HeMate: Enhancing Heap Security through Isolating Primitive Types with Arm Memory Tagging Extension

Yu-Chang Chen
National Taiwan University
Taiwan
r10944012@csie.ntu.edu.tw

Shih-Wei Li
National Taiwan University
Taiwan
shihwei@csie.ntu.edu.tw

ABSTRACT

Memory safety vulnerabilities are a significant challenge for programming languages like C and C++. Among these vulnerabilities, heap-based issues have become more prevalent in recent years. Exploiting these vulnerabilities allows adversaries to execute arbitrary memory reads, writes, and even code execution. The Memory Tagging Extension (MTE), introduced in the Arm v8.5-A processor architecture, is an example of such a security feature. MTE has been utilized in modern software to implement probabilistic protection for heap-based memory safety vulnerabilities, including use-after-free and heap-based buffer overflow. However, the existing MTE-based approaches offer probabilistic protection and are vulnerable to brute-force attacks. Moreover, these approaches offer inter-object isolation but are vulnerable to intra-object overflow. Further, adversaries leverage memory confusion to manipulate or leak pointers, leading to arbitrary memory read/write and code execution. In response to the limitation and security, this work introduces a novel usage of MTE, called HeMate, to isolate memory storing different primitive types of data on the heap to enhance memory safety. This approach provides a non-probabilistic constraint on vulnerability exploitation against memory objects with different primitive data types, such as intra-object overflow and use-after-free. We have implemented a HeMate prototype compiler for C programs based on the LLVM framework. Our approach effectively leverages MTE to protect against memory safety vulnerabilities while preserving the functionality of commonly used Linux applications.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

Software Security, Arm MTE

ACM Reference Format:
Yu-Chang Chen and Shih-Wei Li. 2024. HeMate: Enhancing Heap Security through Isolating Primitive Types with Arm Memory Tagging Extension. In *The 19th International Conference on Availability, Reliability and Security (ARES 2024), July 30-August 2, 2024, Vienna, Austria.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3664476.3664492

ARES 2024, July 30-August 2, 2024, Vienna, Austria

ACM ISBN 979-8-4007-1718-5/24/07.
https://doi.org/10.1145/3664476.3664492

1 INTRODUCTION

Memory safety has been a challenge for programs written in programming languages, such as C and C++, for a long time. According to the latest MITRE ranking [6], which assesses the most prevalent software vulnerabilities, buffer overflow and use-after-free (UAF) remain at the top. Additionally, statistics [7, 23] indicate that approximately 70% of the vulnerabilities in some of their products relate to memory safety issues. Although this percentage may vary between different programs, it highlights the continued prevalence of memory safety problems in software. The statistics reveal the types of vulnerabilities that are commonly observed. For instance, Microsoft's statistics indicate that the most frequently occurring vulnerability class is a heap-based buffer overflow, followed by use-after-free vulnerabilities. Similarly, Google's statistics show that over 50% of memory safety issues are use-after-free vulnerabilities. It is worth noting that both UAF and buffer overflow vulnerabilities are related to heap memory usage. These statistics highlight the significance of enhancing heap-based memory safety.

Software-based approaches have been proposed to enhance memory safety. AddressSanitizer [25] detects illegal memory accesses by tracking the memory state during allocation and deallocation and checking it during access. Although it effectively detects memory errors, it incurs significant performance overhead and is used as a memory error detection tool rather than a runtime mitigation technique for production applications. On the other hand, various schemes were introduced [3, 15, 20] to mitigate UAF vulnerabilities. For instance, Apple incorporated kalloc_type [3] for the zone memory allocator in iOS to isolate the allocation of memory object types. Objects with different types are allocated from different zones, preventing confusion between two types via UAF. However, these schemes require developers' efforts to manually adopt the scheme to allocate specific data types [3, 15] and specialized memory allocators that support the scheme.

Hardware-based mitigation mechanisms have been proposed in recent years to bolster memory safety. New processor security features, including Arm's Memory Tagging Extension (MTE) [19] allows software [4, 18] to use specific instructions to leverage hardware mechanism to enforce memory isolation/compartments, branch target validation, and pointer integrity validation. MTE was introduced among these security features in Arm v8.5. MTE provides instructions to assign a tag, represented as a small integer value, to a range of memory. Consequently, during subsequent program execution, the program must use a pointer carrying the corresponding tag to access the tagged memory. MTE detects tag

mismatches and aborts unsafe memory accesses. MTE supports 16 unique tags. Modern software utilizes MTE to offer probabilistic protection against heap-based memory safety issues. Memory allocators tag an allocated memory chunk with a randomly generated tag. The probabilistic nature renders the mechanism vulnerable to brute force attacks. Attackers who obtain a pointer with a tag that matches the allocated chunk can bypass such MTE-based protection. Furthermore, since a single tag is assigned to an allocated chunk, the current scheme cannot protect against intra-object buffer overflow [24]. An attacker can perform out-of-bounds memory writes to members (e.g., of a C structure) within a chunk via a pointer for the same chunk.

In this paper, we propose *HeMate*, a new usage for MTE to implement runtime mitigation for C language programs, specifically targeting common heap-based memory safety issues such as use-after-free and heap-based buffer overflow. Compared to the previous works [8–10], HeMate provides non-probabilistic protection against heap safety vulnerabilities for instances of data with different primitive types. This new approach extends protection to unmitigated weaknesses by existing software usage of MTE, such as intra-object overflows and brute force attacks. We have implemented a prototype compiler for HeMate using the LLVM compiler framework while retaining the support of memory allocators from the standard libraries. The HeMate LLVM compiler instruments C programs to enhance memory safety. Our prototype supports Linux/Ubuntu on Arm systems with the MTE feature. We demonstrated that HeMate preserves the functionality of Ubuntu's core utilities and commonly used applications, such as x264 while enhancing their security with MTE.

2 BACKGROUND

2.1 Memory Tagging Extension

MTE [19] is a hardware-assisted mechanism introduced in Arm v8.5-A. It provides a hardware primitive that detects spatial and temporal memory safety issues through memory tagging, also known as memory coloring.

The concept behind memory tagging shares similarities with the lock-and-key mechanism. In the current Arm architecture, specifically Arm v8.5-A, a tag is a 4-bit integer value residing in bits 59-56 of a pointer (a 64-bit value), which serves as the key. MTE supports 16 unique tags. During program execution, the program can assign a tag to every 16-byte memory region, resembling setting a lock on that region. This tag value is stored in physical memory, serving as a lock. The physical memory can only be modified using memory tagging instructions. Consequently, when accessing the tagged memory, the program must employ a pointer that carries the corresponding tag (i.e., key) associated with the target tagged memory (i.e., lock). This requirement ensures the validity of the pointer during memory access operations, requiring each memory access operation to possess a matching key lock. Failure to adhere to this rule triggers a CPU exception, indicating a violation of the memory tagging.

The software can leverage this mechanism to design strategies to modify the lock in different situations and detect spatial or temporal memory safety issues. Besides detecting memory safety issues, MTE is also a hardware primitive to support software compartmentalization [22].

Programs can use a set of instructions to utilize MTE. Three instructions are provided for distinct purposes: generating a tag for a pointer, assigning a tag to memory, and loading a tag from memory. These instructions are `IRG`, `STG`, and `LDG`, respectively. The `IRG` instruction generates and attaches a tag to a pointer. Additionally, programs can provide a value to indicate an exclusion list to prevent the generation of certain tags. By default, this value is taken from the `GCR_EL1.Exclude` register. A user can modify this register using a system call. On the other hand, the `STG` instruction assigns a tag to a memory region, while the `LDG` instruction loads a tag from a memory region.

2.2 How modern software systems use MTE

Modern software tends to leverage MTE in their memory allocator implementation [8–10]. When a program requests memory from the allocator, the allocator assigns a randomly generated tag to the allocated memory and returns a pointer carrying this tag to the program. The program can freely access the allocated region with matching tags between the pointer and the allocated memory. When the program frees the allocated memory, the allocator removes the respective tag assigned to the allocated memory. When different allocations have distinct tags, a tag mismatch can occur if a pointer belonging to one allocation is used to access a different allocation. This mechanism can help detect spatial security issues, such as heap-based buffer overflows, because out-of-bound memory access causes a tag mismatch. Similarly, the mechanism helps detect temporal safety issues like use-after-free vulnerabilities. A violation may occur if the program uses a pointer carrying the previous tag to access the deallocated memory, which is now assigned to a different tag.

2.3 Limitations of current scheme

Memory allocators [8–10] were extended to leverage MTE. The allocator-based schemes rely on distinct tags assigned to different allocations, protecting against temporal and spatial memory safety issues. The allocators, by default, randomly generate a tag out of the 16 values. Accessing a different allocation using a pointer from a specific allocation may not lead to a violation if tags conflicted. To mitigate linear heap-based buffer overflows of out-of-range allocation, the current memory allocator implementations incorporate non-probabilistic schemes by either ensuring that two adjacent allocations have different tags or placing a guard granule between two allocations. This design ensures access that touches the out-of-bound memory region causes a violation. In scenarios where adversaries can launch repeated exploit attempts indefinitely, they can employ brute force attacks against the randomly generated tag to bypass the MTE protection. This attack scenario has been observed in real-world situations, such as local privilege escalation through setuid binary [2] or exploiting child processes of fork-exec daemon [21].

Moreover, the allocator-based schemes assign a single tag for all memory within the same allocation. This makes them ineffective in preventing vulnerabilities related to spatial memory safety that occur within the same allocation. These vulnerabilities, commonly

called intra-allocation safety issues, remain inadequately addressed by allocator-based schemes. An example of such an issue is intra-object overflow [13], where a buffer overflow occurring in a specific structure field can still overwrite fields behind this field.

CVE-2018–1160 is a classic example of a heap-based buffer overflow vulnerability by intra-object corruption. We provide a simplified version of the corresponding source code in Listing 1. This vulnerability is observed in Netatalk, an open-source file server. Line 12 specifies a function `dsi_opensession` that takes a pointer to a `dsi` object allocated from the heap. During program execution, an attacker can exploit this vulnerability by sending a crafted packet, which allows the program to reach line 19 in Listing 1. At this point, both the content of `dsi->commands + i + 1` and the value of `dsi->commands[i]` are derived from the packet and under the control of the attacker. Consequently, the attacker can manipulate `dsi->commands[i]` to a large value, enabling the overwriting of other member fields after `dsi->attn_quantum`. Subsequently, the attacker can exploit this vulnerability to achieve arbitrary write capabilities by modifying the value of `dsi->commands` and manipulating code pointers to gain control over the program's execution flow. Significantly, this vulnerability occurs within a structure's member field and can be exploited through the corruption of other member fields, i.e., an intra-object vulnerability. The state-of-the-art memory allocator employing a scheme that uses a single MTE tag for the same allocation cannot mitigate this vulnerability. Our work proposed HeMate to address the limitations of the current MTE usage.

Listing 1: Simplified code of CVE-2018–1160

```
1  // structure definition
2  typedef struct DSI {
3      ...
4      uint32_t attn_quantum, datasize, server_quantum;
5      uint16_t serverID, clientID;
6      uint8_t  *commands; /* DSI recieve buffer */
7      uint8_t  data[DSI_DATASIZ];    /* DSI reply buffer
       */
8      ...
9  };
10
11 // root cause of vulnerability
12 void dsi_opensession(DSI *dsi)
13 {
14   ...
15   /* parse options */
16   while (i < dsi->cmdlen) {
17     switch (dsi->commands[i++]) {
18     case DSIOPT_ATTNQUANT:
19       memcpy(&dsi->attn_quantum, dsi->commands + i + 1,
       dsi->commands[i]); // heap buffer overflow here
20       dsi->attn_quantum = ntohl(dsi->attn_quantum);
21   ...
22 }
```

3 THREAT MODEL AND ASSUMPTIONS

HeMate aims to protect user space C programs. We assume trusted sources developed these programs, but they contain vulnerabilities. An adversary can exploit the vulnerabilities to gain arbitrary memory read and write or code execution capabilities. We aim to mitigate such exploitation. This work focuses on heap-based spatial and temporal memory safety issues, which comprise well-known bug patterns such as buffer overflow, out-of-bound access, and use-after-free. Other safety issues, such as using uninitialized variables, integer overflow, format string bugs, and microarchitecture vulnerabilities [16, 26], are out of the scope of our work. We assume that several modern hardening mechanisms are enabled, including ASLR, W⊕X, and the hardware primitive MTE is supported by the CPU. Furthermore, we assume that the OS Kernel and the memory allocator of the user program are equipped with support for enabling MTE protection on heap pages.

4 DESIGN

4.1 Design Overview

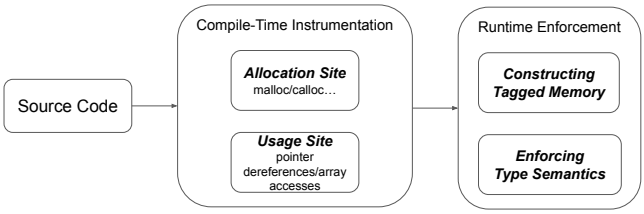

Figure 1: Workflow of HeMate

An overview of HeMate is presented in Figure 1. HeMate leverages MTE and provides run-time mitigation for programs written in the C language to mitigate memory safety issues. Specifically, it aims to mitigate heap-based spatial and temporal safety issues through compile-time instrumentation and run-time enforcement.

During compile time, HeMate performs allocation site detection and gathers relevant information, such as the allocation's base address, size, and type. Subsequently, it inserts instrumentation to maintain the memory assigned with the expected tag at run-time. The tag is a combination of two fields, one randomly generated after run-time memory allocation, similar to the tag in an allocator-based scheme, and the other field is determined at compilation time based on the intended type of memory.

If the memory is used for an aggregate data type, such as a `struct`, we tag it based on the type of each field within the aggregate data type. Therefore, HeMate enables assigning different tags to distinct memory regions within the same allocation according to their intended types instead of employing a uniform tag across all allocated memory. Because of that, HeMate relies on every allocation having a concrete type at the allocation site, which is typically expected in the coding style of C. At pointer dereferencing, usage site instrumentation is employed to attach a tag corresponding to the intended type of the pointer's dereferencing operation. Consequently, the intended type of the specific memory region at both the allocation site and the usage site must be consistent. Any mismatch in the tags would result in a run-time tag violation. This consistency provides a significant advantage in the event of a vulnerability, as it prevents the adversary from using a specific type of pointer to corrupt memory that belongs to different types. Hence, this mitigates the adversary's exploit capability and breaks some well-known techniques. HeMate also provides a variant to

developers who relax the type-matching enforcement scheme to support pointer casting (see Table 1).

4.2 Hybrid Tag Scheme

As mentioned above, HeMate divides the 4-bit MTE tag (i.e., 16 tags) into two fields with distinct 2-bit. The first field, referred to as **type field**, is assigned statically during compile time based on the intended usage of the memory (i.e., data type), and it is stored in bits 57-56 of the pointer. On the other hand, the second field, referred to as the **random field**, is randomly generated after runtime memory allocation and is stored in bits 59-58 of the pointer. This field is similar to the tag used in an allocator-based scheme. We refer to our proposed scheme, which utilizes two 2-bit fields, as the **hybrid tag scheme**. Figure 2 illustrates our tag scheme. This scheme compartmentalizes data of different types into separate groups while providing probabilistic protection within the same group. The **type field** significantly reduces the attack surface adversaries can exploit, while the **random field** constrains vulnerability exploitation's stability and success rate in the remaining attack surfaces. HeMate incorporates a fall-back scheme to leverage all MTE tags; that is, HeMate could utilize the 4-bit random bits when HeMate fails to distinguish variable type. In combination with the constraints that impose non-probabilistic protection that achieves data type isolation, HeMate results in improved protection guarantees compared to the state-of-the-art MTE usage scheme.

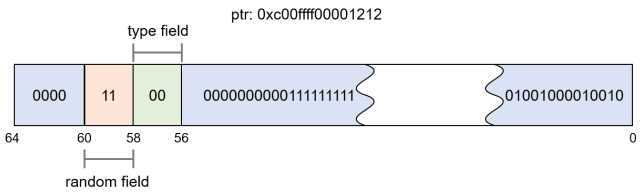

Figure 2: Hybrid Tag Scheme

4.3 Grouping primitive types

HeMate uses the type field to isolate primitive types into different groups. Since the type field is 2-bit, it can distinguish four group identifiers. We classified primitive types into four distinct groups based on their common usage.

(1) **Character**, Types are used for string literals or raw byte data, which are usually the most vulnerable to manipulation by malicious adversaries through user input or network packets. We classified the primitive data type **char** in C language to this group.
(2) **Numeric**, Types are used for numeric or bit-wise operations, which may still be derived from malicious adversaries. However, these types have distinct usage compared to character types, and we make the distinction to mitigate the impact of arbitrary corruption or tampering. This group includes all integer and floating-point data types, such as **int** and **float** in the C.
(3) **Pointer**, Types refer to specific addresses within the memory of a process. The data of those types is often the target of malicious adversaries and plays a crucial role in the exploitation chain. Therefore, it is imperative to isolate these types from others. All pointer types in C language belong to this group.
(4) **Untyped**, Types have no explicit data type or semantic meaning and may belong simultaneously to none or more of the aforementioned groups. We classified those types into separate groups. The **union** and **void** in C language belong to this group. HeMate allows developers to annotate variables in programs as the untyped group manually.

To prevent pointers of one group from pointing to memory regions belonging to another group, we can divide the primitive types into different groups, embed the group identifier in the type field of the tag, and then assign the memory with this tag. This approach provides an advantage regarding memory allocation and helps maintain data integrity by ensuring that the tag in the pointer and the tag assigned to memory regions of different groups are always distinct. Thus, including group identifiers within the tag's type field facilitates isolation, ensuring non-probabilistic protection, and prevents cross-group memory access.

We observed that specific legacy C applications perform typecasting from the Character group to the Numeric group during execution. As a result, dividing them into separate groups may pose challenges when applying our scheme to such applications. We have devised a relaxed variant of our scheme to support these applications. In this variant, we combine the Character and Numeric groups into a single group called the Data group. The relaxed scheme encompasses three groups: pointer, data, and untyped. Table 1 describes the type field tags corresponding to the groups in the normal and relaxed variant.

Table 1: Type group and corresponding type field tag

Normal		Relaxed	
Group	**Tag**	**Group**	**Tag**
Untyped	`0b00`	Untyped	`0b00`
Pointer	`0b01`	Pointer	`0b01`
Numeric	`0b11`	Data	`0b10`
Character	`0b10`		

4.4 Tagging memory

HeMate aims to protect against heap-based vulnerabilities and guarantee spatial and temporal memory safety. Specifically, this addresses heap-based buffer overflow and use-after-free, which correlates with allocation lifetime and spatial relationship. Therefore, HeMate tags the memory for each allocation after the lifetime begins and before the first access, incorporating specific information about the spatial relationship, such as allocation size and the intended layout of the allocated memory. In order to accomplish this, HeMate identifies each function call that triggers the allocation function. It then passes the allocation address and size to the corresponding memory tag constructor, created based on the specified type of allocation. The memory tag constructor is responsible for setting tags for the allocated memory using the MTE instructions. During allocation, the associated pointer to the allocated memory

does not contain the type field value. HeMate attaches the tag at the usage site according to the data type of the associated allocated memory.

Listing 2: Example of tagging memory

```
1  struct obj {
2    char name[0x10]; // character
3    int *list; // pointer
4    int cnt; // numeric
5  };
6
7  void foo () {
8    struct obj *pObject = malloc(sizeof(struct obj));
9    pObject->cnt = 1;
10 }
```

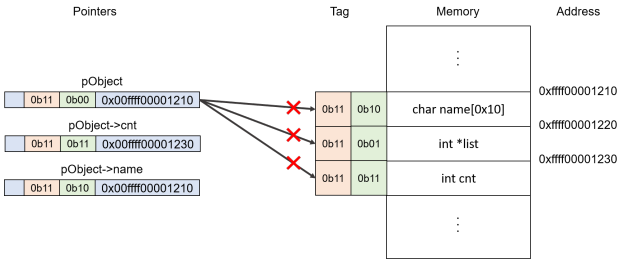

Figure 3: Run-time memory layout, accompanied by the corresponding memory tag.

Figure 3 presents an example of the run-time memory layout, accompanied by the corresponding memory tag, for the code provided in Listing 2. In line 8 of Listing 2, a function call is made to the allocation function `malloc`. Subsequently, we introduce allocation site instrumentation after the return of the allocation function. In the allocation site instrumentation, The address and size of the allocated memory are then passed to the memory tag constructor for the type `struct obj*`. The responsibility of the memory tag constructor is to set tags for the allocated memory using the MTE instructions. Notably, since `struct obj` is an aggregate type, the memory tag constructor assigns tags with distinct type field values to the memory associated with the various fields within the structure based on the types of the fields. Consequently, allocated memory from the same allocation can be tagged with different tags rather than a uniform tag. As depicted in Figure 3, each member of the `struct obj` has a unique tag based on their respective type (values in the green squares). Finally, the pointer to allocated memory is stored to `pObject` without type field value.

4.5 Load and Store Semantic

The absence of a type field in stored pointers causes a mismatch with the memory tags they point to. To address this, it is necessary to compute the actual pointer by including the appropriate type field value before dereferencing it. Based on contextual semantics, this approach prevents type confusion between different type groups and mitigates exploitation techniques. Additionally, computing the actual pointer with a contextual type field instead of using the pointer directly eliminates the risk of modifying the type field in the stored pointer. In a specific example provided, storing a value to `pObject->cnt` in heap memory without the type field value leads to a tag mismatch during memory load operation. This mismatch occurs because the memory region's tag is based on the type of `cnt`, while the stored pointer `pObject` lacks a type field value. To resolve this issue, usage site instrumentation is introduced to attach the type field value to the pointer `pObject->cnt`. As shown in Figure 3, the scheme prevents attackers from hijacking a pointer with a mismatch `pObject` type to access members in the structure.

5 IMPLEMENTATION

Our prototype of HeMate is built upon LLVM Project 16.0.0 [11], focusing on its C compiler sub-project, Clang. The majority of modifications are made within Clang's LLVM Intermediate Representation (IR) code generation (*CodeGen*); the implementation extended its C to LLVM IR code generation phase. Our approach relies on compile-time instrumentation, divided into two main components: allocation site and usage site instrumentation. The decision to generate instrumentation before the IR optimization pass is due to the availability requirement of semantic information from the C source code, including type semantics and the Abstract Syntax Tree (AST). As this crucial information might not be present in the LLVM IR optimization pass, it became imperative to perform instrumentation at an earlier stage to ensure the availability of the necessary semantic information.

5.1 Aligning structure layout with MTE granularity

MTE supports tag assignment to memory at a 16-byte granularity. However, in cases where adjacent fields within a structure belong to different type groups and, therefore, have different tags, they cannot be placed within the same 16-byte granule. To address this issue, we utilize padding to ensure that the second field aligns with the MTE granularity. During the construction process of the structure layout, if a member field's tag differs from the following member field's tag or the following member field is a structure, we insert padding after the current member field. This padding ensures that the following member field aligns with the MTE granularity.

5.2 Detecting memory allocation functions

Instrumenting allocation sites is a vital aspect of HeMate. It is essential to effectively detect all memory allocation functions to ensure that the allocated memory is consistent with the expected tags. Failure to do so may lead to additional developer effort or run-time tag violations, as our instrumentation is also inserted at the usage site.

The HeMate prototype conducts a function target analysis for each function call during compilation time. We rely on the target function's information, such as function attributes, to determine if it corresponds to a memory allocation function. Several function attributes [5] in Clang, such as `allockind` and `alloc_size`, are commonly utilized by memory allocation functions. These attributes provide the compiler with additional information for performing checks or optimizations. As a result, most library providers annotate these attributes in their memory allocation functions. We chose to detect the `alloc_size` attribute. By utilizing the information

provided by `alloc_size` attribute, we can extract valuable information from the function call to the allocation function. Specifically, we can obtain details such as the requested allocation size and the base address of the allocated memory. This information is then utilized in our allocation site instrumentation. The `alloc_size` attribute is commonly present in the function declarations of standard memory allocation functions, including `malloc()`, `realloc()`, and `calloc()`. However, it is worth noting that certain functions that implicitly allocate memory might not include the `alloc_size` attribute. Examples of such functions include `strdup()`, `readline()`, or custom allocation functions defined by the application. We have added attributes for these functions, enabling the developer to specify how to get the base, size, and type of the allocation. For applications that define the custom allocation functions, developers may append these attributes in their custom-define allocation functions to ensure the appropriate generation of our instrumentation.

5.3 Instrumentation

5.3.1 Allocation Site Instrumentation. In HeMate, we insert instrumentation after function calls that invoke the memory allocation function. This instrumentation collects information such as the base address and size of the allocation. In addition, the HeMate compiler infers the allocation's destination types by traversing the source program's AST. This inferred type is then used to find the appropriate memory tag constructor. Subsequently, the base address and size will be passed as parameters to the memory tag constructor to tag the allocated memory further. We generate the appropriate constructor at the first memory allocation for each type. Since the memory allocation may require memory for one or more elements of a particular type, we treat the allocated memory as an array. Therefore, we utilize a loop to call the actual constructor element by element, which utilizes MTE's `STG` instruction to assign the tag we expect to the memory of a particular type.

Listing 3: LLVM IR of allocation site instrumentation

```
1 %call = call noalias ptr @malloc(i64 noundef 16) #4
2 %3 = call ptr @llvm.aarch64.irg(ptr %call, i64 61166)
3 %4 = call ptr @"__memtag_ctor_struct_foo_*"(ptr %3, i64 16)
```

Listing 3 shows an example of allocation site instrumentation. A function call to the allocation function is present in the first line. We introduce instrumentation immediately after the function return. Subsequently, we generate the tag with only a random field value in the second line using the MTE instruction `IRG` and indicating the excluded tags. The allocation size and base address are passed as arguments to the memory tag constructor in the third line. Within the memory tag constructor, the allocated memory is treated as an array of `struct foo`, and appropriate tags are assigned to the allocated memory using MTE instruction `STG`.

5.3.2 Usage Site Instrumentation. Before memory load or store instructions, we must attach a type field value to pointers stored without a type field value at the allocation site. We achieve this by inserting instrumentation while loading and storing variables. These operations involve memory loading or storing instructions when reading and storing variables. To prevent an adversary from forging the tag, a bit mask instruction eliminates the type field value before attaching the type field to the pointer. Listing 4 and Listing 5 respectively illustrate the un-instrumented and instrumented IR for memory load operation, with the latter demonstrates the results of HeMate's usage site instrumentation. It is worth noting that HeMate instruments store operations similarly to the load operations.

Listing 4: LLVM IR example of a memory load

```
1 %2 = load i32, ptr %1, align 4
```

Listing 5: LLVM IR example of instrumented memory load

```
1 %2 = ptrtoint ptr %1 to i64
2 %3 = and i64 %2, -216172782113783809
3 %4 = or i64 %3, 216172782113783808
4 %5 = inttoptr i64 %4 to ptr
5 %6 = load i32, ptr %5, align 4
```

In the first line of the Listing 4, the memory load operation of an integer type is executed on the address indicated by the pointer `%1`. On the other hand, Listing 5 demonstrates the insertion of the instrumentation for this memory load operation. In lines 1 and 4 of Listing 5, we perform a pointer-to-integer cast operation followed by a reverse operation, which is necessary for the LLVM IR to perform the subsequent bit-wise operation. This only affects the semantic information on the IR, and it does not reflect on the final machine instructions. After line 1, in line 2, the type field information is eliminated from the pointer by performing a bit-wise `and` operation with a bitmask `-216172782113783809`, which equals to `0xfcffffffffff` in hexadecimal. This bitmask includes all 1s except for bits 56 and 57. Then, in line 3, the type field information is attached to the pointer through the bit-wise `or` operation with a type field value. In the example shown in Listing 4, the type value `0x11` is used. The value `216172782113783808` corresponds to the value `0x300000000000000` in hex, where bits 56 and 57 are set to 1s, representing the type field value for the numeric group. Finally, line 5 performs the same memory load operation via the secured pointer.

6 EVALUATION

6.1 Functional Evaluation

We aim to validate whether the binaries compiled with the HeMate prototype preserve functionality when running an MTE-based environment. We were unable to acquire the Arm hardware with MTE support at the time of writing. Therefore, we utilized the QEMU 7.0.0 emulator, which emulates the hardware functionality of MTE, to test the resulting binaries. The software stack is based on Linux 5.18.0 and GLIBC 2.36. Both are enabled with MTE support. We selected roughly 70 binaries from GNU Coreutils for functional testing and compiled them with HeMate to evaluate their primary usage. All resulting binaries compiled by HeMate functioned correctly, suggesting HeMate's compatibility of real hardware with MTE support.

6.2 Performance Evaluation

Additionally, we collected the applications and benchmarks for performance evaluation and executed them with the corresponding test data to measure performance. To assess the performance overhead of our instrumentation required for HeMate, we conducted experiments using 15 benchmarks selected from the MiBench suite [14] as well as four real-world applications, xz, x264, md5, and

baseenc. All benchmarks were executed on a machine with Arm Neoverse-N1 CPU and 32G RAM, running a software stack based on Ubuntu 22.04.1 LTS, with Linux 5.10.107 and GLIBC 2.35. All benchmarks were compiled by HeMate with the -O3 optimization.

The Arm CPUs on our machine do not support the MTE feature. Thus, we used an analog instruction method to estimate the upper bound overhead, as previously done in MTE-related works[17, 22]. Specifically, we replaced MTE instructions, each with a set of Arm instructions during the machine instruction generation process of LLVM. The respective set of Arm instructions results in an overhead similar to the MTE instruction. The instruction overhead is referenced to exec latency, execution throughput, and utilized pipelines in the Arm Cortex-X2 Core Software Optimization Guide [12], which is for ARMv9-A based CPU with MTE feature.

Our proposed method only uses two MTE-related instructions: `STG` and `IRG`. The specific instructions are listed in Listing 6 and Listing 7.

Listing 6: STG Instruction Analogue

```
stg xT, [xN]:
    ldr     x17, [xN]
    str     x17, [xN]
```

Listing 7: IRG Instruction Analogue

```
irg xD xN:
    mov x17, #0x300000000000000
    bics xD, xN, x17
```

Since MTE is unsupported by our testing hardware, we make the same assumption as previous MTE-based works regarding tag checking during memory access: that the overhead of hardware implementation for tag checking would be minimal and can be ignored.

To evaluate the performance overhead of our approach, we measured benchmarks compiled with two different implementations: **allocation and usage site instrumentation**, **allocation site instrumentation**. In subsequent discussions, we compared with un-instrumented binaries compiled with clang, which we refer to as **beseline**. For each benchmark, we executed them ten times and recorded the userspace execution time using the **time** command from GNU Bash. The results are presented in Figure 4, where the results are normalized against the execution time of the baseline. The average overhead was calculated by taking the geometric mean of the execution times of each benchmark.

As shown in Figure 4, the average overhead of HeMate with full protection (**allocation and usage site instrumentation**) is approximately 25%. The average overhead caused by allocation site instrumentation is approximately 3%. The overhead incurred by allocation site instrumentation aligns with the overhead observed in allocator-based schemes mentioned in the previous study[24]. The allocation site instrumentation involves recoloring the allocated memory after each allocation request, thereby introducing recoloring overhead. In most benchmarks, this overhead is relatively low, typically below 10%. This is because memory allocation typically occurs as a one-time event during program initialization or object initialization. As a result, the recoloring overhead does not significantly impact the overall overhead.

We found that usage site instrumentation attributed to most overhead. We analyzed the overheads observed in different benchmarks to investigate the root cause. The usage site instrumentation of HeMate resulted in various degrees of overhead in different applications. As mentioned in Section 5, the current HeMate implementation instruments the program during LLVM's IR generation. Because it is impossible to identify runtime memory usage during the IR generation phase, we currently insert usage site instrumentation to all variable accesses through pointer dereferences and array accesses, regardless of whether the accesses were made from heap memory or not. In other words, usage site instrumentation is inserted even if the pointer is used against the stack or global memory, in addition to heap accesses. We detail this limitation further in Section 8. Moreover, given the instrumentation is made during the IR generation stage, it affects the efficacy of subsequent LLVM optimization phases, making them unable to optimize the program's memory accesses in the generated binary. This also results in performance slowdown in memory-intensive applications. For the above reasons, we observed higher overhead incurred by HeMate in x264, which resulted in roughly 98 % longer execution time compared to the baseline. We discuss a potential solution to optimize the overhead in Section 9.

We found that using HeMate led to increased memory overhead in applications that are memory-intensive. Applications that frequently perform memory load/store operations through pointer dereferences and array accesses will experience high overhead. This is evident in programs dedicated to data encoding, like x264. Since the data encoding process takes up most of the program's execution time, frequent memory load and store operations through pointer dereferencing or array accessing will cause significant performance overhead when instrumentation is added at such points. On the other hand, HeMate incurs modest overhead to mathematical applications that focus mainly on arithmetic operations and do not involve frequent pointer dereferencing and array accessing.

7 SECURITY ANALYSIS

We evaluated real-world known vulnerabilities and exploitation techniques to assess the effectiveness of HeMate in mitigating vulnerabilities.

7.1 Memory Safety Protection

Heap-based intra-object buffer overflow. HeMate utilizes different tags for member fields within distinct groups. As for CVE-2018–1160 that we discussed in Section 2.3, if the vulnerable program is compiled by HeMate, the attacker cannot perform intra-object corruption against the `commands` pointer and overwrite it. Note that when a program is compiled by HeMate, adjacent fields that belong to the same group, such as `datasize` and `server_quantum`, `serverID` and `clientID` share the same type tag. HeMate assigns a randomly generated tag to the random field of each pointer to provide isolation between these types.

Use after Free. HeMate is effective against real-world techniques that exploit use-after-free vulnerabilities. An attacker may reallocate memory to a different type, controlling the values on the memory through the field that stores data in one structure. In contrast, another structure may treat the same memory as a pointer.

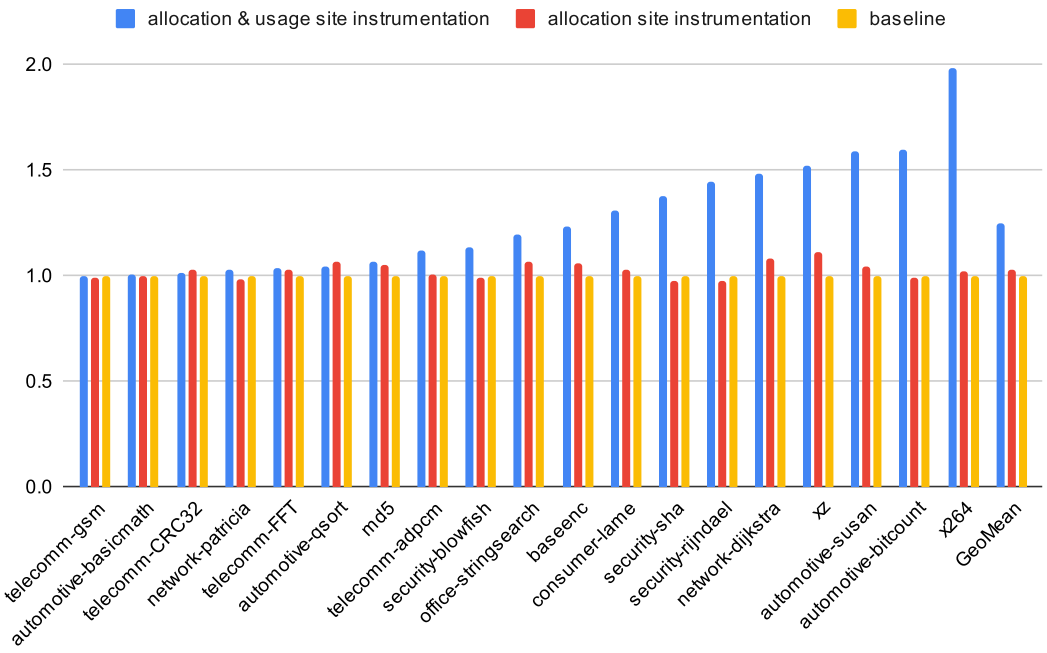

Figure 4: Performance overhead

These scenarios are the most common because the data type of memory is the easiest for the attacker to manipulate, while the memory that stores pointers is the target that the attacker wants to corrupt[3]. Our work protects against this scenario because the exploit involves confusion across type groups. In addition, an attacker can exploit the memory that stores the same type in both structures to cause unexpected behavior. However, we still provide a 2-bit random field to prevent such an exploit.

7.2 Security Guarantees Discussion

Here, we assess the potential for an adversary to tamper with the tag to compromise our isolation. The attacker may attempt to modify the pointer stored in memory or manipulate the type field during pointer arithmetic. The current allocator-based scheme (see Section 2.3) is vulnerable to both attack scenarios since the tag is permanently attached to the pointer value. In HeMate, any attempt by an adversary to corrupt the pointer stored in memory using a character pointer or a numeric pointer will lead to a tag mismatch. Moreover, upon dereferencing the pointer at the usage site, we apply a bit-wise mask operation to eliminate the type field value before performing the bit-wise OR operation to attach the type field value. Consequently, any modification to the type field before attaching the actual type field would be ineffective.

8 LIMITATIONS

8.1 Classifying untyped variables

In the current HeMate implementation, we classify variables that we cannot determine the actual type to the untyped group. For types in this group, we assign the untyped value to the type field of the entire memory region and a randomly generated value to the random field. This offers similar protection to the allocator-based MTE schemes [8–10]. We list all scenarios that HeMate classifies as untype below.

8.1.1 Missing type at allocation site. HeMate assumes that every allocation is associated with a type at the allocation site. However, when the type is unknown (e.g. `void*`) at the allocation site, a runtime tag violation may occur at the usage site, which expects the actual type. As shown in Listing 8, the type of allocation is defined as a pointer to void at the allocation site; this pointer is cast and dereferenced with char in the subsequent code. Developers can explicitly classify the allocation as untyped or modify the program to match the types used at the allocation to the usage site.

Listing 8: Example of missing type at allocation site

```
void allocate_site() {
  void *ptr = malloc(0x10);
  char *buf = ptr;
  buf[1]; // crash
}
```

8.1.2 Mixed use of multiple types. Another scenario is illustrated in Listing 9, where the allocation type mismatches the usage site between the numeric and character groups. As shown in Table 1, HeMate provides a relaxed variant that groups the two types into a single data type to support such casting.

Listing 9: Example of mixed use of multiple types

```
void usage_site() {
```

```
  char *ptr = malloc(0x10);
  int *arr = ptr;
  arr[1]; // crash
}
```

8.2 Compatibility

8.2.1 Compatibility with external unprotected functions. Compiling all functions and structures with HeMate is best to ensure efficacy. However, this is not always practical when dealing with precompiled binaries. These binaries may be compiled from proprietary sources. HeMate allows developers to use the untyped annotation for structures that cannot be compiled with the HeMate compiler. When linked against binaries (e.g., dynamically linked libraries) unprotected by HeMate, our prototype currently preserves the original layout for structures to maintain compatibility with external functions.

8.2.2 Compatibility with memory manipulation functions. In C, memory manipulation functions such as `memcpy()` or `memset()` are commonly used for setting values within a memory region. It is also common practice to use these functions for initializing or copying structure instances instead of individual assignment statements for each member field of the structure. However, these functions pose a challenge for our scheme since an aggregate type's memory region is chromatic, and using a pointer with a specific type field information cannot represent all member fields since there may be member fields of different types within the structure. The HeMate implementation identifies invocations of memory manipulation functions during compilation. It unwraps these functions into multiple memory manipulation functions, each dedicated to handling consecutive memory regions in the structure that share the same tag.

Listing 10: memset before slicing

```
1  struct obj {
2    char name[0x10]; // character
3    int *list; // pointer
4    int cnt; // numeric
5  };
6
7  void foo() {
8      struct obj *pObject = malloc(sizeof(struct obj));
9      memset(pObject, 0, sizeof(struct obj)); // zero-
       initialization
10 }
```

In Listing 10, line 8 uses an `memset` to zero-initialize the memory that is pointed by `pObject`. Listing 11 shows that HeMate slices the single `memset` into three `memset` for each members.

Listing 11: memset after slicing

```
1  void foo() {
2      struct obj *pObject = malloc(sizeof(struct obj));
3      memset(pObject->name, 0, offsetof(struct obj, list)
       );
4      memset(pObject->list, 0, offsetof(struct obj, cnt)
       - offsetof(struct obj, list));
5      memset(pObject->cnt, 0, sizeof(struct obj) -
       offsetof(struct obj, cnt));
6  }
```

8.2.3 Reference untyped object in a function call. A pointer that refers to an untyped memory region may be passed to a function. In that case, the developer needs to ensure that the callee function is aware that the argument from the caller may be untyped. This is necessary because we insert instrumentation at the usage site. The scenario is illustrated in Listing 12. In the function foo, the char array within a union is classified as untyped and passed as an argument to the function bar. Within the function bar, there is a pointer, p, pointing to the char type. This pointer will later point to buf, which may be an untyped pointer. At the usage site of p, the instrumentation is inserted to load char from p. Therefore, extra efforts are required for the developer to annotate p to inform the callee function about the actual type of p, which may belong to the untyped group. HeMate does not attach the type field value here. Instead, it uses the untype value attached before the function call.

Listing 12: Reference untyped object in a function call

```
1  void bar(char *buf) {
2    char *p;
3    p = buf;
4    p[1]; // crash
5  }
6
7  void foo() {
8    union {
9      char buf[0x10];
10   } *u = malloc(sizeof(*u));
11   bar(u->buf);
12 }
```

9 FUTURE WORK

The current HeMate implementation instrument program during LLVM's IR generation phase. This decision gives us some advantages in reusing LLVM's optimization. However, in our experiments, we have observed that this approach also leads to the pattern matching of the existing optimization not being detected correctly. A possible alternative is to embed the type information into the LLVM IR during the IR generation phase and then instrument the program after the optimization phase. Moreover, the current HeMate implementation instrument memory load/store operations via pointer dereference and array accesses, regardless of whether the pointer points to a heap region. This decision is because we cannot determine which region the pointer points to during LLVM's IR generation phase. An improvement would be to analyze all possible regions the pointer could point to during the LLVM's optimization phase and then decide whether to insert the instrumentation. This analysis could allow us to eliminate some of the redundant usage site instrumentation.

10 RELATED WORK

MTE-based protection. MTE has been widely integrated into various memory allocators used in software applications, such as Scudo [10], PartitionAlloc [8], and Ptmalloc2 [9]. These allocators adopt a similar scheme to leverage MTE, providing probabilistic protection against spatial and temporal vulnerabilities on the heap. In these allocators, the memory is retagged during memory allocation and deallocation. However, since the allocator only knows the size of the allocation request and has no other information, unlike HeMate,

it can only assign a uniform tag to the allocated memory. Therefore, it cannot prevent intra-allocation spatial safety issues, i.e., intra-object overflow, and only provides probabilistic protection against memory safety violations that between different data types. In contrast, we use the type of allocation and the expected layout to do the tagging so that an allocation can contain regions with different tags to protect against intra-allocation spatial safety issues and non-probabilistic protection.

In addition to memory allocators from the heap, MTE is also utilized to protect the OS kernel and stack usages. For instance, HAKC[22] leverages PAC and MTE to achieve compartmentalization in the kernel. This allows developers to assign different resources, such as variables, to separate compartments. In contrast, we target user space applications and emanating tags by type rather than developer assignment, reducing the developer effort. Color My World [17], a stack-based protection mechanism that utilizes MTE to achieve compartmentalization. It employs LLVM's StackSafetyAnalysis [1] to classify stack variables as safe or unsafe and isolates these variables into different compartments. However, this work is only for stack protection, which cannot be directly applied to heap allocation, while our work focuses on heap protection and provides novel approaches to apply MTE to heap allocation.

Heap-based UAF mitigation. Kalloc_Type [3] is a mitigation technique introduced to address the exploitation of UAF vulnerabilities from MacOS. The UAF vulnerabilities usually involve confusing memory regions as different types, particularly between pointer types and data types. To prevent this exploitation, Kalloc_Type generates a signature for each structure at compilation time, concatenated by the type of each member field. These signatures are then provided to the XNU zone memory allocator, which allocates memory from distinct memory zones given the different signatures. As a result, attackers cannot induce type confusion between pointer types and data types on member fields since they possess different signatures. Google Chrome's PartitionAlloc [20] adopts a similar approach to Kalloc_Type to allocate different types of objects from distinct partitions. IisoHeap [15] ensures that a virtual address is assigned to a particular type only once, and it will not be reused for any other type. Unlike HeMate, the approaches did not support memory allocators from standard libraries (e.g., ptmalloc) and did not use MTE in their design to leverage its hardware protection.

Hardware assisted mitigation. CHERI [27] is an extension to the Arm architecture that introduces capabilities, unforgeable and bounded pointers, to enhance the representation of traditional pointers. The need for this extension arises from the need for more distinction between data and pointers in Arm's ISA. To address this issue, CHERI utilizes a 1-bit memory tag to differentiate between pointers and data stored in memory. This rationale is similar to our motivation to separate primitive types based on their intended usage. In addition to distinguishing data and pointers, CHERI employs bounded pointers to address spatial safety concerns. By bounding pointers' capabilities, CHERI enhances the security of memory accesses. It is worth noting that while CHERI provides spatial protection, it does not inherently address temporal issues such as use-after-free vulnerabilities, as attackers can still exploit existing pointers. To mitigate such vulnerabilities, CHERI must collaborate with a specialized memory allocator [28, 29]. This collaboration introduces challenges related to performance overhead and applicability. In contrast, HeMate can be used directly with existing allocators from legacy C libraries.

11 CONCLUSIONS

This paper introduces a new system, HeMate, which improves the existing MTE by adding a non-random field to the tag. This provides a protection mechanism that is not based on probability, in addition to MTE's probabilistic protection. HeMate specifically focuses on isolating primitive types within the heap memory. This choice is based on the observation that attackers often exploit vulnerabilities by treating a memory region as different types. We use semantic information during memory allocation and pointer dereferencing to achieve this isolation and ensure consistency. To validate our approach, we created a prototype compiler for C language programs based on the LLVM project and tested it on real-world Linux applications and vulnerability patterns, demonstrating the effectiveness of enhancing programs' memory safety. In addition, we showed that our approach preserves the functionality of Linux applications while incurring modest performance to most these applications.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback. This research was supported in part by the National Science and Technology Council of Taiwan under research grants 111-2628-E-002-022- and 112-2628-E-002-027-.

REFERENCES

[1] 2020. Stack Safety Analysis — LLVM 11 documentation. https://releases.llvm.org/11.0.0/docs/StackSafetyAnalysis.html

[2] 2022. CVE-2021-3156: Heap-Based Buffer Overflow in Sudo (Baron Samedit) | Qualys Security Blog. https://blog.qualys.com/vulnerabilities-threat-research/2021/01/26/cve-2021-3156-heap-based-buffer-overflow-in-sudo-baron-samedit [Online].

[3] 2022. Towards the next generation of XNU memory safety: kalloc_type. https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety [Online].

[4] 2023. Arm Memory Tagging Extension. https://source.android.com/docs/security/test/memory-safety/arm-mte [Online; accessed 25. Aug. 2023].

[5] 2023. Attributes in Clang — Clang 17.0.0git documentation. https://clang.llvm.org/docs/AttributeReference.html#amd-gpu-attributes [Online; accessed 30. Jun. 2023].

[6] 2023. CWE - 2022 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html [Online].

[7] 2023. Memory safety. https://www.chromium.org/Home/chromium-security/memory-safety [Online].

[8] 2023. PartitionAlloc Design. https://chromium.googlesource.com/chromium/src/+/master/base/allocator/partition_allocator/PartitionAlloc.md

[9] 2023. Ptmalloc2. https://sourceware.org/git/?p=glibc.git%3Ba=blob%3Bf=malloc/malloc.c%3Bh=bd3c76ed310c4c2cbf8f141eb6b76182926cf24a%3Bhb=refs/heads/release/2.36/master

[10] 2023. Scudo. https://source.android.com/docs/security/test/scudo

[11] 2023. The LLVM Compiler Infrastructure Project. https://llvm.org [Online].

[12] Arm. 2022. Arm Cortex-X2 Core Software Optimization Guide r2p1. *Arm* (Jan. 2022). https://developer.arm.com/documentation/PJDOC-466751330-14955/latest

[13] Jacob Baines. 2021. Exploiting an 18 Year Old Bug. *Medium* (Dec. 2021). https://medium.com/tenable-techblog/exploiting-an-18-year-old-bug-b47afe54172

[14] 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)*. IEEE, 3–14.

[15] Apple Inc. 2021. IsoHeap from WebKit. https://github.com/WebKit/WebKit/blob/main/Source/bmalloc/bmalloc/IsoHeap.h

[16] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2020. Spectre attacks: Exploiting speculative execution. *Commun. ACM* 63, 7 (2020), 93–101.

[17] Hans Liljestrand, Carlos Chinea, Rémi Denis-Courmont, Jan-Erik Ekberg, and N. Asokan. 2022. Color My World: Deterministic Tagging for Memory Safety. arXiv:2204.03781 [cs.CR]

[18] 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 *28th USENIX Security Symposium (USENIX Security 19)*. 177–194.

[19] Arm Limited. 2024. Arm Memory Tagging Extension Whitepaper.

[20] Google LLC. 2024. PartitionAlloc Design. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/base/allocator/partition_allocator/PartitionAlloc.md

[21] m4drat. 2023. CVE-2013-2028-Exploit. https://github.com/m4drat/CVE-2013-2028-Exploit [Online; accessed 16. May 2023].

[22] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. 2022. Preventing kernel hacks with HAKC. In *Proceedings 2022 Network and Distributed System Security Symposium. NDSS*, Vol. 22. 1–17.

[23] microsoft. 2023. SSTIC2020 - Pursuing Durably Safe Systems Software. https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2020_06_SSTIC/SSTIC2020-PursuingDurablySafeSystemsSoftware.pdf [Online].

[24] Kostya Serebryany. [n. d.]. ARM Memory Tagging Extension and How It Improves C/C++ Memory Safety.

[25] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, USA, 28.

[26] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 88–105.

[27] Robert N. M. Watson. 2019. Capability Hardware Enhanced RISC Instructions (CHERI). https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/

[28] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. 2020. Cornucopia: Temporal Safety for CHERI Heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*. 608–625. https://doi.org/10.1109/SP40000.2020.00098

[29] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. 2019. CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 545–557. https://doi.org/10.1145/3352460.3358288