.2.0 | `-march=armv8.5-a+pauth -mbranch-protection=pac-ret` |
| G3-LLVM-PA | Armv8.4-A | Graviton 3 | Linux 6.8 | LLVM 18.0.0 | `-march=armv8.5-a+pauth` |
| G4-GCC-PA | Armv9.0-A | Graviton 4 | Linux 6.8 | GCC 13.2.0 | `-march=armv8.5-a+pauth` |
| M2-Apple-PA | Armv8.6-A | Apple M2 | Darwin 23.2.0 | LLVM 15.0.0 | `-arch arm64 -fptrauth-indirect-gotos...`[b] |
| TG3-LLVM-PA | Armv9.0 | Tensor G3 | Android | LLVM 17.0.2 | `-march=armv8.5-a+pauth` |
| M2-Apple-MTE | Armv8.6-A | Apple M2 | Darwin 23.2.0 | LLVM 15.0.0 | `-march=armv8.5-a+memtag` |
| G3-GCC-MTE | Armv8.4-A | Graviton 3 | Linux 6.8 | GCC 13.2.0 | `-march=armv8.5-a+memtag` |
| G4-GCC-MTE | Armv9.0-A | Graviton 4 | Linux 6.8 | GCC 13.2.0 | `-march=armv9.0-a+memtag` |
| TG3-LLVM-MTE[c] | Armv9.0-A | Tensor G3 | Android | LLVM 17.0.2 | `-march=armv8.5-a+memtag` |
| G3-GCC-TBI | Armv8.4-A | Graviton 3 | Linux 6.8 | GCC 13.2.0 | `-fsanitize=hwaddress` |
| G3-LLVM-TBI | Armv8.4-A | Graviton 3 | Linux 6.8 | LLVM 18.0.0 | `-fsanitize=hwaddress` |
| G4-GCC-TBI | Armv9.0-A | Graviton 4 | Linux 6.8 | GCC 13.2.0 | `-fsanitize=hwaddress` |
| CHERI/Morello-conservative | CHERI/Morello[d] | CheriBSD/Linux 6.5.0 | | LLVM 14.0.0 | -O2 -fuse-ld=lld |
| CHERI/Morello-no-revoke | CHERI/Morello[e] | CheriBSD/Linux 6.5.0 | | LLVM 14.0.0 | -O2 -fuse-ld=lld |
| CHERI/Morello-strong | CHERI/Morello | CheriBSD/Linux 6.5.0 | | LLVM 14.0.0 | `-cheri-bounds=everywhere-unsafe` |

*a*. Set environment variable `GLIBC_TUNABLES=glibc.cpu.hwcaps=SHSTK`.
*b*. `-fptrauth-intrinsics -fptrauth-returns -ftrivial-auto-var-init-skip-non-ptr-array -fptrauth-vtable-pointer-type-discrimination -fptrauth-vtable-pointer-address-discrimination -fptrauth-calls`
*c*. The 3rd gen Google Tensor SoC, marketed as "Google Tensor G3".
*d*. Extra default flags for CHERI: `-mno-relax -march=rv64gcxcheri -mabi=l64pc128d`; and Morello: `-march=morello -mabi=purecap`.
*e*. CHERIvoke is disabled in kernel.

VTV [17], LLVM CFI [17], and MSVC control flow guard (CFG) [43]. VTV behaves exactly the same as UBSan in preventing COOP, although it requires recompiling runtime libraries with `--enable-vtv`. Arguably, LLVM CFI provides the strongest CFI enforcement by enforcing a fine-grained CFI with type verification on function arguments and pointer casting ⓗ. However, it requires all classes to be link-time visible, which breaks its support for dynamic linking. CFG provides forward CFI by recording and verifying jump/call targets using a bitmap, but it fails to verify VTable pointers <205> ⓘ. Last but not least, MSVC's security development lifecycle checks (i712-MSVC-SDL) detect two UAF cases using stale pointers <147,151> and one ROP case faking a stack frame <182>.

### *B. Hardware defenses available on commercially available platforms*

This section evaluates the hardware defenses (ISA extensions) directly available (without installing other libraries) on each platform, including Intel CET [19], Arm BTI, Arm TBI, Arm PA [22], Arm MTE [21], and CHERI [10]. Platforms are listed in Table IV and the test results are depicted in Fig. 6. Most of these defenses are enabled by extra compiler flags while some require extra kernel or environment tweaks. This section also tries to answer *Q1* and *Q2* raised in the Introduction. From the test results, we can clearly observe whether a hardware defense is enabled for user programs and what level of protection is provided.

**Intel CET:** As a control-flow security extension released in 2016, Intel CET has been supported by compilers since GCC 11 [44] and LLVM 6.0 [45]. It consists of two components: Shadow stack (SHSTK) and indirect branch target (IBT). According to the results shown in Fig. 6 (X64), Intel CET has no effect on the 8th gen as it is not supported. On the 12 gen processors, SHSTK is supported by both Linux (requiring recompiling glibc with `--enable-cet` [46]) and Windows [45] while IBT is supported by OpenBSD. None of the OSes supports both. SHSTK with GCC (i712-GCC-CET) stops almost all ROP-related attacks, including the most evasive one <173> that hijacks the return to a wrong return site complying with static control flow analysis. The only escaping case found is one that attacks `exec()` spawned processes <181> as SHSTK is not enabled for them by the kernel. It is less effective on MSVC as it fails to protect the return addresses of virtual functions <171> ⓐ. User mode support of IBT is available only on OpenBSD (i712-BSD-GCC-CET). Despite that 5 `cfi-f` cases <187,189,220–222> have failed ⓑ, this protection is still weak, because the `ENDBR` instruction is added to not only function entries but also goto labels, where the latter are non-functional gadgets remaining available for CFI attacks. Instead of relying on IBT, Windows decides to enforce forward CFI using its own CFG, which fails even fewer `cfi-f` cases than IBT.

Using Intel CET as an example to answer question *Q1* raised in the Introduction: Intel CET is finally available for protecting user land applications after 8 years since its initial proposal. It indeed provides protection against both forward and backward CFI attacks. The GCC implemented SHSTK is potentially strong, as it fails almost all `cfi-b` cases. However, none of the platforms has yet supported both SHSTK and IBT,

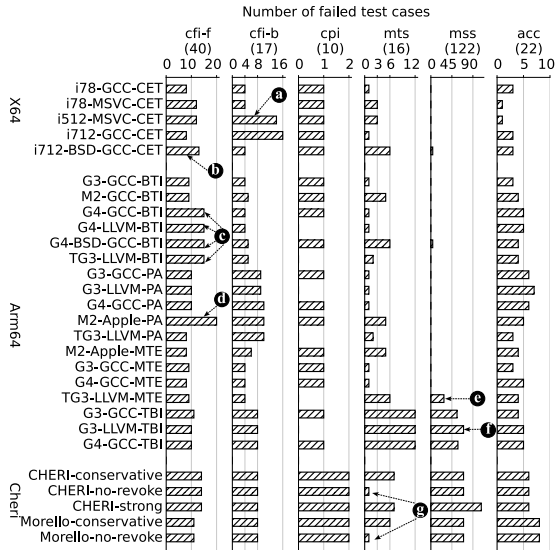



Fig. 6. The results of hardware defenses on commercially platforms.

and the fine differences of the protection achieved on various platforms are clearly observable using our test suite.

**Arm BTI**: Branch target identification (BTI) is introduced in Armv8.5 [47] to provide forward CFI protection similar to what achieved by IBT provided by Intel CET. Although compilers are ready for instrumenting programs, no protection is observed on Graviton 3 and Apple M2, as the support for BTI is optional and the processor hardware is not ready yet, until Graviton 4 and Goolge's Tensor G3. On these two platforms, BTI seems to provide better protection than IBT. It not only reduces available code gadgets but also prevents attackers from hijacking calls to goto labels <220,222>, because the latter are noted by operands of the BTI instruction ⓒ. Notably, only OpenBSD enables BTI by default.

**Arm PA:** Introduced in Armv8.3-A [20], PA enforces the integrity of pointers using a pointer authentication code (PAC) calculated at runtime. This PAC is stored at the highest unused bits of pointers and checked against the recalculated PAC at dereference time [20]. The backward CFI protection provided by PA is weaker than SHSTK. Both GCC and LLVM use the stack pointer as a key in the calculation of PAC for return addresses. This allows attackers to forge return addresses by reusing stack frames <182>. A small variance is also observed on different platforms. In Graviton 3, an authentication failure does not immediately terminate the program <172,178>, leaving room for brutal forced guesses. This loophole is later blocked by Graviton 4, Apple M2, and Tensor G3. Instead of asking all parts to have protection enabled at link time as required by SHSTK, PA can be partially enabled only for critical safety parts <170,177>.

M2-Apple-PA is the only platform (M1 and M3 produce identifical results) deploying PA for additional forward CFI protection ⓓ using an experimental ABI arm64e, which covers function pointers, VTable pointers, and the virtual function pointers stored in VTables [48]. By using both function name and argument types for diversifying PACs, the protection on VTable related pointers is strong. It stops almost all VTable related hijacks <194–205>, include those cases neglected by sanitizers of reusing the VTables of sibling or parent classes. The protection on function pointers is substantially weaker. All function pointers share a globally defined salt in their PACs, allowing for potential control-flow bending by manipulating function arguments <207–211>. Finally, scanning the code segment reveals that some VTables are not covered and remain vulnerable <188>.

**Arm MTE**: Introduced in Armv8.5-A [21], MTE implements a memory coloring machanism typically used for enforcing memory spatial and even temporal safety. To our surprise, our tests show that no runtime protection is observed after enabling MTE in compilers on many of the recent Arm platforms, including Graviton 3 (Armv8.4-A with Armv8.6-A features [49]), Graviton 4 (Armv9.0-A), Apple M2 and M3 (Armv8.6-A) [50]. It is very likely that MTE is opted out on these platforms as it is optional. Google Tensor G3 (TG3-LLVM-MTE) is the only platform found with proper support of MTE. It prevents linear and pinpointed inter-object overflow on heap and stack <38–39,42–43>, but not for global data <40,44> ⓔ. As the same color is shared by each consecutive 16 bytes, MTE fails to detect small overflows within this granularity <67–72>. Intra-object overflow is not detected as well as the whole object shares the same color <54–65>. Generally speaking, MTE provides observable protection against spatial overflows but the protection is not strong.

**Arm TBI**: Introduced in Armv8.5-A [51] and supported by Android 11 and Linux 5.4 [52], Arm top-byte ignore (TBI) is used by both GCC and LLVM to implement HWASan. It is more effective than ASan (X64-GCC-ASan), its software implemented counterpart, on both spatial and temporal safety, because the tag stored in the top-byte of pointers can be used to check the correct ownership of the pointed memory objects [53]. According to our test results, HWASan successfully intercepts extra pinpointed overflows <38–45>, UAF attacks <153,154>, and backward CFI attacks <172>. It achieves the best temporal safety in all tested defenses and strong spatial safety which is weaker only than CHERI (CHERI-strong). GCC's HWASan is relatively weaker than LLVM's on the protection against pinpointed overflows. While GCC's HWASan prevents pinpointed overflows occurred on heap <39,43>, the LLVM's version is able to protect pointers in all regions, including stack and global data <38–45> ⓕ. Similar to MTE, intra-object overflow is not detected.

**CHERI and Morello**: Morello [34] is Arm's implementation of the capability enforced CHERI architecture [10] developed by the University of Cambridge. Both CHERI and Morello have been tested and the results are revealed in Fig. 6. Both platforms enforce strong spatial safety by checking each memory access according to the capability stored inside a 128-bit fat pointer denoted by a 1-bit memory tag. The capability-based check has five predefined restriction levels chosen at compile-time. The default level is `conservative` (CHERI-conservative), where capability checks are already enforced for almost all memory accesses and resulting protection is impressive. The number of failed spatial safety (`mss`) cases is comparable to that achieved by HWASan. When the level is raised to the strictest, `everywhere-unsafe` (CHERI-

TABLE V
DEFENSES COMBINATIONS TESTED ON VARIOUS PLATFORMS.

| Env. | Abbr. | Processor | Compiler | Enabled Defenses |
|---|---|---|---|---|
| Develop | i712-BSD | i7-12700 | LLVM 16.0.0 | CET-IBT, stack pointer |
| | i512-Windows | i5-12400 | MSVC 19.38 | ASan, SDL, CET-SHSTK, stack pointer |
| | i712-Linux | i7-12700 | LLVM 18.0.0 | ASan, UBSan, FORTIFY, CET-SHSTK, stack pointer, CFI |
| | G4-Linux | Graviton 4 | LLVM 18.0.0 | ASan, UBSan, FORTIFY, stack pointer, PA, BTI, CFI |
| | M2-MacOS | Apple M2 | LLVM 15.0.0 | ASan, UBSan, FORTIFY, stack pointer, PA |
| | TG3-Android | Tensor G3 | LLVM 17.0.2 | ASan, UBSan, FORTIFY, stack pointer, PA, BTI, CFI |
| | Morello-CheriBSD | | LLVM 14.0.0 | part of ASan, FORTIFY, capability, revoke |
| Production | i712-BSD | i7-12700 | LLVM 16.0.0 | CET-IBT, stack pointer |
| | i512-Windows | i5-12400 | MSVC 19.38 | SDL, CET-SHSTK, stack pointer |
| | i712-Linux | i7-12700 | LLVM 18.0.0 | FORITFY, CET-SHSTK, stack pointer, safe stack |
| | G4-Linux | Graviton 4 | LLVM 18.0.0 | HWASan, FORTIFY, stack pointer, PA, BTI |
| | M2-MacOS | Apple M2 | LLVM 15.0.0 | FORTIFY, stack pointer, safes stack, PA |
| | TG3-Android(TBI) | Tensor G3 | LLVM 17.0.2 | HWASan, FORTIFY, stack pointer, PA, BTI |
| | TG3-Android(MTE) | Tensor G3 | LLVM 17.0.2 | FORTIFY, stack pointer, PA, BTI, MTE |
| | Morello-CheriBSD | | LLVM 14.0.0 | FORTIFY, capability, revoke |

strong), the capability based check is extended to detect incorrect type conversions <136–143> and overflow inside objects <54–65>, resulting in the strongest spatial safety protection in all tested defenses. Both CHERI and Morello have been carefully designed to avoid capability-protected pointers from being modified by non-capability registers in embedded assembly codes. CHERI would report errors at compile-time while Morello chooses to allow such code passing compilation for compatibility but raise exceptions at runtime. However, the protection of spatial safety still leaves a small vulnerability as indicated by our test suite. The capability recorded size information becomes imprecise for large arrays, which allows for undetected overflows <46–53>. Comparing between CHERI and Morello, CHERI provides better `cfi-f` protection by further detecting all code pointer casting errors <208–210>.

Both CHERI and Morello provide temporal safety on heap through CHERIvoke, a garbage collection based on fast memory sweeping and revocation. It is enabled by default but can be disabled in kernel ⓖ. According to our test results, CHERIvoke sufficiently prevents attackers from reclaiming freed objects, but the memory sweeping might be slow to catch all accesses after free <153>. Protection for UAF on stack is also notably missing <147–152>. Overall, the achieved protection is comparable to that provided other enhanced memory allocators.

Let us use Morello as an example to answer question *Q2* raised in the Introduction: Morello/CHERI claims to achieve strong spatial safety through capability-enforced compartimentalization. Our test results have successfully demonstrated that the protection is indeed strong even at the conservative level. When the level is increased to everywhere-unsafe, both incorrect type conversion and intra-object overflows are prevented, achieving the strongest protection in all defenses. However, the protection still has loopholes as the size information stored in tags is imprecise for large arrays.

### C. Comparison of memory safety crossing platforms

In this section, we would like to answer question Q3 raised in the Introduction by comparing the memory safety crossing platforms when all available defenses are enabled. Considering that software ASans are regarded as debug tools used during development, we conduct the comparison in two settings: *develop* environment where all defenses, including ASans, are enabled, and *production* environment where all defenses other than ASans are enabled. The defense combinations are listed in Table V while the test results are presented in Fig. 7.

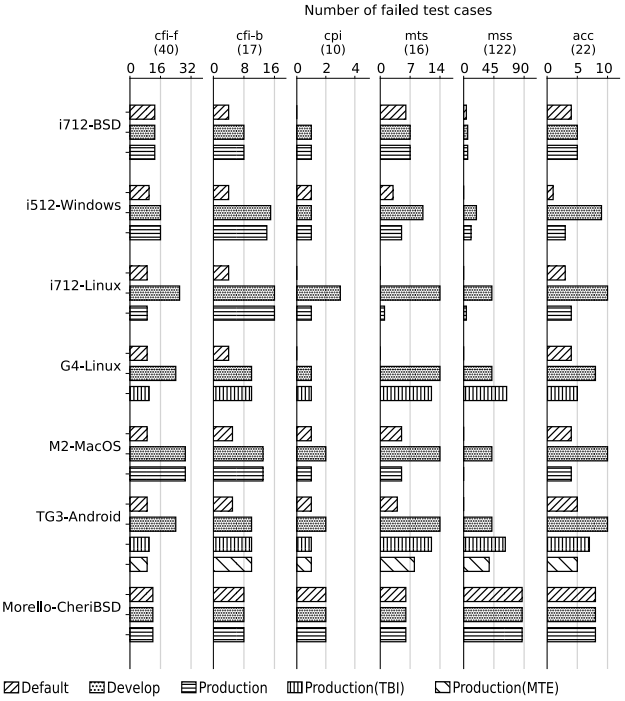



Fig. 7. The results of enabling all defenses on commercially platforms.

According to our test results, there is no clear winner in the develop environment, largely because of the different combinations of defenses supported on various platforms. UBSan is effective in protection against certain types of overflows (`mss`) and COOP attacks (`cfi-f`), but it is not fully supported on BSD and Windows, leading to their being worse than most other platforms with respect to spatial safety and forward CFI. BSD is especially weak on spatial safety, because ASan is unavailable too. For temporal safety, ASan is actually the most effective as it thwarts both UAF on heap and on stack. All the platforms supporting the full

ASan outperform Windows (due to the partial support of ASan) and BSD (i712-BSD and Morello-CheriBSD, relying on enhanced memory allocators). CET-SHSTK is currently the strongest defense against ROP attacks. For this reason, i512-Windows and i712-Linux are better than the Arm PA enforced G4-Linux, M2-MacOS and TG3-Android on backward CFI, while BSD platforms again are the least protected. Thanks to the capability-enforced compartimentalization on Morello, it presents the strongest spatial safety, but protection on other aspects shows no significant advantage.

In the production environment, software implemented sanitizers (ASan and UBSan) are disabled and HWAsan is enabeld if it is available (G4-Linux and TG3-Android(TBI)). No difference is observed for BSD platforms (i712-BSD and Morello-CheriBSD) as they support none of ASan, UBSan, or HWASan. The platforms (i512-Wiindows, i712-Linux and M2-MacOS) supporting ASan or UBSan but not HWASan clearly suffer substantially from weak both spatial (`mss`) and temporal (`mts`) safety. Comparing between HWASan and the combination of ASan+UBSan, HWASan is actually more effective than ASan in spatial safety but weaker than UBSan in temporal safety, which is observable by the results of G4-Linux and TG3-Android. Arm MTE is also effective in enforcing spatial and temporal safety (TG3-Android(MTE)), but the provided protection is weaker than HWASan (TG3-Android(BTI)). Without the protection of UBSan, Apple's PA is used to enforce CFI on M2-MacOS, providing almost the same level of protection. However, i712-Linux, G4-Linux and TG3-Android resort to CDT-IBT or Arm BTI for forward CFI, which is much weaker than the Apple implemented PA. Overall, Morello/CHERI remain strong in the production environment; HWASan and Apple's PA help TG3-Android, G4-Linux and G4-Linux maintain their safety level in the absence of ASan and UBSan; i512-Windows and i712-Linux suffer from weaker protection with respect to spatial and temporal safety.

It is also important to summerize the common vulnerabilities identified when all available defenses are enabled. It is found that some critical information is still easily acccessible on most platforms, such as locating RA on stack <17–19>, reading function pointers <16> or VTable pointers <160–163>, deciphering GOT table entries <21>, detecting redzones <4–11>, checking ASLR status <14,15> or CET status <20>, etc. Manipulating pointers is not prevented even on platforms enforced with Arm PA, such as arbitrary modification on function and VTable pointers <167,168> or adding pointers with arbitrary offset <164–166>. Morello/CHERI is the only platform capable of detecting intra-object overflows <54–65>, and small sized overflow (overflow by one) remains possible due to the imprecise recording of size (tag in MTE <67–72> and CHERI<73>). Although UAF on heap is carefully protected using enhanced memory allocators and detectable by UBSan/HWASan, UAF on stack is largely overlooked, allowing attackers to reuse stack frames <147–152>. The lack of stateful CFI enforcement on most platforms allows for potential control-data attack and evasive control hijacking (complying with static CFI analysis) <173>. Existing COOP defenses are ineffective in detecting attacks reusing the VTables of sibling or parent classes, or even released objects <201,202,206>.

Finally, let us answer question *Q3* raised in the Introduction. The results in Fig. 7 perfectly demonstrate the ability of using our test suite to compare the memory safety provided by defense combinations on various platforms. The same architecture extensions, such as Intel CET and Arm PA, are utilized by different platform vendors in different ways when building user land defenses. The unique characteristics of our test suite make the resulting differences in safety levels observable. For example, Apple's PA provides much stronger protection than Arm's PA and Windows' own backward CFI protection is weaker than CET-SHSTK.

## VI. Conclusion

In this paper, a memory safety test suite, namely *MSTest*, is implemented. It is the first portable memory safety test suite conducting property-oriented testing, automatically resolving dependency between test cases, providing a comprehensive coverage on attack and defense capabilities, and capable of comparing memory safety cross platforms.

MSTest currently has 227 test cases covering access capability, memory spatial and temporal safety, code-pointer integrity, and backward and forward control-flow integrity. It has been ported to 19 platforms crossing all major architectures, including x86-64, Armv8/9, RV64 and CHERI, operating systems, including Linux, Windows, MacOS and Android, and compilers, including GCC, LLVM and MSVC. Utilizing MSTest, we have analyzed the protection provided by various software and hardware implemented defenses directly available on these platforms, including compiler shipped sanitizers (ASan, UBSan, HWASan), stack protection, safe stack, GCC VTV, LLVM CFI, MSVC CFG, MSVC SDL, Intel CET, Arm BTI, Arm TBI, Arm PA, Arm MTE, and CHERI.

The test results demonstrates that MSTest can be used to: (*Q1*) verify whether a defense, which is claimed supported on a platform, can be actually deployed in user land and provide proper protection; (*Q2*) quantitatively measure the protection provided by a defense on a platform and reveal the remaining vulnerability; and (*Q3*) compare the memory safety of two platforms implementing similar types of defenses. To be specific, we quantitatively verify that the software implemented ASan+UBSan is effective in detecting violations in spatial and temporal safety, along with forward CFI. With these software sanitizers, HWASan is a good substitute for spatial and temporal safety. Apple's PA is capable of providing similar forward CFI enforcement, while CET-IBT and Arm-BTI is significantly weaker. CET-SHSTK provides stronger protection against ROP attacks than Arm PA. Finally, CHERI is impressive in protecting spatial memory safety.

## Acknowledgments

Yuhui Zhang and Boya Li has contributed to the early versions of MSTest. Zhuxin Yang had helped review PRs towards a recent version of MSTest. The RISC-V platforms evaluated in this paper are donated by Xiongfei Guo, Wei Wu, and the PLCT Lab of the Institute of Software, CAS.

## REFERENCES


[1] S. Nagarakatte, "Full spatial and temporal memory safety for C," *IEEE Secur. Privacy*, pp. 2–11, 2024.

[2] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 48–62.

[3] PaX Team, "PaX address space layout randomization (ASLR)," 2003. [Online]. Available: https://pax.grsecurity.net/docs/aslr.txt

[4] M. Tran, M. Etheridge, T. K. Bletsch, X. Jiang, V. W. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *Proc. Int. Symp. Recent Adv. Intru. Detect*, Sep. 2011, pp. 121–141.

[5] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proc. ACM Symp. Inf. Comput. Commun. Secur.*, Mar. 2011, pp. 30–40.

[6] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 745–762.

[7] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Proc. USENIX Secur. Symp.*, Jul. 2005, pp. 177–191.

[8] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-Sanitizer: A fast address sanity checker," in *Proc. USENIX Annu. Tech. Conf.*, Jun. 2012, pp. 309–318.

[9] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hard-bound: Architectural support for spatial safety of the C programming language," in *Int. Conf. Archit. Support Prog. Lang. Oper. Syst.*, Mar. 2008, pp. 103–114.

[10] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *Proc. Int. Symp. Comput. Archit.*, Jun. 2014, pp. 457–468.

[11] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov, "Memory tagging and how it improves C/C++ memory safety," 2018. [Online]. Available: https://arxiv.org/abs/1802.09517

[12] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 1–40, Nov. 2009.

[13] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: Cryptographically enforced control flow integrity," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2015, pp. 941–951.

[14] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *Proc. USENIX Conf. Oper. Sys. Design Impl.*, Oct. 2014, pp. 147–163.

[15] U. Dhawan, N. Vasilakis, R. Rubin, S. Chiricescu, J. M. Smith, T. F. Knight, B. C. Pierce, and A. DeHon, "PUMP: A programmable unit for metadata processing," in *Proc. Workshop Hardware Archi. Support Secur. Priv.*, Jun. 2014.

[16] W. Chang, B. Streiff, and C. Lin, "Efficient and extensible security enforcement using dynamic data flow analysis," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2008, pp. 39–50.

[17] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *Proc. USENIX Secur. Symp.*, Aug. 2014, pp. 941–955.

[18] R. Ramakesavan, D. Zimmerman, P. Singaravelu, G. Kuan, B. Vajda, S. Gibbons, and G. Beeraka, *Intel® Memory Protection Extensions Enabling Guide*, Apr. 2016.

[19] Intel Corporation, "Control-flow enforcement technology specification (revision 3.0)," May 2019. [Online]. Available: https://kib.kiev.ua/x86docs/Intel/CET/334525-003.pdf

[20] D. Brash, "Armv8-A architecture: 2016 additions," Oct. 2016. [Online]. Available: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/armv8-a-architecture-2016-additions

[21] K. Serebryany and S. Herle, "Adopting the Arm memory tagging extension in Android," Aug. 2019. [Online]. Available: https://security.googleblog.com/2019/08/adopting-arm-memory-tagging-extension.html

[22] Apple Inc., "Pointer authentication," 2019. [Online]. Available: https://github.com/apple/llvm-project/blob/apple/main/clang/docs/PointerAuthentication.rst

[23] Arm Limited, "Arm® architecture reference manual supplement morello for A-profile architecture," Jan. 2022. [Online]. Available: https://developer.arm.com/documentation/ddi0606/latest

[24] J. Poe and T. Li, "BASS: a benchmark suite for evaluating architectural security systems," *ACM SIGARCH Comp. Arch. News*, vol. 34, no. 4, pp. 26–33, 2006.

[25] X. Xu, M. Ghaffarinia, W. Wang, K. W. Hamlen, and Z. Lin, "CONFIRM: evaluating compatibility and relevance of control-flow integrity protections for modern software," in *Proc. USENIX Secur. Symp.*, Aug. 2019, pp. 1805–1821.

[26] Y. Li, M. Wang, C. Zhang, X. Chen, S. Yang, and Y. Liu, "Finding cracks in shields: On the security of control flow integrity mechanisms," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2020, pp. 1821–1835.

[27] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "RIPE: Runtime intrusion prevention evaluator," in *Proc. Annu. Comput. Secur. Applications Conf.*, 2011, pp. 41–50.

[28] Y. Jiang, R. H. Yap, Z. Liang, and H. Rosier, "RecIPE: Revisiting the evaluation of memory error defenses," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, May 2022, pp. 574–588.

[29] Gallopsled, "Pwntools: CTF framework and exploit development library," 2024. [Online]. Available: https://github.com/Gallopsled/pwntools

[30] T. Boland and P. E. Black, "Juliet 1. 1 C/C++ and Java test suite," *Computer*, vol. 45, no. 10, pp. 88–90, 2012.

[31] C. Yagemann, M. Pruett, S. P. Chung, K. Bittick, B. Saltaformaggio, and W. Lee, "ARCUS: Symbolic root cause analysis of exploits in production systems," in *Proc. USENIX Secur. Symp.*, Aug. 2021, pp. 1989–2006.

[32] A. Waterman, T. Newsome, C.-M. Chao, Y. Lee, S. Beamer, and others, "Spike, a RISC-V ISA simulator," 2024. [Online]. Available: https://github.com/riscv-software-src/riscv-isa-sim

[33] R. N. M. Watson, "CHERI-QEMU," 2024. [Online]. Available: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-qemu.html

[34] Arm Limited, "Arm Morello program," 2024. [Online]. Available: https://www.arm.com/architecture/cpu/morello

[35] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," in *Proc. IEEE Symp. Secur. Privacy*, May 2019, pp. 870–887.

[36] C. Ouyang, D. Xie, H. Ma, and W. Song, "Supplementary of MSTest," Sep. 2025. [Online]. Available: https://github.com/comparch-security/cpu-sec-bench/blob/TDSC/supplement.pdf

[37] S. Inc., "SiFive FU540-C000 manual (v1p4)," Mar. 2021. [Online]. Available: https://www.sifive.com/boards/hifive-unleashed

[38] Arm Limited, "Fast models fixed virtual platforms (FVP) reference guide (version 11.25)," Mar. 2024. [Online]. Available: https://developer.arm.com/documentation/100966/

[39] c0ntex, "How to hijack the global offset table with pointers for root shells," Mar. 2006, https://www.exploit-db.com/papers/13237.

[40] K. Nanako, "'check failed: sanitizer_allocator_primary32.h:292' when running on RISC-V 64 systems using SV48," Nov. 2023. [Online]. Available: https://github.com/google/sanitizers/issues/1707

[41] M. Larabel, "RISC-V user-space pointer masking appears ready for Linux 6.13," Oct. 2024. [Online]. Available: https://www.phoronix.com/news/RISC-V-Pointer-Masking-Linux

[42] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "SoK: Sanitizing for security," in *Proc. IEEE Symp. Secur. Privacy*, May 2019, pp. 1275–1295.

[43] Microsoft, "/GUARD (enable guard checks)," Sep. 2022. [Online]. Available: https://learn.microsoft.com/en-us/cpp/build/reference/guard-enable-guard-checks

[44] M. Larabel, "Intel CET support still getting squared away for Linux in 2020," May 2020. [Online]. Available: https://www.phoronix.com/news/Intel-CET-2020-WIP

[45] ——, "GCC lands Cannonlake, Skylake costs; LLVM/Clang gets Intel CET," Nov. 2017. [Online]. Available: https://www.phoronix.com/news/GCC-Skylake-Costs-LLVM-CET

[46] ——, "Glibc updated for recent Linux CET shadow stack support," Jan. 2024. [Online]. Available: https://www.phoronix.com/news/Glibc-Intel-CET-Shadow-Stack

[47] M. Gretton-Dann, "Arm® A-Profile architecture developments 2018: Armv8.5-a," Sep. 2018. [Online]. Available: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-a-profile-architecture-2018-developments-armv85a

[48] Apple Inc., "Apple platform security: Operating system integrity," Dec. 2024. [Online]. Available: https://help.apple.com/pdf/security/en_US/apple-platform-security-guide.pdf

[49] Arm Limited, "Neoverse v1," Sep. 2024. [Online]. Available: https://developer.arm.com/Processors/Neoverse%20V1

[50] T. Lelegard, "Arm features in the Apple M1 and M2 chips," Apr. 2024. [Online]. Available: https://github.com/lelegard/arm-cpusysregs/blob/main/docs/apple-m1-features.md

[51] Arm Limited, "Arm architecture reference manual Armv8, for Armv8-A architecture profile," Jul. 2019. [Online]. Available: https://developer.arm.com/documentation/ddi0487/ea

[52] J. Corbet, "Pointer tagging for x86 systems," Mar. 2022. [Online]. Available: https://lwn.net/Articles/888914/

[53] The Clang Team, "Hardware-assisted addresssanitizer design documentation," 2024. [Online]. Available: https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html

**Ciyan Ouyang** received his B.E. degree from Chang'an University in 2019 and his M.E. degree from the University of the Chinese Academy of Sciences (UCAS) in 2022. He is currently working toward the Ph.D. degree at the Institute of Information Engineering, CAS. His research interests lie in system security, with a focus on testing and defense.

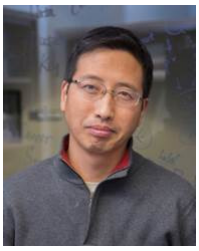
**Da Xie** received the B.S. degree from Huazhong University of Science and Technology. He is currently a Ph.D. student at the Institute of Information Engineering, CAS. His current research focuses on secure computer architectures.

**Hao Ma** received the B.S. degree from Xi'an University of Science and Technology, in 2017, and the M.S. degree from Xidian University, in 2020. He is currently pursuing the Ph.D. degree at the Institute of Information Engineering, CAS. His research interests include computer architecture and cache side-channel attacks.

**Wei Song** (Senior Member, IEEE) received the B.S. and M.S. degrees from Beijing University of Technology, and the Ph.D. degree from the University of Manchester. He is an Associate Professor at the Institute of Information Engineering, CAS. He had worked as a postdoctoral researcher in the University of Manchester and the University of Cambridge. His current research focuses on the security enhancement of computer architectures, such as the defenses for cache side channel and control-flow hijacking attacks.

**Jiameng Ying** received the B.E. degree from China University of Geosciences, and the Ph.D. degree from the University of the Chinese Academy of Sciences, in 2024. He currently works at Big Data Center of the Ministry of Public Security. His current research interests include processor security and system security.

**Sihao Shen** received the M.S. degree from the University of the Chinese Academy of Sciences, in 2022. He worked at Alibaba T-Head for an extended period, engaging in GPU driver development. His research interests include processor cache side-channel attacks, GPU architecture, and performance optimization.



**Peng Liu** (Member, IEEE) received the B.S. and M.S. degrees from the University of Science and Technology of China, and the Ph.D. degree from George Mason University, in 1999. He is the Raymond G. Tronzo, MD Professor of Cybersecurity, and the Director of the Cyber Security Lab at Penn State University. His research interest include computer security. He has published over 350 technical papers. He has served on over 100 program committees and reviewed papers for numerous journals. Currently, he is the Co-Editor-in-Chief of *Journal of Computer Security*.