CrypTag: Thwarting Physical and Logical Memory Vulnerabilities using Cryptographically Colored Memory

Pascal Nasahl
pascal.nasahl@iaik.tugraz.at
Graz University of Technology

Robert Schilling
robert.schilling@iaik.tugraz.at
Graz University of Technology

Mario Werner
mario.werner@iaik.tugraz.at
Graz University of Technology

Jan Hoogerbrugge
jan.hoogerbrugge@nxp.com
NXP Semiconductors Netherlands

Marcel Medwed
marcel.medwed@nxp.com
NXP Semiconductors Austria

Stefan Mangard
stefan.mangard@iaik.tugraz.at
Graz University of Technology
Lamarr Security Research

ABSTRACT

Memory vulnerabilities are a major threat to many computing systems. To effectively thwart spatial and temporal memory vulnerabilities, full logical memory safety is required. However, current mitigation techniques for memory safety are either too expensive or trade security against efficiency. One promising attempt to detect memory safety vulnerabilities in hardware is memory coloring, a security policy deployed on top of tagged memory architectures. However, due to the memory storage and bandwidth overhead of large tags, commodity tagged memory architectures usually only provide small tag sizes, thus limiting their use for security applications.

Irrespective of logical memory safety, physical memory safety is a necessity in hostile environments prevalent for modern cloud computing and IoT devices. Architectures from Intel and AMD already implement transparent memory encryption to maintain confidentiality and integrity of all off-chip data. Surprisingly, the combination of both, logical and physical memory safety, has not yet been extensively studied in previous research, and a naïve combination of both security strategies would accumulate both overheads.

In this paper, we propose CrypTag, an efficient hardware/software co-design mitigating a large class of logical memory safety issues and providing full physical memory safety. At its core, CrypTag utilizes a transparent memory encryption engine not only for physical memory safety, but also for memory coloring at hardly any additional costs. The design avoids any overhead for tag storage by embedding memory colors in the upper bits of a pointer and using these bits as an additional input for the memory encryption. A custom compiler extension automatically leverages CrypTag to detect logical memory safety issues for commodity programs and is fully backward compatible.

For evaluating the design, we extended a RISC-V processor with memory encryption with CrypTag. Furthermore, we developed a LLVM-based toolchain automatically protecting all dynamic, local, and global data. Our evaluation shows a hardware overhead of less than 1 % and an average runtime overhead between 1.5 % and 6.1 % for thwarting logical memory safety vulnerabilities on a system already featuring memory encryption. Enhancing a system with memory encryption typically induces a runtime overhead between 5 % and 109.8 % for commercial and open-source encryption units.

KEYWORDS

memory safety; tagged memory; memory coloring; memory encryption; RISC-V

1 INTRODUCTION

According to MITRE [35], three out of ten of the most common software weaknesses in 2019 leading to security vulnerabilities are owed to logical memory safety violations. Memory vulnerabilities, which exploit spatial or temporal memory bugs, are the foundation for more sophisticated attack techniques, such as return-oriented programming (ROP) [50] or data-oriented programming (DOP) [20]. Consequently, due to the high impact of memory vulnerabilities, defense strategies, such as code- and data-pointer integrity [24, 28] or the protection of the control-flow [30], were introduced in the past. However, these concepts only complicate the exploitation of a memory vulnerability, but do not fix the root cause. To completely thwart memory vulnerabilities, full logical memory safety for all classes of memory allocations is necessary [53]. Unfortunately, software solutions providing memory safety, such as SoftBound [37] or CETS [38], typically add significant overhead and increase costs if deployed on a larger scale. High performance penalty can be counteracted with hardware assistance. A promising attempt to detect memory safety violations with hardware support are tagged memory architectures [46]. Tagged memory assigns additional metadata to the memory, enforcing different security policies [61, 64]. One policy, allowing to detect memory safety vulnerabilities, is memory coloring, which is implemented on top of tagged memory. The basic idea of memory coloring is to lock each memory allocation through a key. A later memory access is only permitted when using the correct key. This *lock-and-key* approach is implemented in the ARM Memory Tagging Extension (MTE) [2] and provides tagged memory in hardware. Google announced to work on deploying memory coloring based on MTE in Android on a larger scale [48], through the MemTagSanitizer [40] project integrated into LLVM [26]. While this concept is a step in the right direction, the memory overhead for storing the tags is still problematic for large-scale applications. To reduce the memory overhead of memory coloring, ARM decided to limit their concept to small tags. In ARM MTE, a 16 byte memory block is tagged with a 4 bits tag, resulting in a memory overhead of 3.125 %. While this memory overhead might be feasible for most applications, a tag size of 4 bits only leads to 16 individual colors, thus limiting the use of MTE as a security mechanism and making

debugging the main application possible. Increasing the tag size from 4 to 16 bits not only increases the available color space and, therefore, also the security guarantees, but also raises the memory overhead to 12.5 %. In addition to logical memory violations, systems deployed in hostile environments, such as cloud services or IoT devices, need to consider physical attacks on the system memory in their threat model. To maintain confidentiality and integrity of data stored in off-chip memory, memory encryption is a widely used technique. Although deploying transparent memory encryption is costly, vendors like Intel and AMD acknowledge the significant threat of physical attacks and offer schemes like Software Guard Extensions technology (SGX) [32], Multi-Key Total Memory Encryption (MKTME) [9], or Transparent Secure Memory Encryption (TSME) [22] for the consumer market. Memory encryption is also employed on smaller devices, e.g., in the Internet-of-Things (IoT), to protect sensitive data in memory [27, 56].

Despite the immense threat of logical and physical memory safety violations, the efficient combination of both mitigation strategies has not yet been extensively studied in past research and a naïve combination of both security strategies would accumulate both overheads.

Contribution

In our paper, we introduce CrypTag, a hardware/software co-design mitigating a broad range of logical memory safety issues and providing full physical memory safety. We demonstrate that realizing memory coloring on top of an already implemented memory encryption unit hardly costs more. In exploiting properties of the memory encryption scheme, we overcome limitations of traditional memory coloring schemes. While previous tagged memory architectures [1, 2, 21, 51, 52, 64] store the tag in memory and trade security against lower memory overhead, CrypTag completely avoids storing tags in memory and thus allows using larger tag sizes. CrypTag uses the memory color as additional input for the memory encryption scheme to encrypt every allocation differently. Inside the processor, we store the color information directly in the upper bits of the pointer, avoiding any additional storage overhead there. The tag is propagated through the system, stored in the cache, and finally used to encrypt the data when being stored in memory. Based on the capabilities of the underlying memory encryption engine, we derive two security policies for CrypTag.

We further present a software concept utilizing the hardware architecture to mitigate memory safety violations. Our approach assigns each allocated memory object on the heap, stack, and global data a random color. When accessing a memory object with the wrong color, e.g., due to a spatial or temporal memory bug, the CrypTag architecture, in its strongest security policy, identifies the color mismatch. This strategy allows us to successfully detect most spatial and temporal memory vulnerabilities, enhancing logical memory safety.

To evaluate our concept, we implemented an FPGA prototype based on the RISC-V CVA6 core. Furthermore, we extended the LLVM compiler to automatically instrument the code and protect all memory allocations without the need for user annotations. We evaluate the performance of CrypTag by executing different programs, from microbenchmarks to application code on our FPGA-based prototyping platform with Linux as host operating system. The evaluation shows that the performance penalty introduced by CrypTag is less than 6.1 % on a system already featuring a memory encryption engine. Summarized, our contributions are:

Memory Coloring Hardware Architecture: We efficiently combine memory encryption with memory coloring and show that the overhead for storing tags in memory can be entirely eliminated. This allows us to scale the tag size without additional memory cost.

Memory Safety Concept: We develop a hardware-assisted memory safety concept based on our memory coloring architecture. We demonstrate that the increased tag size of CrypTag achieves stronger security guarantees than comparable hardware-assisted memory safety designs, such as ARM MTE.

Prototype Implementation: We extend the RISC-V CVA6 core with a memory encryption engine and our memory coloring approach. We further provide a modified LLVM-based toolchain enforcing hardware-assisted memory safety by automatically instrumenting the application code. We show that the hardware overhead, for a system already using a memory encryption scheme, is less than 1 % and the software overhead is between 1.5 % and 6.1 %. While highly optimized commercial memory encryption systems typically induce an overhead between 5 % and 26 % [42], our evaluation of an open-source memory encryption unit shows a runtime overhead between 58.9 % and 109.8 %.

2 BACKGROUND

In this section, we discuss the backgrounds of memory safety, tagged memory, and memory encryption in general.

2.1 Memory Safety

According to Microsoft [34], 70 % of all security bugs fixed in Microsoft products are related to memory safety. Most of these bugs are critical because they could serve as an entry point for various other attacks. These attack techniques either tamper the control-flow or the data-flow of a program. To limit the impact of exploitable memory safety bugs, several attack mitigations like W⊕X or DEP are deployed in modern computer architectures. However, these countermeasures typically only raise the bar for a successful attack. Although simpler attacks, like the execution of attacker-injected code, can be mitigated, more advanced techniques, such as ROP, still can bypass these protection mechanisms [50]. Even more sophisticated countermeasures, like ensuring the integrity of the control-flow [30], can be defeated by techniques like DOP [20], where an attacker can craft turing-complete exploits. To successfully defeat memory vulnerabilities, memory safety is required [53]. Memory safety can be achieved by preventing all spatial and temporal memory vulnerabilities in the system. A spatial error is classified as dereferencing an out-of-bound pointer, such as accessing an array beyond its bounds, e.g., on the stack. A temporal error, in contrast, occurs when dereferencing a pointer to an already deallocated memory object [62]. In the past, memory safety concepts have already been presented. Watchdog [36], a hardware-based temporal memory protection scheme, assigns metadata to each allocated object and modifies this metadata on each memory deallocation. In comparing this metadata with the identifier stored in the pointer, potential temporal memory violations can be detected. Watchdog can also be extended to find spatial memory bugs. In addition, SoftBound [37] assures spatial memory safety by storing

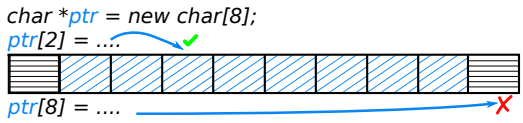

Figure 1: Memory coloring enforcing spatial memory safety.

the memory bounds of objects in a shadow memory. Because of the expensive monitoring of the object bounds by software checks, SoftBound adds an average runtime overhead of around 67 %. By combining SoftBound with CETS [38], temporal memory safety can be guaranteed, leading to full memory safety. However, the large performance overhead of 116 % on average makes the deployment hard on a larger scale.

2.2 Tagged Memory

The concept of tagged memory [8, 16, 31] is long-established and describes the idea of associating blocks of memory with additional metadata, *i.e.*, *tags*. Particularly, *TG*-bytes of memory are linked with a *TS*-bits wide tag, where *TG* denotes the tag granularity and *TS* the tag size. In these early computer architectures, tags were primarily used for debugging and for dynamically tracking the numeric type of data words. However, since tag bits are only memory, somewhat arbitrary policies can be implemented [10, 57]. Many recent designs utilize tags primarily for memory coloring, as shown in Figure 1. In such a coloring scheme, specific tag values, denoted as colors, are assigned to larger memory regions. When accessing the memory, these colors are used to determine if a particular read or write operation is genuine. A mismatch between the color of the accessed memory and the expected color results in a memory error. Memory coloring is used, e.g., for debugging [47], isolation [61], access control [51, 59], and for enforcing memory safety [49]. With ARM's new Armv8.5-A instruction set, the Memory Tagging Extension (MTE) [2] was announced, which embeds a tagged memory architecture into consumer hardware, such as mobile phones. A first attempt using the tagged memory approach on a larger scale is already integrated into the MemTagSanitizer [40] project of LLVM. Similar to the address sanitizer ASan [54] and the hardware-assisted address sanitizer HWASAN [55], Google's MemTagSanitizer intends to detect several spatial and temporal memory bugs. As the MemTagSanitizer benefits from hardware features, the high performance overhead of comparable software-based address sanitizer solutions is reduced to a minimum. Nevertheless, MTE requires the architecture to store the tags in memory. To avoid large memory overheads, MTE uses a small tag size of 4 bits, resulting in only 16 distinct memory colors. However, the security of the memory coloring scheme directly depends on the number of unique colors. Since colors are assigned randomly for each memory object, two adjacent objects can have the same color. For security critical systems, a detection probability of only 93.7 %, when having a tag size of 4 bits, is insufficient. Increasing the tag size from 4 to 16 bits would already result in a detection probability of 99.998 %, but also increases the memory overhead for tag storage from 3.125 % to 12.5 %.

While the Armv8.5-A architecture with the MTE feature has not yet been released in hardware, SPARC already implements a hardware-based memory tagging scheme with the Application Data Integrity (ADI) [1] feature embedded into Oracle's SPARC M7 processor. Similar to ARM MTE, the SPARC ADI feature also only supports a tag size of 4 bits.

2.3 Memory Encryption and Authentication

Memory safety does not protect the system from physical attacks, such as cold-boot attacks [19] or RowHammer [23]. To counteract these attacks, CPU vendors like Intel and AMD deployed memory encryption into their systems. Two strategies of transparent memory encryption/authentication are already widely used. The first and most common variant solely performs encryption to achieve confidentiality. Examples for memory encryption schemes are Intel's Total Memory Encryption (TME) [9] and AMD's Secure Memory Encryption (SME) [22]. The advantage of these schemes is the high performance and the lack of memory overhead. The second variant is based on Authenticated Encryption (AE) and provides both, confidentiality and authenticity, as does the encryption in Intel's Software Guard Extensions (SGX) [17]. Authenticated encryption is clearly the superior approach in terms of security since it protects against spoofing, splicing, and even enables to implement protection against replay attacks [14]. However, the increase in security typically comes at the cost of increased latency and memory overhead to store the integrity information.

Regarding granularity and key handling, different approaches have been proposed so far. Initial approaches relied on a single key for the whole encrypted memory (e.g., Intel's TME, AMD's SME). More recent designs, in contrast, also grant finer control over the used keys. AMD's Secure Encrypted Virtualization (SEV), e.g., supports the use of different keys for the virtualized guest machines. Intel's Multi-Key TME (MKTME) even supports different keys with page-wise granularity by embedding the key ID directly into the physical address. Our approach is perfectly compatible with all these design choices for key handling, but orthogonally extends them with support for tags in the virtual address space. The actual encryption key for each memory block is, subsequently, derived from the page or root key and the respective tag. As demonstrated in Section 5, even encryption with sub-cache line granularity can be supported in this way.

3 THREAT MODEL

Based on the CrypTag architecture, we propose a hardware-assisted memory safety concept. Similar to other threat models in the context of memory safety, we are considering an adversary using an exploitable memory bug to craft a memory vulnerability. Based on the capabilities of the underlying memory encryption engine (encryption only or with authentication), our memory coloring design CrypTag provides two different levels of security guarantees.

S1 Encryption & Authentication: When detecting a spatial memory safety violation, CrypTag immediately triggers a system exception via the inbuilt authentication mechanism of the transparent memory encryption scheme. Here, CrypTag is capable of detecting out-of-bound reads or writes, *i.e.*, a spatial memory bugs. Furthermore, CrypTag also is capable of reporting the exploitation of temporal memory bugs, e.g., use-after-free vulnerabilities.

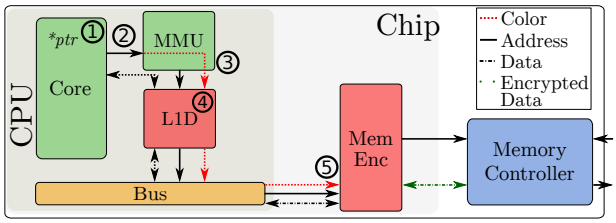

Figure 2: Overall CrypTag architecture. The memory encryption unit, placed between the memory subsystem and the memory controller, uses the color as a tweak.

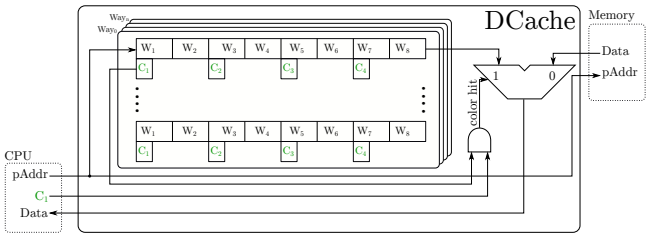

Figure 3: Set-associative cache architecture extended to support a color for each *TG*-bytes. On a cache hit, the data cache also checks the color. On a tag mismatch, the cache line is fetched from the memory.

S2 Encryption: Compared to **S1**, this security policy limits the exploitation of spatial and temporal memory bugs. For out-of-bound memory reads, CrypTag guarantees the confidentiality of the data stored in the target buffer. Since the underlying memory encryption engine does not provide data integrity, CrypTag cannot maintain the integrity of data in the target buffer in an out-of-bound memory write. However, CrypTag with **S2** aggravates the exploitation of temporal bugs and spatial out-of-bound writes.

4 DESIGN

In our architecture, memory is allocated in software ① and a dedicated instruction assigns a random color to the memory object and stores it in the upper bits of the pointer ②. When writing data to the memory, the color information is propagated through the MMU ③, the cache ④, and then finally is used as a tweak in the memory encryption unit ⑤. On a memory access, the hardware transparently performs a cryptographic check without any further instrumentation. This hardware architecture shown in Figure 2 allows CrypTag to protect dynamic, local, and global data.

4.1 Hardware

CrypTag utilizes a built-in memory encryption unit to implement an efficient memory coloring scheme. Initiated by a custom instruction, a random color is assigned for each memory object, which is stored in the upper unused bits of the pointer. When accessing the colored memory object, it requires the correct color to be in place for the memory request. CrypTag implements this *lock-and-key* approach by using the color of the memory object as an additional input for the transparent memory encryption scheme. Due to this strategy, each memory object colored with a random color is encrypted differently.

4.1.1 Memory Coloring. In CrypTag, the color of the memory object is assigned to the pointer. Since memory allocations are a frequent task and assigning and generating a color in software is costly, a custom instruction using a hardware-based random number generator is used. Similar to other designs [25, 41, 45], CrypTag uses the upper bits of the pointer to store the color information. This approach causes zero costs in terms of storing the color information and also minimizes any overhead to use pointers in software. Since the address information and the corresponding color are already stored in the same register, there is no need to extensively modify the instruction set. However, storing the color directly inside the pointer results in two disadvantages. First, the number of colors, which influences the security of memory coloring, directly corresponds to the number of free bits in the pointer. Second, using the upper bits of the pointer reduces the virtual address space of the system. Nevertheless, in practice, a trade-off between the available address space and security guarantees can be found. In most 64-bit platforms, already a reduced address space with free upper bits is used. For example, the AArch64 Linux port [29] limits, by default, the virtual address space to 39 bits and, therefore, supports colors up to 25 bits. While this address space might be sufficient for, e.g., mobile devices, a 512 GB address space is not acceptable for high-performance servers. By using the larger 48 bits addressing model, the address space can be extended to address 256 TB and supporting colors up to 16 bits. Security limitations of different color sizes are discussed in Section 7. When using a colored pointer, the color needs to be propagated throughout the system up to the memory encryption. Since the MMU of the processor only considers the lower bits of a pointer to translate the virtual to the physical address, CrypTag needs to bypass the MMU translation and directly forwards the color information to the cache (see Figure 2).

Cache Architecture. Figure 3 shows our extension to the cache architecture. In CrypTag, each TG-bytes of memory W are tagged with a TS-bits color C. For cache management, the color C is also stored in the cache for each memory object W. A cache hit is only valid if the color stored in the cache matches the color stored in the pointer. The design of CrypTag also supports sub-cache line tag granularities, e.g., one color for two words, which can be configured. In Section 5.1.3, we present a concrete cache implementation supporting the proposed color management.

4.1.2 Memory Encryption. When writing to memory, the color is used as a tweak to encrypt the data using the transparent memory encryption unit. To decrypt this data on a memory access, the read operation needs to have the correct color stored in the upper bits of the pointer. Depending on whether the memory encryption unit provides encryption and authentication or solely encryption, CrypTag either implements detection strategy **S1** or **S2**.

S1: Exception-based Notification. This strategy is possible for memory encryption engines (MEE) providing encryption and authentication. The exception-based notification policy immediately triggers an exception if the system performs a wrong memory access on a color mismatch. The encryption operation $C, T = Enc_{AE}(k, t, P)$ takes the encryption key k, the color as the cipher

tweak t, and the plaintext data P as an input to compute the ciphertext C and the authentication tag T as the output. Both the ciphertext and the authentication tag are stored inside the memory, while the color is *not*. When reading data back from memory, the MEE verifies the integrity of the ciphertext and decrypts the data, *i.e.*, $P \parallel \perp = Dec_{AE}(k, t, C, T)$. On a successful ciphertext verification using the authentication tag, the cipher returns the correct plaintext data P. If the integrity verification fails, e.g., owing to a wrong color, the MEE returns an error $\perp$ and the system automatically triggers an exception indicating an invalid memory access.

S2: Detection-based Notification. The detection-based notification policy corrupts the data when performing a wrong memory access. Here, the architecture uses a tweakable block cipher $C = Enc(k, t, P)$ without authentication. The ciphertext C is computed using the encryption key k, the color as the tweak t, and the plaintext data P. When reading data from memory, the MEE automatically decrypts the ciphertext under the encryption key and the tweak given by the memory color stored in the address. On a correct memory access, this also returns the correct plaintext data. However, using the wrong color on a malicious memory access decrypts the ciphertext with the wrong tweak leading to an invalid plaintext.

4.2 Software

The CrypTag hardware architecture alone does not thwart memory safety errors. It requires software support and protecting all memory allocations to detect most spatial and temporal memory vulnerabilities. The principle idea of the memory protection is that all memory allocations are colored, meaning that every associated pointer to a memory allocation stores a color in the upper bits of the pointer. Only when using the pointer with the correct color, the memory access is successful. Otherwise, depending on the detection strategy, either an error is raised or the payload data is destroyed. In this section, we describe how to use the hardare design in software to thwart memory safety vulnerabilities.

Heap Data. On each dynamic memory allocation, e.g., via a call to `malloc`, the returned pointer is assigned a dedicated random color. Furthermore, the memory is properly aligned to match the tag granularity TG. As discussed previously, our design uses a dedicated hardware instruction to perform this operation and, therefore, only adds a small overhead to manage the colors. When accessing heap data later on, every access encrypts or decrypts the data automatically using the assigned tag information. When releasing dynamic memory through a `free` operation, the color information of the pointer is removed and the memory is released to the OS.

Local Data. Local allocations on the stack are aligned to match the tag granularity TG. The corresponding pointer is colored using the custom instruction. Further accesses then encrypt or decrypt the data when accessing the memory.

Global Data. Protecting global data requires more effort. There are two possibilities to deal with global data. First, the protection of global data can be realized during compile time, where the compiler assigns each global variable a dedicated color. During the compilation, initialized global data is then encrypted using the pre-assigned color so that memory accesses in the program yield the correct data. However, this approach requires additional overhead to manage the colored pointers in software. Furthermore, access to global data always uses the same color for encryption, enabling e.g., replay attacks. To avoid the problem of replay attacks and unnecessary color management in software, we aim for a second approach. Our design replaces all references to global variables with a new pointer. Additionally, the compiler adds a dedicated startup hook function for each global variable. During the startup of the program, this hook function first colors the new pointer. Second, it reads the unencrypted global data from the executable and then writes this data to memory using the new colored pointer. Thereby, the global data automatically gets encrypted using the colored pointer. This approach allows us to use the same instruction to randomly color the new pointer. By using a random color in the pointer, we also mitigate replay attacks since the global data is encrypted differently at every program start. While this approach enhances security, it also simplifies the software support.

5 IMPLEMENTATION

In this section, we introduce the base platform where we integrate CrypTag and show the necessary hardware extensions. We discuss the color generation and propagation and further explain how the memory encryption framework is used to implement the coloring scheme. Finally, we introduce the compiler extension utilizing the CrypTag architecture to protect data.

Base Platform. We build the prototype for CrypTag on top of the CVA6 platform [63], a system-on-chip (SoC) using a 64-bit 6-stage RISC-V processor supporting to run Linux when mapped to a FPGA. The CVA6 is extended with the open-source memory encryption scheme MEMSEC [60]. MEMSEC is placed between the data cache and the DDR3 memory controller and automatically encrypts all data leaving the processor. Furthermore, we adapted the data cache and increased the cache line size from 16 to 64 byte.

5.1 Hardware Extensions

The necessary hardware extensions to implement CrypTag are minimal and only require two adaptions. First, the system requires a mechanism to create a color and to propagate it through the system. Second, the memory encryption engine (MEE) needs to be extended to handle the additional color input for the cipher.

5.1.1 Color Generation. Tagging a memory region with a dedicated memory color is initiated in software. Thus, we extend the RISC-V instruction set with a dedicated instruction to allow performing this operation efficiently in software.

`mstp rd,rs`. To color a pointer, the custom instruction `mstp` is added to the RISC-V instruction set. This instruction takes the value from the source register *rs* (typically the pointer), colors it, and stores the result to the destination register *rd*. Our architecture uses the SV39 addressing model [58] of RISC-V, where the lower 39 bits of the virtual address space are used. The remaining upper bits of the pointer are set to the sign bit of the pointer value (either all-zero or all-one). The `mstp` instruction colors the pointer and replaces the upper 25 bits (TS-bits) with a random color value. To differentiate between a colored and a non-colored pointer, the color bits cannot be set to all-zero or all-one. The random color value is

generated using a hardware-internal pseudo-random number generator, which is initialized during processor startup with a software inaccessible seed value.

5.1.2 Color Propagation. After instrumenting a pointer with the color bits, the MMU translates the virtual to a physical address. In SV39 of RISC-V, the MMU only uses the lower 39 bits of the address for its translation. The upper 25 bits containing the color information bypass the MMU's address translation. Both, the physical address and the color bits, get processed by the L1 data cache and are then propagated to the MEE via the processor's bus architecture.

5.1.3 Cache Design. As data in CrypTag is tagged with the color of the corresponding pointer, the cache also needs to be aware of these colors.

Colors. The prototype implementation of CrypTag uses a tag granularity (*TG*) of 16-bytes and a color size (*TS*) of 25 bits. As denoted in Figure 3, each 64-byte cache line stores four *TS*-bits colors. Internally, the cache differentiates between three values for a color: *no color*, *valid color*, and *invalid color*. When accessing the cache with an address where there is no color stored inside (the upper bits are all-zero or all-one), the cache-internal color is set to *no color*. A *invalid color* color is stored in the cache when prefetching a cache line with the wrong color triggers a decryption exception. When accessing the cache with an instrumented pointer, the *valid color* information is stored in the cache.

Cache Hit. A cache hit is triggered when having the correct data *and* correct color in the cache. If there is a color mismatch, a cache miss is triggered.

Cache Miss. On a cache miss, the cache architecture issues a memory read request to the main memory. Here, the colors of the cache line are used as a cipher tweak to decrypt data from the memory. After fetching the decrypted data from memory, the colors are stored in the metadata structure of the cache. In detection strategy **S1**, accessing the cache with a wrong color leads to a decryption and verification error and the error is forwarded to the processor as an exception.

Cache Prefetching. Cache prefetching is used to reduce the latency for memory accesses by precautionary fetching an entire cache line from memory. This technique speeds up memory accesses but also challenges our colored cache architecture. When performing a cache prefetch, only the color of the first *TG*-bytes of the cache line is known. To decrypt the remaining cache line, the system assumes the later colors of the cache line are the same and uses the first color to decrypt the whole cache line. Due to the memory fragmentation, this assumption is correct with high probability and the MEE decrypts the cache line correctly. Furthermore, the used color is copied to all color entries of the cache line. In case of a wrong decryption operation due to prefetching, detection strategy **S1** detects a wrong decryption and thus invalidates the color entry in the cache but does not raise an exception.

Cache Eviction. During cache eviction, a dirty cache line is written back to memory and is encrypted using the colors stored inside the cache. In case of having invalid colors in the cache, *i.e.*, due to prefetching, the cache only issues memory updates for entries with valid colors. Invalid cache entries are filtered and not written back to memory.

5.1.4 Memory Encryption Engine. MEMSEC, which is directly placed between the cache and the memory controller, transparently encrypts all data leaving the processor on bus level. Similar to other MEEs, the encryption key is randomly generated during the device startup. For the CrypTag architecture, we extended the MEE to support the additional color input. To implement detection strategy **S1**, we use the authenticated encryption cipher ASCON [11], which provides data confidentiality and integrity. We tweak the cipher by using the size-extended color value for the nonce input of the cipher. To re-initialize already encrypted memory, MEMSEC also allows to suppress authentication errors using a defined memory pattern. For implementing strategy **S2**, we use the low-latency tweakable block cipher QARMA [4], which is also used in ARM's pointer authentication scheme [41]. Since QARMA natively supports an additional input, we use the size-extended color value as input for the tweak.

5.2 Software Extensions

To detect memory safety violations and to protect every memory allocation, we need software support. We extended an LLVM-based C compiler [26] with an LLVM IR pass and a tiny runtime support library. The compiler needs to protect three storage classes: heap, local, and global data.

Protection of Heap Data. Protection of heap data is accomplished by using the GNU linker functionality to create wrappers around heap functions such as `malloc`, `free`, and `realloc`. For policy **S2**, the `malloc` wrapper aligns the size argument to *TG* and then calls `malloc` itself. Because heap allocation for 64-bit RISC-V systems is already 16 B aligned, we only have to apply the `mstp` instruction on the pointer before returning to the application. Hence, the overhead therefore is negligible. In Listing 1 we show the implementation of the wrapped `malloc` function for **S2** of the runtime library. When utilizing CrypTag for detection policy **S1**, `malloc` additionally initializes the memory with its color. Since this memory object could already be encrypted with a different color, naïvely recoloring would trigger an authentication error. Thus, CrypTag uses the nullification mechanism of MEMSEC to initialize the memory.

```
void* __wrap_malloc(size_t size) {
  size = roundup(size);
  void *ptr = __real_malloc(size);
  if (ptr == NULL) return NULL;
  return mstp(ptr);
}
```

Listing 1: S2 `malloc` wrapper tagging the returned pointer.

For `free`, the wrapper removes the color from the pointer argument and then calls the `free` function. This operation is done purely in software by using two shift operations. Notice that the heap administration data is stored in plain in between the encrypted heap data. Writing via a heap data-pointer out of bounds into the heap administration overwrites plain data with encrypted data making it hard for an attacker to modify the heap administration in a controlled way.

Protection of Local Data. Local variables on the stack are protected by scanning for `AllocaInst` instructions through a custom LLVM IR compiler pass. For each `AllocaInst`, we align the size argument to TG bytes, we align its address alignment to TG, and we insert an `mstp` instruction between the `AllocaInst` instructions and all its users. Additionally, for **S1**, we re-initialize the allocated memory with the assigned color. We perform a simple analysis on `AllocaInst` instructions to exclude protection of cases where incorrect usage will not be possible. For example, cases where the result of `AllocaInst` is not used by a `GetElementPtrInst` instruction with non-constant indices and the result is not stored in memory or leaves the function as argument of a function call.

Protection of Global Data. For each global definition/declaration of a variable named `foo`, we create a new global definition/declaration of a pointer called `__foo_mst` that points to `foo`. Furthermore, the LLVM IR pass replaces all references to `foo` with `__foo_mst` to get a colored pointer to `foo`. The runtime support library is informed about definitions `foo`, the size of `foo`, and the new pointer `__foo_mst` via a constructor function. During the startup, the constructor function in the runtime library will insert a color on `__foo_mst` by means of an `mstp` instruction. It will also encrypt `foo` using the color that has been put on `__foo_mst`. This happens by simply reading the data in plain using the original all-zero pointer and then writing it back to memory using `__foo_mst`. The runtime overhead of referencing a global variable is therefore one-load instruction. Furthermore, notice that on every execution of a protected application, its global variables will get different colors.

As with local data, we perform an analysis to exclude protection of (static) global data where we can prove that incorrect usage is not possible. Protecting global variables that have initializers with pointers to global variables complicates the protection. The runtime support library is informed via a constructor function about the positions of these pointers in global the initializers and patches those pointers with the color of the global variable it is pointing to.

Backward Compatibility. Application code that is protected by CrypTag can be combined with unprotected code. For example, the unprotected pointer results of `fopen()` or `mmap()` can be used in protected code without problems. Furthermore, protected pointers of heap, local, or global data can arbitrarily be passed to unprotected library functions without problems. The only compatibility issue that we are aware of is sharing global variables between protected and unprotected code, *i.e.*, `stdout`. Unprotected code will expect it unencrypted, while protected code will expect the data to be encrypted. Due to indirection via `__foo_mst` this issue will result in linking errors rather than runtime errors. The user should then manually place these global variables on a blacklist of global variables that are not suitable for protection.

Pointer Arithmetic. Incrementing a pointer to reach data within a colored object is natively supported in the CrypTag architecture because the color information in the upper bits of the pointer is not altered. However, subtracting or adding two pointers or performing shifts or multiplications on such pointers can modify the color and is therefore dangerous. To also support these operations and enable safe arithmetic operations avoiding integer overflows on colored pointers, dedicated instructions could be added. In the Armv8.5-A instruction set [3] supporting MTE, dedicated add and subtract instructions, ignoring the upper bits, are used for pointer arithmetic.

6 PERFORMANCE EVALUATION

To quantify the hardware and software overhead, we synthesize the CrypTag architecture for a Xilinx Kintex-7 series FPGA and run a recent Linux operating system on our platform. We report the performance and hardware overhead introduced by CrypTag for the TS=25 and TG=16 memory coloring configuration using different applications, from microbenchmarks to application code.

6.1 Hardware Overhead

In Table 1, we show the hardware overhead for the FPGA design in terms of lookup-tables (LUTs) and flip flops (FFs). The overhead numbers include the changes required for the new instruction, the color propagation, and the extended cache. Clearly, the hardware overhead of less than 1 % is very attractive and negligible in practice.

Cache Architecture. For CrypTag, the cache architecture is extended to also store colors along with the data. Furthermore, the decision logic to detect cache hits and misses is extended to also consider the colors in the cache. The hardware overhead for this comparison logic is relatively small compared to the overhead for storing the colors. The required hardware overhead for storing the colors in the cache is a function of the used color size TS and tag granularity TG. Note, our design only needs to store colors in the cache and there is no need for a separate, large cache for colors as it is required in other architectures [21, 52] to speed up accessing tags in memory. Thus, the design not only has less hardware overhead but also improves the runtime latency and bandwidth, as there are no memory accesses for colors needed. For an m-way n-set associative cache with a cache line size of C bytes, a tag granularity of TG bytes, and a color size of TS bits, we can compute the required number of color bits T as defined in Equation 1.

$$T = \text{nSets} \cdot \text{mWays} \cdot TS \cdot \frac{C}{TG} \tag{1}$$

Our modified 16 kB data cache of the CVA6 core is organized as a 4-way 64-set associative cache with a 64 byte cache line. For a tag granularity of TG=64 byte and a color size TS=8 bits, the overhead for storing the colors is 2.05 kbit. Although this is already fine-grained, our design even supports sub-cache line tag granularities, e.g., TG=16 B. For a configuration with TG=16 byte and TS=25 bits the overhead for storing the colors is 25.2 kbit. Table 2 shows the total number of color bits for different configurations including the corresponding overhead.

For our Xilinx-based FPGA, the cache is mapped to multiple block RAM instances. For a small memory color configuration (course tag granularity and small color size), the color bits even fit in the already instantiated block RAM resources of the cache and thus

Table 1: Hardware overhead for the TS=25 and TG=16 memory coloring configuration.

	LUTs		FFs	
Config.	Baseline [LUTs]	Overhead [%]	Baseline [FFs]	Overhead [%]
ASCON	57386	0.53	33885	0.14
QARMA	55804	0.67	32173	0.18

have no impact on the block RAM utilization. Only for the worst-case memory color configuration (TG=16 byte and TS=25 bits), an additional block RAM module needs to be instantiated. For other hardware technologies, e.g., ASIC designs, the cache overhead directly results from the color size and tag granularity. For, e.g., a 64 byte cache line and a memory coloring configuration of TG=16 byte and TS=25 bits, a cache line is extended by 100 bits for storing colors, resulting in an overhead for the cache of 19.5 %.

6.2 Runtime Overhead

To measure the software overhead of our system, we compiled different binaries with our custom LLVM-based toolchain protecting all dynamic memory, all locals, and all globals. We evaluate the performance overhead by running different benchmark applications in user mode on the Linux environment running on our FPGA hardware implementation. Note that the entire system, including the Linux operating system, is executed in the encrypted memory domain, but only user applications are additionally instrumented and use memory coloring. We use a set of benchmarks, including SPEC 2017 and smaller microbenchmarks, such as SciMark2 and MiBench. CrypTag, which we envision to be an extension of systems already featuring a memory encryption unit, adds an overhead between 1.5 % and 6.1 % on average for thwarting logical memory safety vulnerabilities on such a system. As we do not have access to state-of-the-art commercial memory encryption engines, which typically yield a performance overhead between 5 % and 26 % [42], we further evaluate the performance of the open-source MEMSEC framework. For transparently encrypting the whole external memory, we measured a performance overhead between 58.9 % and 109.8 % for SPEC 2017. The overall combined overhead for thwarting physical and logical memory safety vulnerabilities with CrypTag and MEMSEC is between 62.0 % and 116.1 %.

Code Size Overhead. As we are linking the 244 bytes runtime library while building the binary, the code size overhead to protect dynamic memory is constant and negligible compared to the overall binary size. Furthermore, to also protect local and global variables, `mstp` instructions are inserted as part of an instrumentation pass. Compiling the "SPECspeed 2017 Integer" testsuite with our toolchain, adds an average code size overhead of 1.02 %.

Runtime Overhead of CrypTag. Instrumenting pointers with the `mstp` instruction and aligning memory objects to the tag granularity increases the number of instructions for executing a program. Furthermore, prefetching data can possibly lead to invalid colors and, thus, requires additional memory accesses slowing down the execution. However, such cases only occur rarely and are not relevant in practice.

Table 2: Number of additional cache bits and total cache overhead for storing colors.

Cache Configuration	Color bits T [bit]	Cache Overhead [%]
TS=8, TG=64	2048	1.56
TS=8, TG=16	8192	6.25
TS=25, TG=64	6400	4.88
TS=25, TG=16	25600	19.53

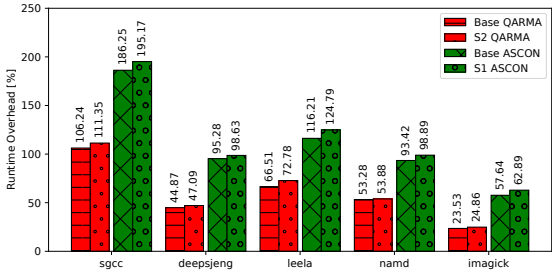

Figure 4: Runtime overhead for SPEC CPU 2017.

SPEC CPU 2017 To quantify the performance overhead introduced by our architecture, we first measured the performance overhead introduced by the memory encryption framework. Then, we analyzed the overhead additionally introduced by the memory coloring scheme. For our evaluation, we used a subset of C/C++ benchmark programs of the "SPECspeed 2017 Integer" and "SPECrate 2017 Floating Point" testsuites. However, due to the missing OpenMP support for RISC-V in LLVM, programs that depend on that are omitted. Furthermore, due to the constrained hardware resources of the FPGA prototype, several benchmarks already failed with an out-of-memory exception (`mcf`, `omnetpp`, `x264`, and `lbm`) and one with a runtime exception (`xalancbmk`). Note, these benchmarks already failed on the unmodified base platform with the native LLVM-compiler. Similar to other memory safety tools [12, 54], CrypTag also found several memory bugs in `perlbench`.

The performance overheads for the subset of "SPECspeed 2017 Integer" and "SPECrate 2017 Floating Point" benchmarks are depicted in Figure 4. This diagram shows that the dominating performance factor is the used memory encryption engine and *not* CrypTag. The relative overhead for thwarting logical memory safety vulnerabilities on a system already featuring memory encryption is 5.2 % for **S2** and 6.1 % for **S1** on average. For the unoptimized MEMSEC framework, we measured a performance overhead of 58.9 % for QARMA and 109.8 % for ASCON on average compared to the base platform. The overall performance overhead for the combined physical and logical memory safety protection averages to 62.0 % for **S2** and 116.1 % for **S1**.

SciMark2. To evaluate realistic computing workloads, we use SciMark2 [39], a benchmark for scientific and numerical computing. We both measure the benchmark performance using SciMark2's internal test score, which is shown in Figure 5, and the runtime overhead using the hardware prototype. Again, the relative overhead for logical memory safety protection is with 3.9 % for **S2** and 4.79 % for **S1** low. When comparing the CVA6 base platform to the system featuring MEMSEC as MEE, we measured an average runtime overhead of 55.45 % for QARMA and 69.09 % for ASCON. For the combined protection, we determined a performance overhead of 61.51 % for **S2** and 77.19 % for **S1** on average when compared to the system without the MEE.

MiBench. MiBench [18] is a free, representative, embedded benchmark suite and it is used by us to evaluate the runtime overhead of our design on application-level code. In Figure 6, we show the

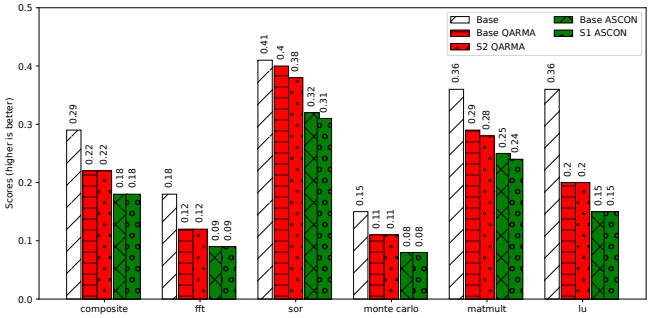

Figure 5: Benchmark score for SciMark2.

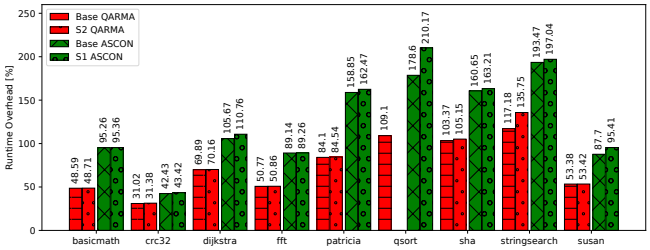

Figure 6: Runtime overhead for MiBench.

runtime overhead in terms of processor cycles relative to the baseline without memory encryption. The evaluation was performed using 10,000 test runs to average out scheduling effects from the operating system. Protecting all dynamic memory, all locals, and all globals with CrypTag introduces an average overhead of 1.5 % for **S2** and 4.9 % for **S1** on the system already featuring memory encryption. Our measurement shows that MEMSEC adds an average overhead of 74.2 % for QARMA and 123.5 % for ASCON compared to the baseline without MEE. For thwarting physical and logical memory safety vulnerabilities, we measured a combined overhead of 76.6 % for **S2** and 129.7 % for **S1** on average.

6.3 Prototype Limitations

As seen in Section 6.2, the main factor of the runtime overhead is the MEE and *not* CrypTag. Hence, our performance evaluation largely is affected by the performance of the underlying memory encryption unit. However, MEMSEC, the only, to the best of our knowledge, open-source MEE available, is not optimized for throughput and latency. Figure 7 depicts the significant impact of MEMSEC with ASCON and QARMA on the latency measured by the `lat_mem_rd 64M 512` benchmark of LMBench [33]. The memory throughput, measured with `bw_mem 4M rdwr`, also dropped from 52 MB/s to 14 MB/s for QARMA and 10.9 MB/s for ASCON. In comparison, state-of-the-art encryption engines typically yield a performance penalty between 5 % and 26 %, as reported by ARM [42]. Although optimizing MEMSEC or designing a high-speed MEE is not part of our contribution, we point out different optimization strategies in Section 9. Finally, we want to emphasize that we envision CrypTag to be an extension of systems already featuring a transparent memory encryption scheme. With major vendors, such as Intel with

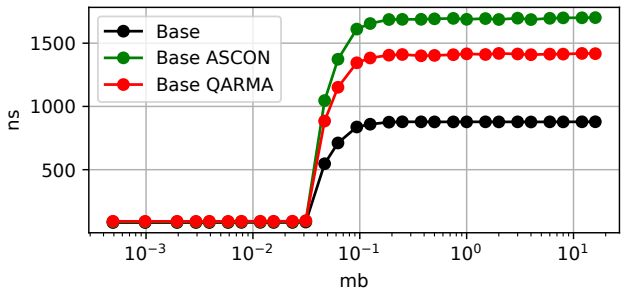

Figure 7: Memory latency measured with LMBench.

SGX, TME, and MKTME [43] and AMD with SME and TSME [22], highlighting the importance of memory encryption, we expect an increasing number of such systems in the near future. Here, CrypTag proposes an efficient solution to realize memory coloring on top of such systems with performance overheads between 1.5 % and 6.1 %.

7 SECURITY EVALUATION

CrypTag enhances security guarantees of applications by mitigating the exploitation of most temporal or spatial memory bugs. Based on the underlying MEE, *i.e.*, encryption and authentication or encryption only, CrypTag either enforces security policy **S1** or **S2**.

Spatial Memory Safety in S1. Spatial memory bugs allow an adversary to access data outside of the objects bound. To detect these bugs, CrypTag utilizes the architecture to color the pointer and to initialize the memory object with a random color on a memory allocation. Any subsequent access to this colored object requires that the access pointer is colored with the identical color, or an authentication error is triggered by the MEE. Hence, out-of-bound read or write accesses to memory objects with a wrongly colored pointer are detected by CrypTag in **S1**. Similar to other tagged memory schemes, CrypTag cannot detect intra-object overflows. Since the vulnerable buffer, as well as the target, are stored in the same memory object, both objects have the same color.

Spatial Memory Safety in S2. Compared to **S1**, this security policy limits the exploitation of spatial memory bugs. Usually, the attacker either uses spatial memory vulnerabilities to leak sensitive data or to modify control or non-control related data to craft ROP, DOP, or other attacks. In out-of-bound read accesses, data encrypted with the original color is decrypted using the wrong color of this pointer. Hence, CrypTag with **S2** maintains the confidentiality of data in spatial out-of-bound reads and provides protection from attacks such as Heartbleed [13]. Since the underlying MEE does not provide data integrity, CrypTag cannot prevent an attacker from overwriting data in a target buffer using an out-of-bound write. However, when reading this data using the corresponding pointer, pseudorandom values are retrieved. Using this pseudorandom value as control-flow related data, e.g., as a return address, most likely will cause an exception. As the attacker cannot overwrite data in the target buffer in a controlled way, CrypTag raises the complexity for performing data-oriented attacks, such as DOP.

Temporal Memory Safety in S1. In a temporal memory safety violation, a memory object is accessed after it is deallocated. Temporal memory safety violations are mostly exploited by use-after-free vulnerabilities, which CrypTag with **S1** can detect. In this attack, a memory object gets deallocated and the space, later on, is used by a new object. The attacker then can use the dangling pointer either to leak sensitive data or to tamper data, e.g., a vtable pointer. A similar concept is used by a double-free attack, where the adversary calls the memory deallocation functionality twice. CrypTag with **S1** mitigates such attacks by assigning a new color on each memory allocation and initialization, reading or writing by using the dangling pointer colored with the previous color will trigger an exception. Since the current implementation of CrypTag does not re-color the memory object on deallocation, a memory read or write to this memory region using the dangling pointer cannot be detected by CrypTag. However, as soon as a new memory object is allocated and initialized on this region, it is tagged with a new color and accesses using the dangling pointer can be detected. To prevent this behavior, CrypTag could be, similar to ARM MTE, extended to colorize memory objects with a new color on each deallocation.

Temporal Memory Safety in S2. Although CrypTag with **S2** cannot detect temporal memory bugs, it prevents the adversary from leaking data, *i.e.*, CrypTag maintains the confidentiality of data. When using this vulnerability to overwrite sensitive data, such as vtable entries, the attacker cannot insert targeted data because the wrong color of the dangling pointer for the encryption is used.

Null Pointers. Pointers created in external libraries, which are not compiled with CrypTag, are not colored and thus have the all-null color. Unlike CHERI [61], where only the pointer is tagged with additional metadata and not the memory itself, CrypTag explicitly tags memory objects with its color. Hence, a read or write vulnerability on a null-colored object only allows the attacker to access other null-colored memory objects and not the whole memory. Colored data that is allocated by protected code can be passed to unprotected code, e.g., external libraries, and is also protected there.

Stack Coloring. By coloring the stack pointer with `mstp` on program initialization, all objects on the stack, which are not explicitly colored, *i.e.*, stack spills, are assigned a random color. This strategy allows CrypTag to separate the stack from null-colored objects, e.g., objects created in unprotected code.

Entropy. Similar to countermeasures like ASLR, PARTS [28], and MTE [2], CrypTag is a probabilistic mitigation technique. A memory safety violation, such as a linear or non-linear buffer overflow, cannot be detected in **S1** by the CrypTag architecture if the color of the target memory object matches the color of the exploited memory object. However, since CrypTag already detects a memory safety violation at the first mismatch and the attacker cannot influence the color assignment of a memory object, the attacker requires a color collision at the first try. The probability of having a color collision of two memory objects directly corresponds to the number of used color bits. A memory safety vulnerability, such as Heartbleed, can be detected with **S1** at the first violation with a probability of 93.7 % for a tag size of 4 bits, with a probability of 99.998 % for a tag size of 16 bits, and for a tag size of 25 bits even with a higher probability. Additionally, since attacks like ROP or JOP require the adversary to build an attack chain, multiple color collisions are required increasing the detection probability. Although larger color sizes also increase the security guarantees, schemes like ARM MTE do not utilize the full available space in the free upper bits of the pointer because storing the color in memory is required, resulting in significant memory overheads. CrypTag prevents this security-overhead trade-off by completely avoiding storing the color in the memory, allowing the scheme to fully utilize the unused bits in the pointer and maximize security guarantees.

Tag Granularity. Due to its nature, memory coloring is an imprecise protection mechanism. For example, when allocating a 30 B memory object in CrypTag with a tag granularity of 16 byte, the full 32 byte are colored with the same color. When accessing byte 31 using a linear buffer overflow, the memory safety violation cannot be detected by any memory coloring scheme. However, in practice, this issue can be circumvented by choosing an appropriate tag granularity. On 64-bit RISC-V systems, objects on the heap are 16 byte aligned. Here, by choosing a tag granularity of also 16 byte, the adjacent target memory object is tagged with a different color and cannot be reached by the attacker in **S1**. Objects on the stack are also aligned to TG and the size is increased to a multiple of TG. Now, the victim and target buffer, e.g., a return address, are in different color domains allowing CrypTag in **S1** to detect a memory safety violation. When the memory object size would not have been resized to a multiple of TG and the tag granularity would be larger than the memory alignment, e.g., $TG = 64$, the same color is assigned to two, e.g., 32 B, adjacent memory objects making it impossible to detect an overflow. Although a smaller TG allows a more fine granular detection mechanism, it also increases the overhead for storing the colors in the cache architecture. Similar to other research [46], we suggest to use a tag granularity of 16 byte on our reference platform.

Color Checking. Schemes like PARTS, which uses ARM's pointer authentication feature [41], or CCFI [30] use dedicated authentication instructions to verify the integrity of the pointer. Since verification and usage is, except for dedicated instructions like the `blraa` instruction in ARM, not atomic, these countermeasures are vulnerable against time-of-check to time-of-use (TOCTOU) attacks. CrypTag in **S1** circumvents this problem by enforcing a color check automatically in hardware for each memory access.

Color Management. Coloring a memory object with a color either can be done using a randomized or a deterministic color assignment strategy. When using a deterministic coloring scheme, a color management mechanism needs to track the color assignment to assure that two adjacent memory objects have a different color. An example of a system deterministically assigning tags is ARM's pointer authentication scheme, which is integrated into Apple smartphones. However, past research [5] showed that an attacker can forge arbitrary signed pointers by using signing gadgets. To prevent color management security issues and avoid additional overhead introduced by the mechanism, CrypTag uses a randomized coloring approach.

8 RELATED WORK

This section summarizes different memory vulnerability schemes and analyzes their performance overhead and security guarantees.

8.1 Overhead Comparison

As shown in Table 3, the performance overhead of less than 6.1 % for CrypTag on a system already featuring a memory encryption unit is low. These numbers show that extending a system with an already integrated memory encryption scheme with CrypTag is reasonable, as memory safety can be implemented relatively cheaply. We argue that with the increasing amount of systems providing memory encryption, such as Intel SGX or AMD TSME, also the number of platforms potentially supporting CrypTag increases. While there is already a trend of using memory encryption in cloud computing, the announcement of Intel [44] introducing memory encryption for commodity processors soon highlights the importance of it for wider deployment. Typically, highly optimized memory encryption units can be implemented with an overhead between 5 % and 26 % [42]. As we do not have access to these commercial MEEs for our prototype, we used the open-source MEMSEC framework, where we measured an average performance overhead between 58.9 % and 109.8 %. For the overall performance overhead of the combined physical and logical memory safety protection, we measured an overhead between 62.0 % and 116.1 %. These numbers show that the dominating performance factor is MEMSEC and *not* CrypTag. Furthermore, we contend that a naïve combination of logical and physical memory safety, such as combining PARTS with memory encryption, accumulates both overheads.

8.2 Security Comparison

In general, logical memory safety strategies can be categorized into schemes that are limiting the attacker's capabilities when exploiting a memory bug (e.g., PARTS, CPI, CCFI) and those detecting the exploitation of a memory safety bug. CrypTag with **S1** uses the latter approach to thwart logical memory safety attacks by detecting a broad range of spatial and temporal memory bugs.

Control-flow integrity. Control-flow integrity (CFI) minimizes the attacker's capability when exploiting a memory bug by limiting the control-flow of a program to only valid paths through the control-flow graph (CFG) [6]. The security of CFI schemes depends on the precision of the CFG, which is typically determined using static analysis, and the reliability of the security enforcement. Cryptographic CFI (CCFI) [30] improves the precision of commodity CFI schemes by dynamically performing pointer classification at runtime. Similar to CrypTag, CCFI utilizes cryptography to enforce runtime security. Each object that influences the control-flow of a program is tagged with the MAC over the pointer and its dynamically determined class. The MAC is then checked before using the object. Since computing and verifying a MAC is costly, CCFI increases the overhead by 52 %. Due to the nature of CFI schemes, CCFI cannot provide spatial and temporal memory safety.

Code-pointer integrity. Similar to CFI, code-pointer integrity (CPI) [24] claims to prevent all control-flow hijack attacks, while simultaneously decreasing the performance overhead. CPI protects sensitive code-pointers by storing them and metadata in a safe region. While the overhead introduced by CPI is negligible, its security completely relies on the isolation of this region. On systems without segmentation protection support, like for x86-64 systems, CPI uses information hiding to protect its safe region making it vulnerable to attacks leaking this location [15].

Code- and data-pointer Integrity. Advanced attack scenarios, like ROP or DOP, show that providing data- or control-flow integrity exclusively is not sufficient. It requires a combination of defense strategies to mitigate against a powerful attacker. One promising attempt utilizing hardware features offered by the underlying architecture is PARTS [28]. PARTS implements a compiler instrumentation, which automatically adds pointer integrity checks to protect all code- and data-pointers. Here, dedicated pointer authentication instructions are used to perform pointer signing and verification. PARTS protects all backward-edge and forward-edge code-pointers, as well as all data-pointers. However, the data-pointer integrity scheme does not provide temporal or spatial memory safety. Therefore, PARTS is vulnerable against attacks targeting the data plane, like Heartbleed [13], or other security-critical attacks on non-control data [7].

Memory Safety. Memory safety prevents the exploitation of memory bugs. As such, it is considered to be a stronger concept than mitigating the effects of an exploited memory bug [53]. However, software-based solutions, like the combination of SoftBound and CETS, typically yield a high performance penalty, making these schemes unrealistic to deploy on a larger scale. To reduce the overhead, hardware support is required. One promising hardware-assisted attempt to detect most spatial and temporal bugs is based on ARM's MTE feature is Google's MemTagSanitizer. However, at the time of writing, MemTagSanitizer is still under development and no performance numbers are released yet and no protection of the heap is implemented. Although we expect that this scheme will provide similar performance than CrypTag, MTE only provides restricted security guarantees. MTE uses a small tag size to limit the memory overhead introduced by storing the tags in memory.

Table 3: Security comparison of different memory vulnerability mitigation schemes.

Scheme	Code-Pointer Integrity	Data-Pointer Integrity	Temporal Safety	Spatial Safety	Overhead
CCFI	✔	✖	✖	✖	52 %
CPI	✔	✔	✖	✖	8.4 %
PARTS	✔	✔	✖	✖	19.5 %
SoftBound+CETS	▲	▲	✔	✔	116 %
MemTagSanitizer	◐	◐	✖	✔	-
CrypTag	▲	▲	◐	✔	6.1 %* 109.8 %†

✔ Full ◐ Partial ▲ Indirect ✖ No Protection

* Including Memory Encryption Overhead

†Additional to the Memory Encryption Overhead

9 CONCLUSION & FUTURE WORK

Current memory security schemes either are incomplete [28, 30], do not provide enough security [2, 24], or add non-negligible overhead [37, 38], especially when combined with physical memory safety, to the system. CrypTag closes these gaps by introducing a memory safety concept based on a hardware-assisted memory coloring scheme. CrypTag combines memory encryption with memory coloring to thwart a broad range of physical and logical memory

safety vulnerabilities. By combining these two mechanisms, we show that memory coloring almost comes for free and memory safety vulnerabilities can efficiently be protected. The design uses a color, stored inside the related pointer, and propagates this value up to the data cache of the system. This color value is used to tweak the memory encryption system, thus avoiding storing the color in memory. Our approach shows that the performance overhead for CrypTag is negligible and, therefore, can be used for large scale deployment. In this paper, we provide an end-to-end solution from the concept to the prototype implementation of our design. We integrated CrypTag to a RISC-V based processing platform and adapted an LLVM toolchain and developed a runtime library to automatically instrument programs and protect all memory allocations of the application without the need for user annotations. Our evaluation shows that the hardware overhead for these changes is less than 1 % and the software overhead compared to a system already featuring a memory encryption unit is on average less than 6.1 %, which makes this design practical for real-life applications.

Future Work. As mentioned in Section 6.3, the performance of CrypTag largely depends on the performance of the memory encryption engine (MEE). Hence, a possible future work would be to optimize the performance of MEMSEC. Currently, MEMSEC operates at the same clock frequency as the processor core. To increase the memory bandwidth and further decrease the latency of memory accesses, MEMSEC could be placed next to the memory to operate on a much higher clock frequency. However, this requires to optimize the inner logic of MEMSEC to avoid any timing violations. Currently, MEMSEC is a highly flexible framework allowing several corner cases, such as AXI bursts and strobes. Here, one strategy to maximize the performance of the MEE could be to adapt MEMSEC to the target architecture and remove functionalities not supported by this architecture. If providing physical memory safety is not needed, a final optimization step could be to only encrypt colored memory objects and bypass the MEE for non-colored objects.

ACKNOWLEDGMENTS

We thank Samuel Weiser, David Schrammel, and the anonymous reviewers for helping to improve this paper. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402) and by the Austrian Research Promotion Agency (FFG) via the competence center Know-Center (grant number 844595), which is funded in the context of COMET - Competence Centers for Excellent Technologies by BMVIT, BMWFW, and Styria.

REFERENCES

[1] Kathirgamar Aingaran, Sumti Jairath, Georgios K. Konstadinidis, Serena Leung, Paul Loewenstein, Curtis McAllister, Stephen Phillips, Zoran Radovic, Ram Sivaramakrishnan, David Smentek, and Thomas Wicki. 2015. M7: Oracle's Next-Generation Sparc Processor. *IEEE Micro* 35 (2015).
[2] ARM. 2019. *Armv8.5-A Memory Tagging Extension.*
[3] ARM. 2020. *Arm Architecture Reference Manual.*
[4] Roberto Avanzi. 2016. The QARMA Block Cipher Family - Almost MDS Matrices Over Rings With Zero Divisors, Nearly Symmetric Even-Mansour Constructions With Non-Involutory Central Rounds, and Search Heuristics for Low-Latency S-Boxes. *ePrint 2016/444* (2016).
[5] Brandon Azad. 2019. *Examining Pointer Authentication on the iPhone XS.*
[6] Nicholas Carlini, Antonio Barresi, Mathias Payer, David A. Wagner, and Thomas R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *USENIX Security Symposium.*
[7] Shuo Chen, Jun Xu, and Emre Can Sezer. 2005. Non-Control-Data Attacks Are Realistic Threats. In *USENIX Security Symposium.*
[8] Brian E. Clark and Michael J. Corrigan. 1989. Application System/400 Performance Characteristics. *IBM Syst. J.* 28 (1989).
[9] Intel Corporation. 2019. *Intel® Architecture Memory Encryption Technologies Specification.* Technical Report.
[10] Udit Dhawan, Nikos Vasilakis, Raphael Rubin, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight Jr., Benjamin C. Pierce, and André DeHon. 2014. PUMP: a programmable unit for metadata processing. In *International Symposium on Computer Architecture – ISCA.*
[11] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. 2016. *Ascon v1.2 Submission to the CAESAR Competition.* Technical Report.
[12] Gregory J. Duck and Roland H. C. Yap. 2016. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016.*
[13] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. 2014. The Matter of Heartbleed. In *Internet Measurement Conference – IMC.*
[14] Reouven Elbaz, David Champagne, Catherine H. Gebotys, Ruby B. Lee, Nachiketh R. Potlapally, and Lionel Torres. 2009. Hardware Mechanisms for Memory Authentication: A Survey of Existing Techniques and Engines. *Trans. Comput. Sci.* 4 (2009).
[15] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard E. Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. 2015. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *IEEE Symposium on Security and Privacy – S&P.*
[16] Edward A. Feustel. 1972. The Rice research computer: a tagged architecture. In *American Federation of Information Processing Societies – AFIPS.*
[17] Shay Gueron. 2016. A Memory Encryption Engine Suitable for General Purpose Processors. *ePrint 2016/204* (2016).
[18] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538).*
[19] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. 2008. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *USENIX Security Symposium.*
[20] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *IEEE Symposium on Security and Privacy – S&P.*
[21] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W. Moore, Alex Bradbury, Hongyan Xia, Robert N. M. Watson, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alfredo Mazzinghi, Alex Richardson, Stacey D. Son, and A. Theodore Markettos. 2017. Efficient Tagged Memory. In *International Conference on Computer Design – ICCD.*
[22] David Kaplan, Jeremy Powell, and Woller Tom. 2016. AMD MEMORY ENCRYPTION. (2016).
[23] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *International Symposium on Computer Architecture – ISCA.*
[24] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2018. Code-pointer integrity. In *The Continuing Arms Race: Code-Reuse Attacks and Defenses.*
[25] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight Jr., and André DeHon. 2013. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Conference on Computer and Communications Security – CCS.*
[26] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization – CGO.*
[27] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Dawn Song, and Krste Asanovic. 2019. Keystone: A Framework for Architecting TEEs. *arXiv abs/1907.10119* (2019).
[28] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. 2019. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *USENIX Security Symposium.*
[29] Catalin Marinas. 2020. *Memory Layout on AArch64 Linux.*
[30] Ali José Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: Cryptographically Enforced Control Flow Integrity. In *Conference on Computer and Communications Security – CCS.*
[31] Alastair J. W. Mayer. 1982. The Architecture of the Burroughs B5000: 20 Years Later and Still Ahead of the Times? *SIGARCH Comput. Archit. News* 10 (1982).
[32] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. 2016. Intel® Software Guard Extensions

(Intel® SGX) Support for Dynamic Memory Management Inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. Association for Computing Machinery.
[33] Larry W. McVoy and Carl Staelin. 1996. lmbench: Portable Tools for Performance Analysis. In *USENIX Annual Technical Conference*.
[34] Matt Miller. 2019. Trends, Challanges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape. *BlueHat IL* (2019).
[35] MITRE. 2019. *CWE Top 25 Most Dangerous Software Errors*.
[36] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *International Symposium on Computer Architecture – ISCA*.
[37] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for c. In *Programming Language Design and Implementation – PLDI*.
[38] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *International Symposium on Memory Management – ISMM*.
[39] Roldan Pozo and Bruce Miller. [n.d.]. *Scimark 2*.
[40] LLVM Project. 2020. *MemTagSanitizer*.
[41] Inc. Qualcomm Technologies. 2017. *Pointer Authentication on ARMv8.3*.
[42] Avanzi Roberto-Maria. 2020. *Memory Protection for the ARM Architecture*.
[43] JIM SALTER. 2020. *Intel promises Full Memory Encryption in upcoming CPUs*.
[44] Jim Salter. 2020. *Intel promises Full Memory Encryption in upcoming CPUs*.
[45] Robert Schilling, Mario Werner, Pascal Nasahl, and Stefan Mangard. 2018. Pointing in the Right Direction - Securing Memory Accesses in a Faulty World. In *Annual Computer Security Applications Conference – ACSAC*.
[46] Kostya Serebryany. 2019. ARM Memory Tagging Extension and How It Improves C/C++ Memory Safety. *login Usenix Mag.* 44 (2019).
[47] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference*.
[48] Kostya Serebryany and Herle, Sudhi. 2019. *Adopting the Arm Memory Tagging Extension in Android*.
[49] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. 2018. Memory Tagging and how it improves C/C++ memory safety. *arXiv abs/1802.09517* (2018).
[50] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Conference on Computer and Communications Security – CCS*.
[51] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. 2016. HDFI: Hardware-Assisted Data-Flow Isolation. In *IEEE Symposium on Security and Privacy – S&P*.
[52] Wei Song, Alex Bradbury, and Robert Mullins. 2015. Towards general purpose tagged memory. In *Proceedings of the RISC-V Workshop*.
[53] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy – S&P*.
[54] The Clang Team. 2020. *AddressSanitizer*.
[55] The Clang Team. 2020. *Hardware-assisted AddressSanitizer Design Documentation*.
[56] Thomas Unterluggauer, Mario Werner, and Stefan Mangard. 2019. MEAS: memory encryption and authentication secure against side-channel attacks. *J. Cryptographic Engineering* 9 (2019).
[57] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. 2008. FlexiTaint: A programmable accelerator for dynamic taint propagation. In *International Conference on High-Performance Computer Architecture – HPCA*.
[58] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A. Patterson, and Krste Asanović. 2016. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.9.1*. Technical Report. EECS Department, University of California, Berkeley.
[59] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. 2019. TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V. In *Network and Distributed System Security Symposium – NDSS*.
[60] Mario Werner, Thomas Unterluggauer, Robert Schilling, David Schaffenrath, and Stefan Mangard. 2017. Transparent memory encryption and authentication. In *Field Programmable Logic and Applications – FPL*.
[61] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert M. Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *International Symposium on Computer Architecture – ISCA*.
[62] Wei Xu, Daniel C DuVarney, and R Sekar. 2004. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*.
[63] Florian Zaruba and Luca Benini. 2019. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Trans. Very Large Scale Integr. Syst.* 27 (2019).
[64] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. 2008. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *USENIX Symposium on Operating Systems Design and Implementation – OSDI*.