CCTAG: Configurable and Combinable Tagged Architecture

Zhanpeng Liu*†, Yi Rong§, Chenyang Li*†, Wende Tan§,
Yuan Li¶†‖‡✉, Xinhui Han*‖‡✉, Songtao Yang¶, Chao Zhang†¶‖
*Wangxuan Institute of Computer Technology (WICT), Peking University
†Institute for Network Sciences and Cyberspace (INSC), Tsinghua University, §Tsinghua University, ¶Zhongguancun Laboratory,
‖JCSS, Tsinghua University (INSC) - Science City (Guangzhou) Digital Technology Group Co., Ltd.

***Abstract*—Memory safety violations are a significant concern in real-world programs, prompting the development of various mitigation methods. However, existing cost-efficient defenses provide limited protection and can be bypassed by sophisticated attacks, necessitating the combination of multiple defenses. Unfortunately, combining these defenses often results in performance degradation and compatibility issues.**

We present CCTAG, a lightweight architecture that simplifies the integration of diverse tag-based defense mechanisms. It offers configurable tag *verification* and *modification* rules to build various security policies, acting as basic protection primitives for defense applications. Its policy-centric mask design boosts flexibility and prevents conflicts, enabling multiple defense mechanisms to run concurrently. Our RISC-V prototype on an FPGA board demonstrates that CCTAG incurs minimal hardware overhead, with a slight increase in LUTs (6.77%) and FFs (8.02%). With combined protections including ret address protection, code pointer and vtable pointer integrity, and memory coloring, the SPEC CPU CINT2006 and CINT2017 benchmarks report low runtime overheads of 4.71% and 7.93%, respectively. Security assessments with CVEs covering major memory safety vulnerabilities and various exploitation techniques verify CCTAG's effectiveness in mitigating real-world threats.

I. Introduction

Memory safety vulnerabilities are the most common at the binary level [2], [1], including out-of-bounds access, dereferencing expired pointers, usage of uninitialized resources, and type confusion in dynamic language runtimes. These issues can create invalid pointers and memory errors, allowing attackers to manipulate the program's internal state and leading to control flow hijacking, data-only attacks, and information leakage [47]. Despite numerous defenses developed to mitigate increasing software vulnerabilities at runtime, most offer only limited protection guarantees. They often target specific vulnerability types [38], [27], [53], [21] or prevent certain exploitation steps in the exploitation workflow [4], [18], [28], [34]. For comprehensive protection, it's advisable to integrate multiple defenses. Based on a step-by-step memory exploitation model proposed by [47], defenses can be combined in two ways:

- **Horizontal integration** combines defense mechanisms at the same layer to cover more types of vulnerabilities or block a broader range of exploitation techniques. For example, integrating spatial and temporal memory safety solutions provides more comprehensive protection than individuals.
- **Vertical integration** blocks different stages of the exploitation workflow, providing a "defense in depth" strategy. With this approach, even though individual protection mechanisms are only moderately precise, the combined protection can significantly hinder attackers. An example of this would be combining memory safety techniques with Control Flow Integrity (CFI) methods.

Integrating multiple defense mechanisms is challenging due to compatibility issues, accumulated performance degradation, and other associated costs. Pure software-based defenses often have high performance and memory overhead [19], [20], [21], and when combined, this overhead can become unacceptable for production deployment. Hardware-assisted defenses leverage specific hardware features to reduce performance impacts [41], [56], [59], [43], [42], [25]. However, integrating these solutions by combining underlying hardware features increases hardware costs (e.g., chip area) and results in a more complex design that's difficult to maintain and verify. *These challenges require a solution that makes hardware more versatile and configurable, enabling multiple defense applications to run concurrently.* Configurable or even programmable security features have been seen in several tagged architecture designs [15], [7], [17], [14], [46], [49]. Existing multipurpose tagged architectures lack flexibility and cannot support efficient defense integration. Their complex designs, high hardware requirements, and significant runtime overhead make them impractical for real-world applications.

In this paper, we present CCTAG, a novel, lightweight tagged architecture designed to be both configurable and combinable, addressing the need for efficient and compatible integration of various defense mechanisms. CCTAG supports a wide range of customizable tag verification and manipulation rules and can be used as building blocks to construct various protection

‡✉Corresponding authors: lydorazoe@gmail.com, hanxinhui@pku.edu.cn

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240862
www.ndss-symposium.org

primitives, including fine-grained access control, data-flow isolation, dynamic information tracking, etc. Such primitives can then be utilized to implement different defense mechanisms. For example, with the primitive to temporarily mark a word as read-only, it can effectively prevent the overwriting of sensitive data. `CCTAG` offers great flexibility and only incurs little complexity to hardware design. On the other hand, `CCTAG` introduces a policy-centric mask that enables multi-level, fine-grained control over tag rules applied at the level of threads, pages, and variable granularity within a cache line. This design facilitates the simultaneous support for multiple defense mechanisms, allowing them to collaborate to provide stronger security properties.

We have developed a prototype of `CCTAG` running on a Field Programmable Gate Array (FPGA), which includes modifications to both the processor and the operating system. We extended the RISC-V instruction set architecture (ISA) and augmented the Rocket Core to support the tag operations. Such extension introduces low hardware resource overhead, as the LUTs usage increases by only 6.77% while the FFs usage increases by 8.02%. Additionally, we have made minor modifications to the Linux kernel, introducing system calls that enable the configuration of tag policies for each thread and each memory page, as well as exception handling for tag check failures. With the prototype, we demonstrate that `CCTAG` can provide various memory protection primitives, and showcase 4 specific primitives in §V, including *fine-grained permission*, *sensitive data flow isolation*, *memory coloring*, and *runtime information tracking*. For each primitive, we port a defense application for verification, including *stack frame protection*, *sensitive pointers isolation*, *heap overflow and use-after-free mitigation*, and *dangling pointer scan*. We also discuss other potential applications of `CCTAG` beyond memory safety mitigation in §VI-D.

We evaluated the performance of the prototype `CCTAG` system with the SPEC CINT2006 and CINT2017 benchmark suites. Additionally, we assessed its security capabilities against real-world CVEs and various exploitation techniques. Results showed that `CCTAG` incurs an acceptable level of runtime overhead (4.71% on CINT2006 and 7.93% on CINT2017) with combined defense mechanisms, which can effectively mitigate a broad range of memory safety vulnerabilities. In summary, we make the following contributions:

1) We propose a policy-centric mask design that enables flexible tag configurations and supports multiple defense mechanisms simultaneously without conflicts.
2) We introduce `CCTAG`, a tagged architecture with a policy-centric mask that is configurable and combinable to meet diverse defense requirements effectively.
3) We demonstrate `CCTAG`'s capability by supporting four memory protection primitives, each verified with a ported defense application.
4) We evaluated `CCTAG` by developing a RISC-V prototype on an FPGA. Results show that `CCTAG` only slightly increases hardware resources and incurs low runtime overhead, confirming its viability for real-world applications.

TABLE I: Statistics of CVE types related to memory safety.

CWE-ID	Name	Count	Rank
CWE-787	Out-of-bounds Write	5,616	1
CWE-416	Use After Free	1,889	4
CWE-125	Out-of-bounds Read	1,467	8
CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	1,338	9
CWE-843	Access of Resource Using Incompatible Type ('Type Confusion')	230	37
CWE-415	Double Free	182	44
CWE-908	Use of Uninitialized Resource	106	64
CWE-824	Access of Uninitialized Pointer	96	66

II. Background & Related Work

A. Memory Safety Vulnerabilities

Memory safety violations are the root cause of many exploitable conditions. These violations generally fall into two categories [44]:

- **Spatial violations.** As seen in buffer overflow attacks, a pointer accesses memory outside its intended object.
- **Temporal violations.** A pointer is dereferenced after the lifecycle of its associated object has ended, leading to issues such as double frees and use-after-free.

To obtain a clear understanding of the prevailing memory safety vulnerabilities in recent years, we conducted a statistical analysis of high-severity Common Vulnerabilities and Exposures (CVEs) from the National Vulnerability Database (NVD) spanning 2020 to 2024. Our study categorized a total of 43,112 CVEs according to their Common Weakness Enumeration (CWE) types. Part of the results are listed in Table I.

Notably, CWE-787 (Out-of-bounds Write) emerged as the most prevalent vulnerability among all CWE types, accounting for 5,616 instances, while CWE-416 (Use After Free) ranked fourth with 1,889 instances. These two types can directly lead to spatial and temporal memory safety violations, respectively. Other CWEs closely associated with memory safety violations include CWE-125, CWE-120 and CWE-415, and so on. Furthermore, we identified 208 instances of vulnerabilities related to uninitialized memory (CWE-457, CWE-824, and CWE-908) and 230 instances of type confusion vulnerabilities (CWE-843). These types of vulnerabilities are also intimately related to memory errors. For example, they can result in the use of out-of-context pointers which might be controlled by attackers. Although these numbers are lower than those for spatial and temporal memory safety violations, they remain substantial enough to warrant serious attention.

B. Tagged Architecture

Tagged architectures represent a significant category of hardware security primitives that can assist memory safety enforcement at runtime. In these architectures, program data is augmented with an additional metadata tag that stores security-related information, typically at the granularity of a word [32]. Benefiting from the close coupling of tags and the data,

the tags can be checked and updated concurrently with the execution of each instruction. Different rules for verifying and updating tags constitute various tag policies, each tailored to meet different and specific security requirements and provide corresponding memory safety guarantees.

C. Memory Safety Violation Mitigation

Many runtime defense mechanisms have been developed to mitigate memory safety violations. These solutions balance protection capability, compatibility, efficiency, and hardware overhead for hardware-assisted methods. Below, we discuss mainstream approaches and introduce representative works.

1) Spatial Memory Safety Enforcement: A direct way to ensure spatial memory safety is to associate each pointer with its base address and size, performing explicit bounds checks during dereferencing. In-Fat [56] achieves this by storing 16-bit metadata in unused pointer bits, allowing hardware to efficiently access bounds and layout information. CHERI [51] extends 64-bit pointers into capability pointers that encode bounds directly, enhancing efficiency but creating compatibility challenges with existing systems and software. The red zone is another approach that adds inaccessible regions around protected buffers. While less accurate, it offers the advantage of significantly lower hardware and performance costs. ARM MTE [5] (memory coloring) enhances this by embedding 4-bit tags in pointers and every 16-byte memory block. A pointer dereference is valid only if its tag matches the memory block's tag. Careful tag allocation ensures strong spatial memory safety.

2) Temporal Memory Safety Enforcement: One-time allocation [53], [26] effectively enforces temporal memory safety by never reusing freed memory, but it incurs high memory overhead. Quarantining delays memory reuse by placing freed blocks in a quarantine pool. MineSweeper [21] verifies no remaining references before release, while Cornucopia [52] (based on CHERI) uses capability pointers to free quarantined memory and invalidate dangling pointers after scanning. ARM MTE supports temporal memory safety by re-tagging memory on each allocation and deallocation, enabling probabilistic detection of violations.

3) Control Flow Integrity: Control Flow Integrity (CFI [4]) is designed to ensure the integrity of a program's control flow, preventing attackers from arbitrarily hijacking it, even if memory safety vulnerabilities are exploited. Although researchers have proposed various CFI schemes[18], [34], [28] to improve the accuracy of the control flow graph, attacks [22], [40], [11] that successfully bypass these defenses have also been continuously introduced. In addition, some works [10], [57] have shown that even fully-precise CFI can still be attacked.

4) Intra-Process Isolation: Intra-process memory isolation extends the principle of least privilege within a single process, restricting memory access for different parts of the code to prevent one compromised component from affecting the entire memory space. Mondriaan [54] and Loki [58] implement word-level data tagging with permission lists, enabling multiple protection domains, while Intel MPK [39] uses page-level granularity for similar protection. CHERI also offers sophisticated support for isolation. This approach facilitates compartmentalization, where programs are divided into compartments with access to specific resources and communicate via controlled channels, as seen in PANIC [55] and HAKCs [37]. Another use case is to isolate sensitive data. VIP [31] uses MPK-protected shadow memory to safeguard sensitive information. HDFI [43] employs 1-bit tags for each memory word to separate sensitive and non-sensitive data flows, revoking tags if unauthorized writes occur.

5) Pointer Integrity: Pointer integrity mechanisms prevent pointers from being tampered with or forged, effectively blocking control flow hijacking and limiting memory corruption. Protecting code pointers offers stronger security than classic CFI [40], [10], while safeguarding data pointers further restricts attack capabilities. ARM Pointer Authentication (PA) cryptographically signs and verifies pointers. With above mentioned intra-process isolation to isolate pointers is also a way to guarantee pointer integrity.

6) Dynamic Information Flow Tracking: Dynamic Information Flow Tracking (DIFT [45]) monitors data movement at runtime to detect unintended or malicious information flows. By tracking data through the system, DIFT can identify suspicious activities like unauthorized access to sensitive data or the misuse of untrusted data in critical operations. Tagged architecture is commonly used in this area [13], [49], [33].

D. Multipurpose Tagged Architecture Designs

Researchers have developed general-purpose tagged architectures to support multiple policy paradigms. We categorize these designs into two main types based on their versatility:

1) Configurable Tagged Architecture: Configurable tagged architectures use hardware modules with predefined rules (set via control registers) for tag checking and updating. This approach offers limited flexibility and cannot be updated after fabrication but maintains low hardware and runtime overhead. Examples include Raksha [13], which employs configurable registers for flexible DIFT policies, and its derivatives FlexiTaint [49] and the DIFT co-processor [33]. LowRISC [36] also supports permission checks and DIFT using function unit masks. `CCTAG` builds on this model by using control registers to configure the tag module with enhanced versatility and flexibility, allowing easier integration of diverse security applications while keeping overhead low.

2) Programmable Tagged Architecture: In programmable tagged architectures, the rules for tag checking and updating can be defined by the program, typically through a software-managed rule cache that allows hardware to quickly lookup. FlexiTaint [49] uses configurable caching for tag checking and updating rules, transferring control to an exception handler on cache misses. This approach is adopted by systems like Harmoni [15], SAFE [7], and PUMP [17], [16], which differ in the context used for tag rule lookup. Increasing context input improves versatility and enables stronger memory safety policies but results in significant hardware and runtime overhead, such as PUMP's 2.1× area increase. Additionally, these

systems often suffer performance penalties from rule cache misses, especially with complex configurations or multiple policies, leading to frequent cache misses and reduced performance. Another approach uses a co-processor or extra core for tag checking and updating. FlexCore [14] employs an FPGA as a programmable co-processor, but it operates slower than the main processor and can block it when busy. Dover [46] uses an additional core for tag management, which is more efficient than FlexCore but doubles hardware resources and power consumption.

Table II compares the discussed general-purpose tagged architectures in terms of versatility, efficiency, and overhead using data from their respective studies. Despite different platforms and evaluation methods, the table provides a general performance overview. Most existing designs either lack the efficiency and compatibility needed for major architectures or do not support a wide range of protection paradigms. CCTAG achieves an optimal balance by supporting various defense mechanisms while remaining simple and elegant, making it easily integrable into industrial architectures.

III. Motivation and Threat Model

A. Motivation to Combine Protections

We propose two ways to integrate defensive mechanisms: horizontal and vertical. Horizontal integration offers clear advantages of expanding the coverage of more vulnerability types. Vertical integration enables each layer to be moderately precise rather than excessively strict, creating opportunities to balance security gains with various costs, such as hardware resources, memory overhead, and performance loss.

Highly accurate defense techniques often incur significant costs. For instance, In-Fat [56] performs bounds checking at sub-object granularity, resulting in over 50% more CPU logic and over 20% overhead in memory and performance. ARM MTE offers a lighter solution but faces intra-object overflow issues and cannot prevent pointer forgery. Incorporating pointer integrity mechanisms can substantially improve overall security. ARM platform also includes PA, which can provide pointer integrity. When used in conjunction with MTE, PA complements its capabilities by addressing gaps in MTE's defense. However, the combination remains insufficient. PA protects against pointer tampering, and MTE enhances object-level memory security, but data-only attacks that exploit intra-object buffer overflows can evade both.

Listing 1 shows a crafted vulnerable program that can not be protected by CFI and MTE-based memory protection. In function `update`, the `name` buffer is copied from the input `name` without any length check, which can overwrite the `permissions` field. MTE can not prevent this attack because the `permissions` field is within the same object as the `name` buffer. And because it is not a pointer, PA can not ensure its integrity either. This example illustrates the inherent limitation of existing combinations of hardware features, which can only provide a limited level of defense. *So there is an urgent need for mechanisms integrating different defense strategies to combat evolving novel attacks.*

```
1  struct Client {
2    char name[8];
3    unsigned long permissions;
4  };
5
6  void update(struct Client *client, char *name)
       {
7    // May overwrite permissions
8    strcpy(client->name, name);
9  };
10
11 void serve(struct Client *client) {
12   do {
13     if (has_admin_perm(client.permissions)) {
14       do_admin_stuff();
15     } else {
16       do_user_stuff();
17     }
18   } while (1);
19 }
```

Listing 1: Motivating Example

B. Threat Model

We assume there are memory safety vulnerabilities (e.g., out-of-bounds access, UAF, uninitialized memory, and type confusion) in victim programs and that adversaries are aware of CCTAG's protections. We aim to prevent control flow hijacking, data-only attacks, and information leakage with the best effort. Though CCTAG's design can be used to protect the kernel, we currently focus on user-space memory safety and assume a trusted operating system kernel with no vulnerabilities. We also assume the hardware is trusted and free from vulnerabilities, so side-channel and row-hammer [35] attacks are beyond the scope of this paper.

IV. System Design

In this section, we introduce the high-level design of CCTAG. We first present our novel policy-centric mask, essential for the system's configurable and combinable features. Then, we discuss the detailed design, including the hardware microarchitecture, supported tag rules, and kernel adaptation.

A. Policy-Centric Mask

In general purpose tagged architectures, though they can be configured with different tag policies, the underlying tag storage and part of the hardware logic are shared, which prevents their coexistence. CCTAG introduces the concept of policy-centric mask to address this issue. We'll first give our motivation and intuitions behind this concept, followed by a detailed explanation of how it is designed.

1) Co-variation of Tag Bits and Tag Granularity: Different tag policies necessitate varying levels of memory granularity. While most tagged architectures operate at word granularity, particular scenarios may require finer or coarser granularity. For instance, when implementing Dynamic Information Flow Tracking (DIFT) policies for taint analysis, byte granularity is preferable. Conversely, ARM's Memory Tagging Extension (MTE) utilizes a 16-byte granularity, which is consistent with the 16-byte alignment of heap-allocated chunks. Interestingly, scaling both the number of tag bits and the granularity of tagging has no impact on memory and data traffic overhead

TABLE II: Comparison of CCTAG and other tagged architectures.

	Versatile Configurability	Efficient Multi-Policy Support	Efficient Policy Switch	Per-page Configuration	Configurable Granularity	Platform	Hardware Resource Overhead	Runtime Overhead
FlexCore	✓	✗	✗	✗	✗	LEON3	32.5%	5%-44%
Harmoni	✓	✗	✗	✗	✗	LEON3	38%-54.9%	1%-8%
lowRISC	✗	✓	✓	✗	✗	Rocket	/	/
Dover	✓	✗	✗	✗	✗	Alpha	/	/
Pump	✓	✗	✗	✗	✗	Alpha	110%	10%
CCTAG	✓	✓	✓	✓	✓	Rocket	6.77%-8.02%	4.71%-7.93%

as long as the memory granularity fits within a cache line. If ARM MTE were to operate at 32 bytes, it could support an 8-bit tag without incurring additional memory overhead.

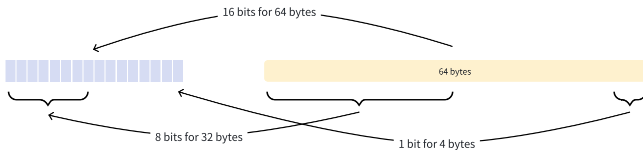

Fig. 1: Memory tag partition.

Thus, to support various tag policies, instead of providing a fixed tag granularity and tag width, the design should provide a fixed tag ratio and allow tag granularity and width to co-variate. As illustrated in Figure 1, in a 16-bit tag for the 64-byte data scenario, we could use 8 bits to tag each of the 32 bytes data, or use 1 bit for each of the 4 bytes data. Such a feature can be achieved with a mask generated based on address and access size.

2) Verify then Update: Tagged memory architectures in defense solutions typically adhere to a universal two-step process for each executed instruction. The first step involves optionally validating register and memory tag with some context. If this verification passes, the system will optionally update the tag as needed. Different tag policies vary in when and how they perform validation and updating. For instance, in DIFT, tags are propagated between registers and memory during ALU operations and may be checked when executing a branch instruction. In contrast, ARM MTE checks memory tags during every load and store operation, while updating only occurs during specialized instructions.

To integrate multiple tag-based defense mechanisms, one must apply and combine their tag verification and updating rules to the same piece of tag. A practical approach to achieve this integration is through the use of policy masks for each policy. Provided that these masks are disjoint, different policies can perform their own partial verification and updating concurrently without interference.

3) Per-Page Configuration: Different memory ranges exhibit distinct access patterns and, thus, require specialized tag policies. For instance, applying memory coloring on the stack can be challenging, as most stack memory accesses are based on the stack pointer or frame pointer. In contrast, heap memory is typically accessed through pointers returned by an allocator, making it more suitable for matching tags in pointers with memory tags. Furthermore, if two policies are applied to different memory ranges, they won't conflict with each other, even if they share the same tag mask.

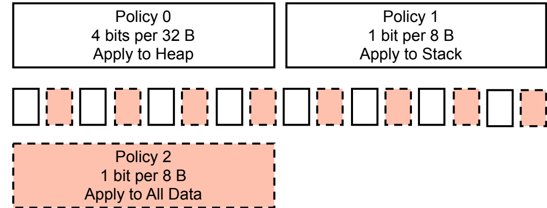

Fig. 2: Distinct policies coexist without interference.

4) Design: Based on the above intuitions, we propose a policy-centric mask design as follows:

- **Tag Policy:** Each tag policy is configured with a policy mask, granularity, and rules for tag checking and updating.
- **Per-Page Configuration:** For each page, a bitmap is used to indicate which policies should be activated when accessing that page. If a policy is not activated for the accessing page, simply generate a mask with all bits set to 0.
- **Access Mask:** Based on the policy's granularity and the memory address accessed, generate a mask to indicate which subunit of the whole piece of the tag should be checked or updated. If the policy's granularity is not the same as the memory access size, the mask should be generated based on the larger granularity. Such a design ensures coarser memory access can perform all finer subunit tag operations.
- **Tag Checking and Updating:** When memory access occurs, for each policy, the access mask is generated as described above, then AND with the policy mask to get the final mask for partial tag checking and updating.

This design provides two key benefits over configuring how individual tag bits should operate. First, it allows different tag policies to work with different granularities and tag bit widths, which is essential for supporting various defense mechanisms. Second, it allows policies to be applied to different memory ranges, maximumly utilizing the available tag bits.

Figure 2 illustrates a scenario where three distinct policies

coexist in a program without interference. For instance, policy 0, functioning similarly to MTE, provides heap protection using 4 bits for every 32 bytes. Policy 1, on the other hand, is designed to safeguard the stack's return address, utilizing a 1-bit tag for every 8 bytes. Since policies 0 and 1 operate on separate pages, they can share the same tag bits without conflict. Together, they utilize 8 of the 16 available tag bits, leaving the remaining 8 bits for policy 2. This final policy can be universally applied across all data pages.

B. `CCTAG` Overview

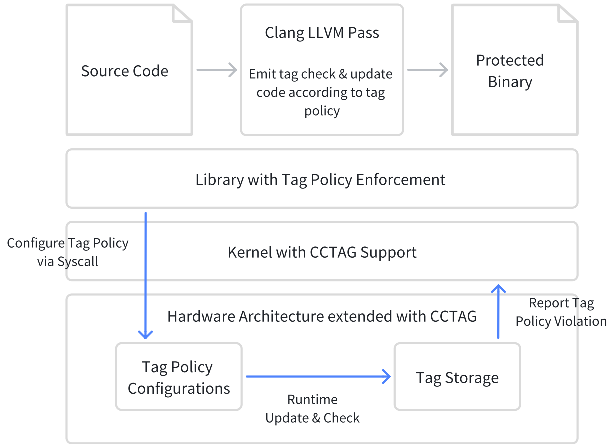

Fig. 3: The overview of `CCTAG` design.

Figure 3 presents the overview of `CCTAG` design, which is comprised of three principal components:

1) **Processor Core.** This component features tag extensions and is responsible for enforcing tag rules during execution. It ensures that all memory and operations comply with the current tag policies, integrating hardware-level security directly into the processor's operational framework.
2) **Operating System.** This layer offers services to configure and manage different tag policies, facilitating dynamic security policy adjustments based on application needs.
3) **Compilation Infrastructure.** Including both the compiler and associated libraries, this infrastructure provides a variety of tag-based protection frameworks.

For software to leverage the protections provided by `CCTAG`, most of the work should be handled by compiler and runtime library developers. This includes adding compiler passes to instrument code, implementing explicit tag checks and updates in memory management and operation functions, setting up policy configurations, and exporting necessary APIs or macros. For instance, implementing the combined defense mechanism discussed in Section VII required approximately 1,000 lines of code added to the LLVM compiler and 400 lines to the musl libc. In contrast, application developers typically only need to use the APIs provided by the library and include specific flags when compiling their programs. However, they retain the option to implement custom protections if their application requires additional security measures.

C. Processor Core Support

1) Tag Extension: In the design of `CCTAG`, tags can be placed in three distinct locations: as memory tags, register tags, and tags within pointers.

Central to the `CCTAG`'s tagged architecture are the memory tags, which are associated with memory units' physical addresses. To accommodate this, each cache line in the data cache should be extended with tag bits, and a portion of memory should be reserved for use as the tag's backing store. Uniquely, `CCTAG`'s memory tags do not conform to a fixed granularity or bit width. Instead, they require a fixed ratio, allowing for a co-variation between granularity and maximum bit width. A smaller granularity permits finer control, although the number of tag bits is limited, necessitating simpler policies. Conversely, a larger granularity provides more tag bits but may result in less precise operations. The advanced functionality in `CCTAG` is achieved by enabling partial operations of tags and supporting tag operations at the highest level of granularity, constrained by the cache line size. When memory access requires tag operations, `CCTAG` determines the relevant portion of the tag based on the configured granularity and the specific memory address involved. The memory tag ratio in the `CCTAG` design is a critical parameter. A higher ratio yields more tag bits, facilitating support for more finely-grained policies and a greater number of policies simultaneously. However, this also necessitates additional hardware resources to store and manage the tags, and a larger proportion of memory must be reserved for tags. Register tags require an extension to the register file. In `CCTAG`, register tags are primarily utilized for DIFT. In this context, tags should be transferred between registers and memory in parallel with the data. As a result, the tag ratio for register tags is equivalent to that of memory tags.

Tags in pointers are embedded within unused high bits of the pointers. `CCTAG` uses these tags as a type of restricted capability, akin to the capabilities seen in the CHERI [51] architecture. Although they lack range information due to limited bits, these tags are capable of performing pattern matching with memory tags. Their length can be equal to the largest granularity of memory tag bits, though a reduced size can still enable matching by repeat. For most applications, an 8-bit tag, which can differentiate up to 256 distinct elements, is usually sufficient.

In addition to incorporating tag storage, `CCTAG` introduces instructions to explicitly manipulate these tags. Instructions for accessing both register tags and memory tags are mandatory. For tags embedded within pointers, dedicated instructions are optional but can reduce the total number of required operations. For broader applicability, these instructions are non-privileged and accessible to user-mode libraries. Notably, such instructions are intended to be used sparingly within programs, making them difficult for attackers to reuse.

2) Tag Policy: As mentioned earlier, tag policies essentially dictate when and how to check and update tags. In `CCTAG`, such checking and updating is performed to both memory and register tags. In `CCTAG`, each memory access includes

TABLE III: Supported tag checking & updating rules.

Constant	Description
MT_CHECK_NONE	On load/store, do not check
MT_CHECK_EQUAL	On load/store, check if the memory tag matches with tag in pointer
MT_CHECK_UNCOND	On load/store, check if the memory tag is 1(or 0), subject to MT_CHECK_VAL
MT_CHECK_COND	On load/store, if the tag in pointer is 1, check if the memory tag is 1 (or 0), subject to L_MT_CHECK_VAL
L_PROP	On load, propagate the memory tag to the register tag
S_NONE	On store, do not change the memory tag
S_SET	On store, set the memory tag to 1
S_UNSET	On store, set the memory tag to 0
S_PROP	On store, propagate the register tag to the memory tag

the capability to check memory tags. If the check fails, an exception will take place, and the instruction will not be committed. The checking can involve validating each tag bit against predefined values (0 or 1), or ensuring they match the tag within the pointer. Additionally, CCTAG supports conditional checks, activated if the corresponding bit in the pointer's tag is set to 1. This feature of pointer-tag-activated conditional checking is critical because, in many tag-based defense mechanisms, only a subset of memory operations requires a tag check. An alternative approach could involve introducing special instructions dedicated to tag checking, as seen in HDFI. However, given CCTAG's support for multiple policies and bit-wise checking, this would necessitate the creation of a large number of new instructions. Such an expansion of the instruction set is impractical. Memory tag updating can only occur during store operations, as memory reads should not change the program's security state. Updating options include leaving the tag unchanged, setting it to 0 or 1, or propagating from a register's tag.

For register tags, each policy in CCTAG allows configuration of whether or not to propagate the memory tag to the register on load instructions, with a default action to write zero. In DIFT, tags should also be propagated between registers during ALU calculations. Currently, we support two options for tag propagation. One option is to OR the tag bits for all arithmetic instructions, which is suitable for taint tracking. The other option is to XOR the tag bits only for AND and SUB instructions, designed for pointer tracking. We do not include AND instructions because the result of ANDing a pointer and a value in binaries is unclear—it could be an aligned pointer or an offset. Such inaccuracies could lead to false negatives in pointer tracking. However, since pointer alignment operations are rare in most user programs, this issue is limited to a few cases. In the future, we plan to add dedicated pointer arithmetic instructions to address this issue.

The full range of supported policy configuration rules related to memory tags is enumerated in Table III. Note that CCTAG allows distinct configuration of tag-checking processes for load and store operations. We intentionally designed the rules to be simple to simplify the hardware logic. All checking rules can ultimately be converted to an optional check of the memory tag as either 0 or 1. Similarly, all updating rules can be simplified to write a 0 or 1 optionally. Provided that different tag policies utilize different masks, the combined result can be easily obtained by OR-ing the masked values. Despite their simplicity, as we will demonstrate in Section V, these rules can support a variety of defense applications.

D. Kernel Support

The kernel is responsible for configuring policies for each thread and each page, as well as handling tag mismatch exceptions. Initially, all policies are inactive and applied to no pages to ensure compatibility. Subsequently, each thread can modify tag policies and apply protections to pages through system calls. If any memory access results in a tag mismatch exception, indicating a violation of tag policy, the kernel must address this issue, potentially by terminating the process.

E. Security of CCTAG

Since CCTAG is designed to support defense mechanisms, its own security is a critical concern and must be carefully examined. Attackers could potentially target CCTAG in two ways: by exploiting its configurability to disable protection policies, or by directly manipulating tags to circumvent tag-checking mechanisms. However, both approaches are exceedingly difficult to execute.

First, attackers cannot bypass protections by altering rules, as configurations are defined by developers, embedded at compile-time, and managed by the kernel during runtime. Since CCTAG relies on a trusted kernel, attackers with non-privileged access cannot exploit the system's configurability. Exploiting the trusted kernel as a confused deputy would require the attacker to manipulate both the syscall number—typically hard-coded in the code—and its arguments, which is a non-trivial task with the present security mechanisms in place. Second, regarding tag manipulation, dedicated instructions for such operations should be infrequently encountered in typical programs. For attackers to successfully exploit these instructions would necessitate prior hijacking of control flow—an objective that CCTAG actively seeks to prevent. Consequently, we regard this attack vector as nearly impossible.

Another concern is whether the potential for user misconfiguration of policies can lead to conflicts. However, since policies are managed by a trusted kernel, the kernel has complete awareness of current policies and which memory areas use specific protections. If incompatible tag rules are applied to the same tag bits and the same memory pages, the kernel is responsible for detecting and reporting the error.

Such detection is feasible with a straightforward solution. For instance, when a syscall is made to apply a new policy to a memory area, the kernel can verify whether the tag mask of the new policy overlaps with those of existing applied policies. However, handling syscalls to modify an existing policy's tag mask is slightly more complex. The kernel must examine all memory areas using the policy to ensure that the updated tag mask does not overlap with any other policy's tag mask. In

either case, if an overlap is detected, the kernel can reject the change and report an error.

V. Security Capabilities

With CCTAG's support to check and update tags on each instruction, it can support a variety of protection primitives.

Fine-grained Permissions. By unconditionally checking tags for specific values, CCTAG enables a fine-grained permission system, enforcing read-only, write-only, or no-access permissions at sub-page levels. For instance, to make certain data in memory read-only, a policy can be configured to check that specific tag bits are not set when a write operation is attempted. Subsequently, tag `1` indicates a read-only status, and any write attempt will be blocked. These tag bits must first be cleared to modify the data, ensuring permission changes are intentional and controlled.

Sensitive Data Flow Isolation. CCTAG supports data flow isolation similar to HDFI but without dedicated load and store instructions for sensitive data. Instead, it achieves a similar effect by setting up a policy to clear tags for all storage operations, with an additional tag set instruction to mark sensitive data. The policy also mandates conditional checks on data loads; before loading sensitive data, the tag bit in the pointer is set to activate the check. Though it requires a bit more instructions to work, it allows dataflow to be separated into multiple domains, providing stronger security guarantees.

Memory Coloring. CCTAG supports a policy that matches tags within pointers to memory tags, making it readily adaptable to any protection schemes based on ARM MTE. However, unlike ARM MTE's fixed 4-bit tag per 16-byte scheme, CCTAG offers greater flexibility by allowing configurable granularity. This adaptability means that with larger alignment for memory chunks, CCTAG can accommodate more tag bits, thereby enhancing its capacity to detect memory safety violations.

Direct Information Flow Tracking. CCTAG, equipped with memory and register tags, is well-suited for adapting to track direct data flows. Compared to existing works, CCTAG offers the flexibility to support both coarse-grained tracking with a diverse array of tag types and fine-grained tracking with a limited variety of tag types. Currently, support for DIFT in CCTAG's design is limited compared to existing works specifically designed for DIFT, as it only tracks direct information flows.

These primitives enable a variety of existing protection mechanisms. Memory coloring supports spatial and temporal memory safety, while fine-grained permissions and sensitive data flow isolation provide intra-process isolation to protect pointers and sensitive data. Direct information flow tracking can also support taint-based security applications. However, CCTAG prioritizes simplicity and efficiency, which means some protection methods are excluded. For instance, explicit bounds checking with base and size info, which offers precise spatial memory safety, is not included due to the significant overhead seen in In-Fat and CHERI. Additionally, classic CFI is not supported, as code pointer integrity generally provides a stronger defense against control flow hijacking.

VI. Implementation

CCTAG is compatible with any 64-bit system that supports memory paging. For prototype, we implement it using Berkeley's open-source Rocket Chip, which features a simple five-stage, in-order execution RISC-V CPU, and make necessary modifications to the Linux kernel, LLVM, and musl libc.

A. Rocket Processor

1) Extend Data with Tag: We expand data storage and links to include tags in the D-cache, register file, and pipeline registers. The CPU-D-cache data channel now carries tag bits matching the cache line tags. We add a tag field in TileLinks for communication between the data cache, system bus, and memory bus, and implement a TBI (Top Bits Ignore) feature in RISC-V to disregard bits 55-48 reserved for tags during pointer dereferencing. Since 64-bit DDR is common, we allocate separate memory for tag storage, requiring an extra access for each DDR operation. To manage this, we introduce a TileLink module called the *data tagger* to handle tag retrieval and updates.

2) Tag Policy Configuration: Tag policy configuration is supported by adding custom supervisor CSRs for each policy, specifying tag check/update rules, granularity, and tag masks. To apply policies at the page level, each TLB entry in the data cache includes a 4-bit bitmap to activate specific policies as needed. We also modified the page table walking logic to ensure the bitmap is accurately updated.

3) Extra Instructions: We have introduced new instructions for manipulating memory tags, register tags, and pointer tags. For register tags, we provide `rtr` (read), `rtw` (write), `rts` (set), and `rtc` (clear). To optimize pointer tag operations, we added `ptw`, `pts`, and `ptc`. For memory tag manipulation, we offer two instruction sets based on granularity. The first set, `mtr` and `mtw`, enables efficient tag-setting over large memory areas. The second set, `mtrd`, `mtwd`, `mtsd`, and `mtcd`, operates at word granularity for detailed scenarios. Additionally, we added `ldp` and `sdp` to support operations like `memcpy` and `memset`, allowing load and store operations with data regardless of memory page propagation settings.

4) Memory Tag Checking and Updating: We implement partial tag operations using masked operations through a new tag control logic module, which determines expected tags and update methods. For each policy, the module calculates expected and new tag values based on configured functions, derives a mask from policy granularity and memory access address, and aggregates the masked required and update tag values. These mask, tag check, and update values are then forwarded to the data cache. Our prototype uses a blocking cache synchronized with the core's instruction execution to simplify exception handling. The data cache compares current tags with required ones under the mask and triggers exceptions on mismatches. If no issues are found and updates are needed, it performs masked tag updates. Additionally, we modified the core to handle new exceptions from tag check mismatches.

5) Optimizations: To reduce the increased memory traffic from tag storage access, we implemented several optimizations. First, we add a tag valid bit and a tag dirty bit to each D-cache line, along with an extra bit in the TileLink interface to indicate if a request requires a tag. This allows the D-cache to request only data for cache misses that don't need tag operations. Additionally, the D-cache writes back only data if the tag is neither valid nor dirty and skips fetching tags when all tag bits are written, reducing traffic when tags are rarely updated or when tagging large memory ranges. Second, we implement a tag cache for the data tagger. A small tag cache can cover a large memory range with a high hit rate, reducing direct DDR accesses for tag operations. Although this adds hardware complexity and a 2-cycle delay with added buffers for data path timing, the tag cache significantly lowers average memory delay.

B. Linux Kernel

Per-thread policy configuration is supported by extending the `thread_struct` to store configurations and saving/restoring policy CSRs during context switches. We add a system call for threads to set their policies and modified `mprotect` and `mmap` to activate tag policies on relevant pages. Additionally, the kernel initializes memory tags to zero for anonymous pages.

For interrupts and system calls, `CCTAG` uses page-granular policies with minimal kernel changes. This design prevents conflicts from user-defined policies during system calls, simplifying memory transfers between kernel and user space. When the kernel accesses user memory, it follows the same tag-checking and updating rules as user space. If a tag check fails, the memory operation is aborted, and an error code is returned to the user program. Handling interrupts requires saving both register values and their associated tags. To achieve this, we introduced new supervisor CSRs to store register tags during transitions between user and kernel space. The kernel maintains these CSRs within the trap frame, similar to regular registers.

C. Ported Defense Application

We then introduce the defense applications we ported to verify the effectiveness of `CCTAG`.

1) Return Address Protection: We use `CCTAG` 's fine-grained permissions to protect return addresses, implemented as a machine function pass added to the LLVM compiler. As demonstrated in Listing 2, this pass tags return addresses when saving them to the stack (using `mtsd` at line 6) and clears the tags before loading them (using `mtcd` at line 9). Our policy makes tagged stack words inaccessible to standard load/store instructions, ensuring return addresses remain secure and enforcing strong backward control flow integrity. To ensure compatibility, we modified musl's `longjmp` and libunwind's exception handling to clear relevant tags. In `longjmp`, tags between the old and new stack pointers are cleared, and in exception handling, return address tags are cleared during stack unwinding using dwarf information.

```
1   1090c: 01 11          addi    sp, sp, -32
2   1090e: 06 ec          sd      ra, 24(sp)
3   10910: 22 e8          sd      s0, 16(sp)
4   10912: 00 10          addi    s0, sp, 32
5   10914: 09 45          li      a0, 2
6   10916: 2b 6c a1 00    mtsd    a0, 24(sp)
7   ...
8   10950: 09 45          li      a0, 2
9   10952: 2b 78 a1 00    mtcd    a0, 24(sp)
10  10956: e2 60          ld      ra, 24(sp)
11  10958: 42 64          ld      s0, 16(sp)
12  1095a: 05 61          addi    sp, sp, 32
13  1095c: 82 80          ret
```

Listing 2: Return address protection.

```
1   1241c: 05 46          li      a2, 1
2   1241e: 88 e1          sd      a0, 0(a1)
3   12420: 2b e0 c5 00    mtsd    a2, 0(a1)
4   ...
5   13a9c: 5b 65 14 01    pts     a0, s0, 17
6   13aa0: 08 61          ld      a0, 0(a0)
7   13aa2: 10 6d          ld      a2, 24(a0)
```

Listing 3: Vtable pointer integrity.

2) Code Pointer and Vtable Pointer Integrity: `CCTAG` can enforce the pointer integrity in a similar way to HDFI. The policy uses 1 bit for every 8 bytes, where a store operation unsets the tag, and a conditional check can be performed during load. To protect code and vtable pointers, we introduce another LLVM pass, as detailed in Listing 3. In line 3, we tag protected pointers with bit `1` after storing them. In line 5, the `pts` instruction creates a modified pointer that activates tag checking before loading. Object initialization requires additional work. Intrinsic functions such as `cmalloc` or `memset` may not only initialize data to 0 but also set pointers to `NULL`. Currently, we assign a tag value of `1` to objects initialized to zero by these functions, which can lead to unnecessary tag assignments and extra time overhead. A precise type analysis could optimize this process.

3) Heap overflow and UAF Mitigation: As `CCTAG` supports pointer tag and memory tag matching, it can cover the ability of ARM MTE. We reference glibc's tagged memory support and implement memory coloring in musl's mallocng. To utilize tag bits more efficiently, we increase the `malloc` alignment to 32 bytes and make additional amendments to ensure its functionality. The modified allocator can effectively prevent adjacent heap overflow and mitigate nonadjacent overflow and Use-After-Free (UAF) vulnerabilities.

4) Dangling Pointer Nullification: Minesweeper[21] holds freed memory chunks in a quarantine pool until it reaches a set proportion of the heap, then performs a full memory sweep to check for pointers referencing these chunks. If no such pointers exist, the chunks are safely released. `CCTAG`'s DIFT allows tracking pointers without false positives, optimizing this process similarly to Cornucopia [52], [23]. Before returning a pointer, the allocator tags the register, and this tag propagates through ALU operations and between registers and memory. During the sweep, a memory word with a tag is guaranteed to be a pointer targeting a heap chunk. If the pointer points to a quarantined chunk, we can nullify it. Furthermore, if a cache

line's DIFT tag is clear, it contains no heap pointers, allowing us to skip the line and accelerate the sweeping process. After the sweep, no pointers will be left pointing to quarantined chunks, allowing them to be safely released.

D. Other Possible Applications

Besides the previously mentioned defenses, CCTAG can theoretically support a wide range of other defenses and facilitate analysis and debugging.

1) Intra-object Overflow Detection: CCTAG can detect both inter and intra object overflows by adding a policy with a disjoint mask and finer granularity, such as 1-bit tags per 4 bytes. By alternately assigning tags (e.g., 0 and 1) to different fields within a struct and adjusting pointers before accessing them, CCTAG can identify adjacent intra-object overflows.

2) Type Confusion Detection: CCTAG can detect and mitigate type confusion, common in dynamically-typed languages and caused by memory safety violations, by assigning unique tags to different types, initializing these tags at construction, and tagging pointers before accessing object fields.

3) Memory Watch Points: Without hardware support, memory watchpoints are slow because GDB must single-step and check memory after each instruction. On x86, GDB can efficiently watch up to four bytes using debug registers [24], which is often insufficient for debugging needs. CCTAG can use its fine-grained permissions to implement memory watch points efficiently and without limitations.

4) Taint Analysis: CCTAG's DIFT primitive enables flexible taint analysis. Unlike existing tagged architectures, it supports both fine-grained taint tracking with limited types and coarse-grained tracking with more types.

VII. Evaluation

This section evaluates CCTAG with our prototype and ported defense applications to answer the following questions:

- **Lightweight Profile:** Is it possible to implement the required features of CCTAG on hardware in a lightweight manner?
- **Performance Efficiency:** How much execution time and memory overhead does CCTAG incur to the system?
- **Integration Efficiency:** When multiple defense applications are integrated, is the overhead still acceptable?
- **Security Effectiveness:** Does defense mechanisms ported into CCTAG can accurately prevent attacks?

A. Experimental setup

We evaluated CCTAG by instantiating both the original and modified RISC-V Rocket Cores using the Rocket Chip Generator and synthesizing them on an FPGA with Vivado. The setup includes a blocking D-cache with PLRU replacement policy and an FPU, keeping other settings default. The prototype features a maximum memory tag granularity of 64 bytes with a 16-bit tag per granule. A 4-way associative tag cache with 64-byte lines is placed between the broadcast hub and memory bus; we test configurations with both 8 sets and 32 sets, corresponding to tag cache sizes of 2KB and 8KB, respectively. Each integer register has a 2-bit tag, and pointer tags use 8 bits. Finally, the system supports up to four distinct tag policies. We use LLVM 15.0.7 to compile the kernel and user mode codes. The kernel is Linux 6.1.62 with default configurations. For the standard library, we utilize LLVM's runtime library (compiler-rt, libcxx, libcxxabi, libunwind) along with musl 1.2.5. The FPGA development board is the Xilinx Kintex UltraScale (XCKU060), with 4GB of RAM on board.

B. Hardware Resource Consumption

Table IV presents CCTAG's overhead of hardware resource cost in FPGA with the parameter mentioned previously. It shows a moderate increase in resource utilization, with Look-Up Tables (LUTs) expanding by approximately 6.77% and registers (FFs) by around 8.02%. In addition, it utilizes 10 block rams to store tags in both D-cache and the data tagger's cache. Importantly, this augmentation had a negligible effect on the system's timing. On the Kintex Ultrascale platform, the original Rocket Core operates at a peak frequency of 125 MHz. Our CCTAG-enhanced core achieves the same frequency.

We compare CCTAG's hardware resource overhead with CHERI and PUMP to highlight its lightweight profile. The Flute microarchitecture [9] is an open-source, 5-stage in-order RISC-V CPU from Bluespec, with a complexity similar to that of the Rocket. Previous work has extended Flute to support CHERI-RISC-V [12]. We synthesized both versions using Vivado. The synthesis stage estimated results indicate that adding CHERI support increases LUTs by 75.8% and FFs by 59.4%. PUMP, while not providing an RTL implementation, reports an estimated area overhead exceeding 110%, even though it employs a more complex processor with L2 caches.

These findings demonstrate that CCTAG *features low-cost and lightweight hardware implementation.*

C. Performance Overhead

We evaluate the performance of CCTAG enhanced system with SPEC CINT2006 and CINT2017 benchmarks. We include CINT2006 benchmark for two reasons: 1) our development board has only 4GB of RAM, which limits us to running only parts of CINT2017's reference workload; 2) many existing works also use CINT2006 benchmark for evaluation, making it easier to compare our results with theirs.

In SPEC CINT2006, benchmark 400.perlbench is excluded due to compilation issues. And in benchmark 471.omnetpp, we fix a semantic mistake that related to union in the source code to make it run correctly under our defense mechanisms. The rest of the benchmarks are compiled without any source code modifications. Regarding SPEC CINT2017, we can run only the standard reference workloads for 620.omnetpp_s, 623.xalancbmk_s, 625.x264_s, 641.leela_s. For 631.deepsjeng_s, we disable the BIGMEM option; otherwise, it would require about 7GB of RAM to run regardless of the input. Benchmarks 602.gcc_s, mcf_s, and 657.xz_s experience out-of-memory (OOM) errors when

TABLE IV: Hardware resource cost of the baseline and CCTAG when synthesized on an FPGA.

	RISC-V Rocket Cores					Whole Systems					
	#LUT	%	#FF	%	#BRAM	#LUT	%	#FF	%	#BRAM	Worst Neg Slack (ns)
baseline	34,039	_	14,939	_	20	57,298	_	48,448	_	115	0.50
CCTAG	36,342	+6.77%	16,137	+8.02%	30	59,616	+4.05%	49,701	+2.59%	125	0.53

running the reference workloads, so we use the train workloads for these cases instead.

In addition to the baseline experiment on an unmodified Rocket Chip, we conduct 6 additional experiments on a CCTAG-enhanced architecture equipped with 2KB tag cache, under the following scenarios:

- **No Protection.** Unmodified software running on the CCTAG-enhanced architecture.
- **Return Address Protection.** All return addresses are protected by fine-grained permission (§ VI-C1).
- **Code Pointer & Vtable Pointer Protection.** Code pointer and vtable Pointer are protected by sensitive information isolation (§ VI-C2).
- **Heap Protection.** Heap chunks are colored to mitigate overflow and UAF (§ VI-C3).
- **Dangling Pointer Sweeping.** Use quarantine and track heap pointers to scan and nullify dangling pointers (§ VI-C4).
- **Integrated Protection.** The combination of return address protection, code pointer and vtable pointer protection, heap protection. Dangling pointer sweeping is not included, as heap protection through memory coloring already mitigates dangling pointer issues.

We also tested a CCTAG-enhanced architecture with an 8KB tag cache to evaluate the impact of tag cache size. Although larger than the previous setup, the 8KB cache remains small compared to the Rocket architecture's 64KB L1 cache.

All benchmarks are compiled with the O2 optimization and statically linked, with different protection LLVM passes enabled and linked to the specific protection's corresponding standard library. Since Rocket is an in-order processor without an L2 cache, the extra memory access for tags could significantly impact performance. To mitigate this, we chose not to enable tagging on heap chunks allocated directly from mmap, keeping the tag's working set relatively small. A similar approach is used in the Scudo allocator [3], which only tags chunks smaller than 0x10000. Before execution, all executables and input data of benchmarks are placed on *tmpfs*, an in-memory file system, to eliminate performance impacts from disk operation. The only exception is that the input for 625.x264_s is too large and therefore not transferred to *tmpfs*.

All benchmarks were completed under all protection scenarios, demonstrating that our processor and kernel modifications maintain backward compatibility and that our protections have no false positives. Figures 4 and Figures 5 illustrate the runtime overhead for each benchmark suite under various protection settings.

1) Individual Protections: The no-protection scenario incurs a runtime overhead of around 1.79% / 1.35% (SPEC CINT2006 / CINT2017) on average, which arises primarily from the extra memory access delay introduced by Tilelink buffer. The return address protection, code pointer and vtable pointer protection, and heap protection scenarios exhibit an average runtime overhead of 4.38% / 4.29%, 5.03% / 7.72%, and 4.72% / 5.74%, respectively. With code pointer and vtable pointer protection, 464.h264ref exhibit unexpected performance gains, which we tentatively attribute to our modifications to memory manipulation functions. The performance gain under integrated protection should be attributed to the same reason.

The dangling pointer sweeping scenario incurs overhead at 6.97% / 7.12%. In addition to the extra memory traffic for tags and the memory sweeping procedure, the overhead also arises from the quarantine itself, specifically the delayed memory reuse. As reported in Cornucopia [52], in the worst-case scenario (471.omnetpp), the delay caused by quarantine accounts for a third of the total overhead. Compared to Cornucopia (2.0%) and Minesweeper (5.4%), our evaluation shows a higher overhead because our single-processor prototype cannot perform parallel memory sweeping. However, it achieves better CPU utilization than Minesweeper (9.6%).

Regarding memory overhead, the 16-bit tag for a 64-byte data configuration results in 3.125% (1/32) of DRAM being allocated for tag storage. This overhead is inherent to the tagged architecture, independent of the software it runs. Additionally, memory overhead for return address and code/vtable pointer protection is minimal (under 1%). Heap protection incurs an average overhead of 5.57% / 5.32%, primarily due to changes in allocator alignment. Musl's mallocng allocator uses per-object headers, which expand significantly with increased alignment. If an allocator without per-object headers, such as jemalloc, were used, this memory overhead could be reduced.

2) Integrated Protection: For integrated protection, the runtime overhead averages around 6.68% / 10.88%. Since individual protections share the same tag cache and the overhead for extra memory access, the overhead of integrated protection is lower than the sum of the overheads from individual protection mechanisms (13.98% / 17.32%).

SPEC CINT2017 generally incurs higher runtime overhead than CINT2006, likely due to the increased complexity, which expands the tag's working set and leads to more L1 and tag cache misses, as well as additional memory accesses for tags. As Rocket is an in-order, single-issue processor, these memory delays significantly impact performance. When increasing tag cache size from 2KB to 8KB, the overhead dropped to 4.71% / 7.93%, respectively. We anticipate these overheads could be further reduced in a more realistic setup with a larger, multilevel cache and out-of-order execution.

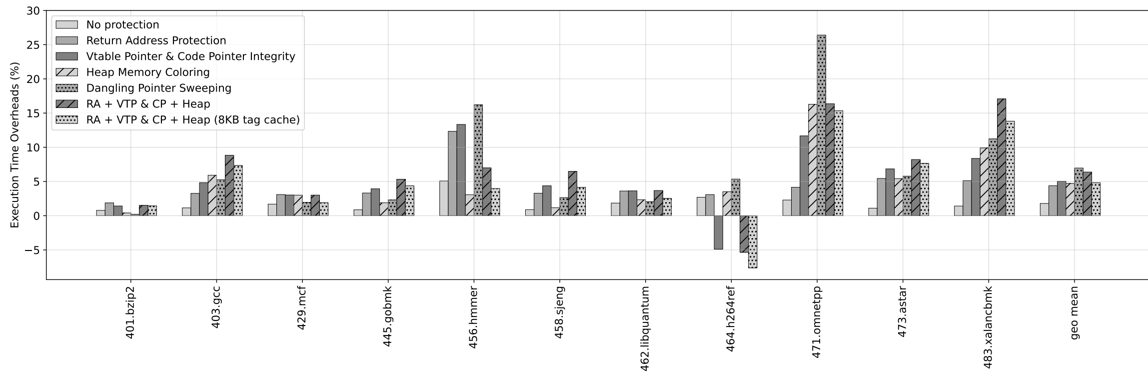

Fig. 4: Relative runtime overhead of CCTAG on SPEC CINT2006

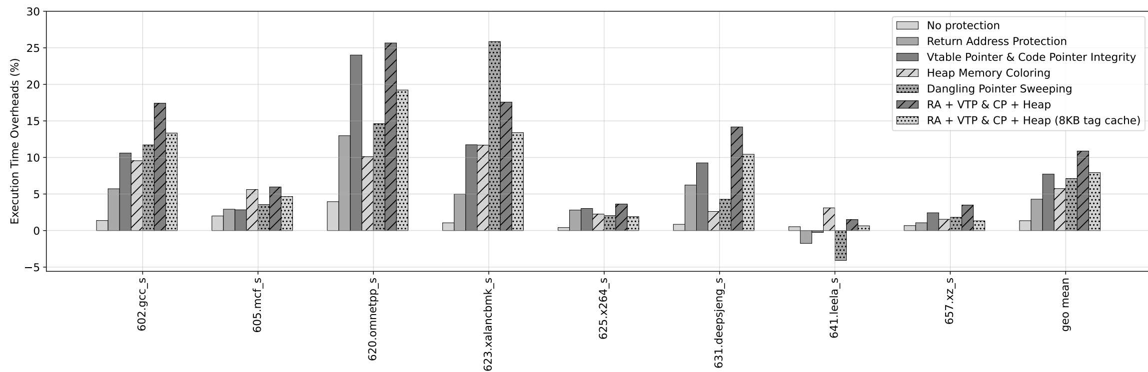

Fig. 5: Relative runtime overhead of CCTAG on SPEC CINT2017

TABLE V: Relative runtime overhead of CCTAG's integrated protection, compared to PACTight + MTE, PUMP's composite policy, and CHERI's pure capability.

	PACTight + MTE	PUMP	CHERI	CCTAG
401.bzip2	1.5%	2.7%	4.7%	1.4%
403.gcc		10.6%		7.3%
429.mcf	2.3%	9.6%		1.9%
445.gobmk	2.8%	1.1%	26.6%	4.4%
456.hmmer	2.8%	2.1%	2.3%	4.0%
458.sjeng	-0.5%	0.4%	19.7%	4.1%
462.libquantum	3.3%	4.7%		2.5%
464.h264ref	2.3%	0.2%	26.7%	-7.6%
471.omnetpp	21.6%	54.0%	100.7%	15.3%
473.astar	5.8%	11.4%	7.0%	7.7%
483.xalancbmk	14.5%		86.5%	13.8%
geo mean	5.4%	8.8%	30.5%	4.8%

We compare the performance of CCTAG's integrated protection on SPEC CINT2006 with the combination of ARM PA-based CFI and MTE-based heap protection, PUMP's composite policies [16] and CHERI's pure capability protection [50]. The results are shown in Table V. Since ARM PA and MTE are roughly orthogonal, we can add their overheads to approximate the combined overhead. PACTight [30] is the state-of-the-art solution using ARM PA to achieve control flow integrity. For ARM MTE, we could not find any evaluation on the SPEC CINT2006 benchmark suite. Therefore, we use our heap protection overhead to approximate the MTE overhead. Because our prototype adopts the same memory tag ratio as MTE, the overhead should be similar. A test with Google's Pixel 8 on Geekbench 6 [48] shows an average overhead of 7.1%, which is close to ours. CCTAG's integrated protection incurs an average overhead of 4.71%, similar to the combined overhead of PACTight and MTE at 5.45%. Nonetheless, our integrated protection offers stronger security than PACTight. First, PACTight requires a meta table to track the context of each pointer's signature, which can be corrupted by an attacker. Second, ARM PA's security relies on the QARMA encryption algorithm [6]. If QARMA is compromised, ARM PA's security will be as well. Lastly, our integrated protection can safeguard not only pointers but also other sensitive data, such as secret keys or credentials, which ARM PA cannot protect. PUMP's composite policies provide similar security to ours but with higher overhead. Additionally, PUMP's integrated protection overhead exceeds the combined overhead of individual protections [17], making it non-scalable. CHERI's pure capability model offers stronger security guarantees but incurs substantially higher overhead.

In summary, the results show that CCTAG *supports com-*

bined tag policies and multiple protection applications simultaneously while maintaining modest runtime and memory overhead.

D. Security Effectiveness

In this subsection, we assess the effectiveness of defense mechanisms powered by `CCTAG` with real-world CVEs and various exploitation techniques. Test programs are compiled with integrated protection unless otherwise noted.

1) Real World CVEs Mitigation: The `CCTAG`-enhanced protection is capable of mitigating a wide range of CVEs, rendering them either non-exploitable or significantly harder to exploit. Table VI summarizes the real-world CVEs we test on `CCTAG`.

- **Spatial Memory Safety** With its memory coloring mechanism, `CCTAG` can deterministically prevent linear buffer overflows. We test this capability using three cases: a buffer over-read (CVE-2022-40320), a buffer over-write (CVE-2018-18557), and a buffer under-write (CVE-2018-8905). In all three tested programs, a segmentation fault is triggered due to a tag mismatch as soon as the memory access exceeds the bounds. For non-linear buffer overflows, we test an out-of-bound read (CVE-2020-28603) and an out-of-bound write (CVE-2018-8905). Both cases allow memory access to cross into another tagged area, resulting in segmentation faults with a probability of 14/15.
- **Temporal Memory Safety** In single-threaded programs where no new memory allocation occurs after the victim chunk is freed, `CCTAG` can reliably detect UAF and double-free vulnerabilities since the freed chunk's tag is stripped. We verify this capability using CVE-2021-3518 (UAF) and CVE-2022-39170 (Double Free). If the freed chunk can be reallocated, there remains a 14/15 probability of detecting UAF and double-free cases. We test this scenario with CVE-2023-45666.
- **Use of Uninitialized Memory** The use of uninitialized memory is an instance of undefined behavior that can lead to serious consequences. For example, leaking an uninitialized pointer can bypass Address Space Layout Randomization, while using an uninitialized pointer could result in memory errors. `CCTAG` mitigates this type of vulnerability by using tags to track memory initialization status. To enable this feature, we introduce a new policy and modify the allocator. The allocator assigns a tag of `1` to newly allocated memory, marking it as uninitialized. When a write occurs, the tag is cleared. Unfortunately, because copying partially initialized objects is a valid operation, we cannot force all loads to unconditionally check that the tag is `0` to ensure the memory has been properly initialized. Therefore, the check should be conditional on critical operations. We validate this protection mechanism using CVE-2023-45663, which can load uninitialized memory into an image and potentially leak residual data. With the policy above and a modification to buffer copy that enables checking, a segmentation fault is triggered when copying the uninitialized data to the image.

TABLE VI: Effectiveness against real world CVEs.

CVE No.	Project	Vulnerability Type	Effectiveness
CVE-2022-40320	libconfuse	Buffer Over Read	✓
CVE-2018-18557	libtiff	Buffer Over Write	✓
CVE-2018-8905	libtiff	Buffer Under Write	✓
CVE-2021-3518	libxml2	UAF	✓
CVE-2022-39170	libdwarf	Double Free	✓
CVE-2020-28603	libcgal	Out of Bound Read	⊕
CVE-2018-12900	libtiff	Out of Bound Write	⊕
CVE-2023-45666	stb_image	Double Free	⊕
CVE-2023-45663	stb_image	Memory Disclosure	⊕
CVE-2018-12900	njs	Type Confusion	⊕

✓: complete protection
⊕: probabilistic protection or conditional protection

- **Type Confusion** As noted previously, type confusion vulnerabilities can be detected and mitigated by assigning unique tags to different data types. Since we have not implemented this feature as an automated compiler pass, we test it using CVE-2021-46463 with manual modification. We add code to tag the promise struct in the constructor and tag the pointer when it is used as a promise pointer. When other pointer types are confused as a promise pointer, since the target object is not tagged, the tag mismatch triggers an exception.

2) Exploitation Prevention: Even if an attacker manages to bypass the protections against the CVEs discussed in the previous section, `CCTAG` provides additional barriers to exploitation. In this section, we disable the memory coloring policy and apply various exploitation techniques to evaluate `CCTAG`'s resilience.

- **Return-Oriented Programming** Since the return address cannot be accessed until the function is about to return, Return-Oriented Programming (ROP) becomes entirely unfeasible. We tested these features using CVE-2021-20294, which provides an Out-of-Bounds Write primitive. When attempting to write to the return address, the system triggers a segmentation fault.
- **Jump-oriented programming** Jump-oriented programming (JOP) attacks rely on chaining gadgets that use indirect jumps instead of returns. Typically, it requires control of both saved registers and a code pointer. `CCTAG` can isolate both code pointers and saved registers to defend against it. We test the effectiveness using the example in the original paper [8], where an intra-object overflow allows an attacker to overwrite a `jmp_buf` struct. Without extra modification, integrated protection can detect the injected code pointer. `CCTAG` can also protect the saved registers in a similar way to protecting the return address. Before `setjump` returns, set tag bits to prevent access to the saved registers, and clear the tag at the start of `longjmp`.
- **COOP and COOPLUS** COOP and COOPLUS exploit legitimate control flow transfers between objects' virtual calls. We test with the example provided by COOPLUS [11]. `CCTAG` counters the attack by refusing to use the forged vtable pointer

in the counterfeit object.

• **Data-Only Attacks** Even without direct control flow manipulation, data-only attacks can alter program behavior by corrupting sensitive data. Testing with the example from [29] verified that CCTAG effectively safeguards sensitive data from tampering to defend against such attacks. Specifically, we protect the g_is_root variable with fine-grained permission primitive, ensuring only intended writes can update it. Any unintended overwrite attempt triggers an exception.

The results conclusively demonstrate that *the defense mechanisms integrated into* CCTAG *can effectively and accurately prevent a range of attacks.*

VIII. Conclusion

Memory safety violations pose a serious threat to real-world programs. Nonetheless, existing deployable defenses offer limited protection and are vulnerable to advanced attacks. To address the challenge of defense integration, we introduce CCTAG, a configurable and combinable tagged architecture optimized for integrating various tag-based defense mechanisms.

We implement the CCTAG prototype based on the RISC-V architecture and conduct extensive evaluation on an FPGA development board. Results indicate that CCTAG is a low-cost, lightweight solution capable of providing a broad range of protections effectively and efficiently. It incurs approximately 8% hardware overhead, in stark contrast to CHERI's over 50% and PUMP's 110%. The integrated protection demonstrates satisfactory runtime overhead of 4.71% on SPEC CINT2006 and 7.93% on CINT2017 benchmarks. Experiments with diverse real-world CVEs and exploitation techniques further confirm CCTAG 's strong security capabilities.

Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments and suggestions that greatly improved the quality of this paper. We also thank Yangyu Chen for his open-source repositories, which have greatly facilitated the development of SoC systems* and hardware function debugging†. This research was supported, in part, by the National Natural Science Foundation of China (U24A20337), National Key R&D Program of China (2021YFB2701000) and the Joint Research Center for System Security, Tsinghua University (Institute for Network Sciences and Cyberspace) - Science City (Guangzhou) Digital Technology Group Co., Ltd.

References

[1] "Queue hardening enhancements," https://security.googleblog.com/2019/05/queue-hardening-enhancements.html, 2019, accessed: 2024-04-25.

[2] "Trends, challenges, and shifts in software vulnerability mitigation," 2019, accessed: 2024-04-25.

[3] "Strengthening the shield: MTE in memory allocators," July 2023, accessed: 2024-07-09. [Online]. Available: https://www.darknavy.org/blog/strengthening_the_shield_mte_in_memory_allocators/

*https://github.com/cyyself/pblaze_soc

†https://github.com/cyyself/soc-simulator

[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, November 2009. [Online]. Available: https://doi.org/10.1145/1609956.1609960

[5] ARM, "Memory tagging extension," https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[6] R. Avanzi, "The QARMA block cipher family," 2016, accessed: 2024-07-10. [Online]. Available: https://eprint.iacr.org/2016/444.pdf

[7] A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach, "A verified information-flow architecture," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 165–178. [Online]. Available: https://doi.org/10.1145/2535838.2535839

[8] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 30–40. [Online]. Available: https://doi.org/10.1145/1966913.1966919

[9] Bluespec, Inc., "Flute: RISC-V CPU, simple 5-stage in-order pipeline, for low-end applications needing MMUs and some performance," https://github.com/bluespec/Flute, 2024, accessed: 2024-10-16.

[10] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-Flow bending: On the effectiveness of Control-Flow integrity," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, August 2015, pp. 161–176. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini

[11] K. Chen, C. Zhang, T. Yin, X. Chen, and L. Zhao, "VScape: Assessing and escaping virtual call protections," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021, pp. 1719–1736. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/chen-kaixiang

[12] CTSRD-CHERI, "Flute: A RISV-V CPU with capability enhancements for research on security extensions," https://github.com/CTSRD-CHERI/Flute, 2024, accessed: 2024-10-16.

[13] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A flexible information flow architecture for software security," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 482–493. [Online]. Available: https://doi.org/10.1145/1250662.1250722

[14] D. Y. Deng, D. Lo, G. Malysa, S. Schneider, and G. E. Suh, "Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 137–148.

[15] D. Y. Deng and G. E. Suh, "High-performance parallel accelerator for flexible and efficient run-time monitoring," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, 2012, pp. 1–12.

[16] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, B. C. Pierce, and A. DeHon, "Architectural support for software-defined metadata processing," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 487–502. [Online]. Available: https://doi.org/10.1145/2694344.2694383

[17] 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 *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2611765.2611773

[18] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient protection of Path-Sensitive control security," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, August 2017, pp. 131–148. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ding

[19] G. J. Duck and R. H. Yap, "Heap bounds protection with low fat pointers," in *Proceedings of the 25th International Conference on Compiler Construction*, 2016, pp. 132–142.

[20] G. J. Duck, R. H. Yap, and L. Cavallaro, "Stack bounds protection with low fat pointers." in *NDSS*, 2017.

[21] M. Erdős, S. Ainsworth, and T. M. Jones, "Minesweeper: a "clean sweep" for drop-in use-after-free prevention," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 212–225. [Online]. Available: https://doi.org/10.1145/3503222.3507712

[22] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 901–913. [Online]. Available: https://doi.org/10.1145/2810103.2813646

[23] N. W. Filardo, B. F. Gutstein, J. Woodruff, J. Clarke, P. Rugg, B. Davis, M. Johnston, R. Norton, D. Chisnall, S. W. Moore, P. G. Neumann, and R. N. M. Watson, "Cornucopia reloaded: Load barriers for cheri heap temporal safety," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 251–268. [Online]. Available: https://doi.org/10.1145/3620665.3640416

[24] GDB Project, "Internals watchpoints," n.d., accessed: 2024-07-09. [Online]. Available: https://sourceware.org/gdb/wiki/Internals%20Watchpoints

[25] R. T. Gollapudi, G. Yuksek, D. Demicco, M. Cole, G. Kothari, R. Kulkarni, X. Zhang, K. Ghose, A. Prakash, and Z. Umrigar, "Control flow and pointer integrity enforcement in a secure tagged architecture," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2974–2989.

[26] F. Gorter, K. Koning, H. Bos, and C. Giuffrida, "Dangzero: Efficient use-after-free detection via direct page table access," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1307–1322. [Online]. Available: https://doi.org/10.1145/3548606.3560625

[27] F. Gorter, T. Kroes, H. Bos, and C. Giuffrida, "Sticky Tags: Efficient and deterministic spatial memory error mitigation using persistent memory tags," in *2024 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2024, pp. 4239–4257. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00263

[28] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing unique code target property for control-flow integrity," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1470–1486. [Online]. Available: https://doi.org/10.1145/3243734.3243797

[29] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 969–986.

[30] M. Ismail, A. Quach, C. Jelesnianski, Y. Jang, and C. Min, "Tightly seal your sensitive pointers with PACTight," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, August 2022, pp. 3717–3734. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/ismail

[31] M. Ismail, J. Yom, C. Jelesnianski, Y. Jang, and C. Min, "VIP: Safeguard value invariant property for thwarting critical memory corruption attacks," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1612–1626. [Online]. Available: https://doi.org/10.1145/3460120.3485376

[32] S. Jero, N. Burow, B. Ward, R. Skowyra, R. Khazan, H. Shrobe, and H. Okhravi, "TAG: Tagged architecture guide," *ACM Comput. Surv.*, vol. 55, no. 6, December 2022. [Online]. Available: https://doi.org/10.1145/3533704

[33] H. Kannan, M. Dalton, and C. Kozyrakis, "Decoupling dynamic information flow tracking with a dedicated coprocessor," in *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, 2009, pp. 105–114.

[34] M. Khandaker, A. Naser, W. Liu, Z. Wang, Y. Zhou, and Y. Cheng, "Adaptive call-site sensitive control flow integrity," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019, pp. 95–110.

[35] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: an experimental study of dram disturbance errors," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, p. 361–372, June 2014. [Online]. Available: https://doi.org/10.1145/2678373.2665726

[36] LowRISC Team. (2017) Tag support in the Rocket Core. [Online]. Available: https://lowrisc.org/docs/minion-v0.4/tag_core/

[37] D. P. McKee, Y. Giannaris, C. Ortega, H. E. Shrobe, M. Payer, H. Okhravi, and N. Burow, "Preventing kernel hacks with hakcs." in *NDSS*. The Internet Society, 2022. [Online]. Available: http://dblp.uni-trier.de/db/conf/ndss/ndss2022.html#McKeeGOSPOB22

[38] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: highly compatible and complete spatial memory safety for c," *SIGPLAN Not.*, vol. 44, no. 6, p. 245–258, June 2009. [Online]. Available: https://doi.org/10.1145/1543135.1542504

[39] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software abstraction for intel memory protection keys (intel MPK)," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 241–254. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/park-soyeon

[40] 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 *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 745–762.

[41] V. Shanbhogue, D. Gupta, and R. Sahita, "Security analysis of processor instruction set architecture for enforcing control-flow integrity," in *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3337167.3337175

[42] K. Sinha and S. Sethumadhavan, "Practical memory safety with rest," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. IEEE Press, 2018, p. 600–611. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00056

[43] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: Hardware-assisted data-flow isolation," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 1–17.

[44] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "SoK: Sanitizing for security," *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 1275–1295, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:48364047

[45] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," *SIGARCH Comput. Archit. News*, vol. 32, no. 5, p. 85–96, October 2004. [Online]. Available: https://doi.org/10.1145/1037947.1024404

[46] G. T. Sullivan, A. DeHon, S. Milburn, E. Boling, M. Ciaffi, J. Rosenberg, and A. Sutherland, "The Dover inherently secure processor," in *2017 IEEE International Symposium on Technologies for Homeland Security (HST)*, 2017, pp. 1–5.

[47] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13. USA: IEEE Computer Society, 2013, p. 48–62. [Online]. Available: https://doi.org/10.1109/SP.2013.13

[48] W. Tan, C. Li, Y. Chen, Y. Li, C. Zhang, and J. Wu, " ROLoad-PMP: Securing Sensitive Operations for Kernels and Bare-Metal Firmware ," *IEEE Transactions on Computers*, vol. 73, no. 12, pp. 2722–2733, December 2024. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/TC.2024.3449105

[49] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "Flexitaint: A programmable accelerator for dynamic taint propagation," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, 2008, pp. 173–184.

[50] R. N. M. Watson, J. Clarke, P. Sewell, J. Woodruff, S. W. Moore, G. Barnes, R. Grisenthwaite, K. Stacer, S. Baranga, and A. Richardson, "Early performance results from the prototype Morello microarchitecture," 2023. [Online]. Available: https://ctsrd-cheri.github.io/morello-early-performance-results/cover/index.html

[51] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "CHERI: A hybrid

capability-system architecture for scalable software compartmentalization," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 20–37.

[52] N. Wesley Filardo, B. F. Gutstein, J. Woodruff, S. Ainsworth, L. Paul-Trifu, B. Davis, H. Xia, E. Tomasz Napierala, A. Richardson, J. Baldwin, D. Chisnall, J. Clarke, K. Gudka, A. Joannou, A. Theodore Markettos, A. Mazzinghi, R. M. Norton, M. Roe, P. Sewell, S. Son, T. M. Jones, S. W. Moore, P. G. Neumann, and R. N. M. Watson, "Cornucopia: Temporal safety for CHERI heaps," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 608–625.

[53] B. Wickman, H. Hu, I. Yun, D. Jang, J. Lim, S. Kashyap, and T. Kim, "Preventing Use-After-Free attacks with fast forward allocation," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021, pp. 2453–2470. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/wickman

[54] E. Witchel, J. Rhee, and K. Asanović, "Mondrix: Memory isolation for linux using mondriaan memory protection," *SIGOPS Oper. Syst. Rev.*, vol. 39, no. 5, p. 31–44, October 2005. [Online]. Available: https://doi.org/10.1145/1095809.1095814

[55] J. Xu, M. Xie, C. Wu, Y. Zhang, Q. Li, X. Huang, Y. Lai, Y. Kang, W. Wang, Q. Wei, and Z. Wang, "Panic: PAN-assisted intra-process memory isolation on ARM," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 919–933. [Online]. Available: https://doi.org/10.1145/3576915.3623206

[56] S. Xu, W. Huang, and D. Lie, "In-fat pointer: hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 224–240.

[57] H. Ye, S. Liu, Z. Zhang, and H. Hu, "VIPER: Spotting Syscall-Guard variables for Data-Only attacks," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, August 2023, pp. 1397–1414. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/ye

[58] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis, "Hardware enforcement of application security policies using tagged memory," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 225–240.

[59] M. T. I. Ziad, M. A. Arroyo, E. Manzhosov, R. Piersma, and S. Sethumadhavan, "No-FAT: Architectural support for low overhead memory safety checks," in *ISCA-48: Proceedings of the 48th Annual International Symposium on Computer Architecture, Worldwide Event*, 2021.