Look Before You Access: Efficient Heap Memory Safety for Embedded Systems on ARMv8-M

Jeonghwan Kang
Pusan National University
jeonghwan@pusan.ac.kr

Jaeyeol Park
Pusan National University
woduf0628@pusan.ac.kr

Jiwon Seo
Dankook University
jwseo@dankook.ac.kr

Donghyun Kwon*
Pusan National University
kwondh@pusan.ac.kr

Abstract

Numerous embedded systems utilize firmware written in memory-unsafe C/C++. So, the firmware may exhibit spatial memory vulnerabilities, such as buffer overflows, which, if exploited by an attacker, can lead to various software attacks. While several studies have proposed defenses against these memory vulnerabilities, they often introduce significant performance and memory overhead or are impractical for application in embedded systems. In this paper, we introduce micro-fat pointer, a novel solution for heap memory safety for embedded systems. Notably, micro-fat pointer leverages TT instructions newly introduced in ARMv8-M to implement an efficient bounds-checking mechanism. Our evaluation results demonstrate that micro-fat pointer exhibits a 41% performance improvement compared to the existing state-of-the-art heap memory safety solution.

1 Introduction

With the widespread adoption of the Internet of Things (IoT), embedded systems are extensively utilized in various fields [8] such as smart homes, automobiles, medical devices, and industrial automation. However, alongside this prevalence, security incidents targeting these embedded systems also continue to be reported [11, 23]. Moreover, due to various system resource management and real-time constraints, many embedded systems are developed using memory-unsafe C/C++ languages, resulting in various spatial memory vulnerabilities (*e.g.*, buffer overflow) within the program. In particular, if an attacker exploits the memory vulnerabilities in the program, she can launch diverse software attacks such as data manipulation attacks and code reuse attacks.

To mitigate spatial memory vulnerabilities, numerous research efforts [1, 18–20, 22, 28, 29, 31, 33, 35] have been conducted. Among them, one well-known approach is *bounds checking*. In this approach, the bounds of each memory object are recorded as bounds metadata (*i.e.*, base and size). Then, during pointer operations, they detect spatial memory errors by checking the bounds metadata whether the pointer resides within the bounds of the corresponding memory object. However, despite this security benefit, early studies [6, 29] for bounds checking have the drawback of introducing significant performance and memory overhead. This is because they need to store bounds metadata in program memory and perform additional memory operations to manage this metadata.

For these reasons, recent bounds checking studies [12–14, 19, 20, 28] have focused on efficient bounds metadata management. In particular, low-fat pointer [12–14] has garnered significant attention by proposing a scheme that embeds bounds metadata in the pointer itself to mitigate the performance overhead associated with bounds metadata management. Specifically, it first partitions the address space into several regions of equal size (*e.g.*, 4 GB regions), ensuring that each region contains objects of the same allocation size. Then, during the bounds checking operation, it can retrieve bounds metadata by just examining the upper bits of the pointer (*e.g.*, [32:63] bits in a 64-bit pointer). These upper bits indicate the region to which the object belongs, facilitating the determination of the object's allocation size. Subsequently, the base address can be calculated by dividing the pointer value by the allocation size. However, according to our preliminary analysis (see § 5), there are several challenges to applying low-fat pointer to embedded systems that do not have virtual memory and only have small physical memory.

In this paper, we propose micro-fat pointer, a new heap memory safety solution for embedded systems. micro-fat pointer has implemented an efficient bounds checking mechanism using memory protection unit (MPU) and test target (TT) instruction provided by the ARMv8-M architecture. Specifically, micro-fat pointer divides the address space into several MPU regions, ensuring that objects with the same allocation size are allocated in each region. By doing this, micro-fat pointer encode bounds metadata in the MPU region number. Then, to efficiently retrieve the MPU region number during bounds checking operation, micro-fat pointer utilizes the TT instruction in ARMv8-M. Our experimental results show that micro-fat pointer incurs only 158.97% of performance overhead, which is much lower than existing low-fat pointer.

2 Related Work

2.1 Spatial Memory Safety

There are a lot of studies for ensuring spatial memory safety [1, 18–20, 22, 28, 29, 31, 33, 35]. Recent research has particularly focused on reducing memory and performance overheads for bounds checking. Notably, Baggy Bounds [1], low-fat pointers [12–14], Delta pointers [19], and SGXbounds [20] have significantly reduced memory overhead caused by bounds metadata by altering pointer representations to include bounds metadata. However, these studies were all designed for systems that support virtual memory, making them unsuitable for direct application to embedded systems without virtual memory support. On the other hand, some researchers have proposed efficient bounds checking mechanisms utilizing instruction set extensions such as ARM pointer authentication [22] and Intel memory protection extension [31]. However, these extensions

*Corresponding Author

DAC '24, June 23–27, 2024, San Francisco, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3655949

are available only for high-performance systems, making the aforementioned studies impractical for embedded systems. In contrast, micro-fat pointer has been designed based on the TT instruction in ARMv8-M which is widely used in embedded systems.

2.2 Embedded System Security

Numerous studies have been conducted on embedded system security; memory safety [21, 27], compartmentalization [9, 15, 16, 26, 37], and control-flow integrity [2, 25, 30, 36]. ret2ns [25] proposes a novel control-flow hijacking attack that exploits the switching mechanism between a non-secure state and a secure state in TrustZone-M. They also present a code instrumentation technique using the TT instruction to detect their proposed attack by inspecting the destination address during state switching. Similarly, micro-fat pointer also utilizes the TT instruction but differs in its application, focusing on bounds checking during pointer operations. Like micro-fat pointer, nesCheck[27] and Tock [21] have introduced memory safety solutions targeting embedded systems. However, applying them requires rewriting programs in languages such as nesC or Rust, whereas micro-fat pointer can be directly applied to programs written in C.

3 Background

In this section, we provide brief information on several features of ARMv8-M, which are necessary for comprehending micro-fat pointer.

3.1 System Address Map

In ARMv8-M, the address map consists of several regions such as *ROM*, *SRAM*, *Peripheral*, and *Private Peripheral Bus (PPB)* regions. The firmware code and read-only data are placed in the ROM. The SRAM accommodates read-write data, including globals, stack, and heap. In the peripheral region, attached peripherals and controllers, such as GPIO and UART, are memory-mapped. The PPB contains system registers for controlling system configuration and monitoring system status, including the system timer, interrupt controller, and MPU.

3.2 Memory Protection Unit

The ARMv8-M MPU offers a memory access control feature for the address map. It provides several memory-mapped registers to define a fixed number of MPU regions (typically 8, 16, or 32) [3, 4]. Programmers can use these registers to configure the base address, size, validity, type, and access permissions (*e.g.*, Read, Write, and eXecute-Never) for each region.

3.3 Test Target Instruction

The ARMv8-M provides the TT instruction [5] to query the memory attributes for a given memory address. The primary purpose of the TT instruction is to query the access permissions for a given memory address. The format of assembly code for the TT instruction is {`TT Rd, [Rn]`}. The `Rd` is the destination register to store the result of TT instruction, and `Rn` is the source register to query the memory address. Specifically, the TT instruction generates a 32-bit response, with the lowest 8 bits indicating the MPU region number.

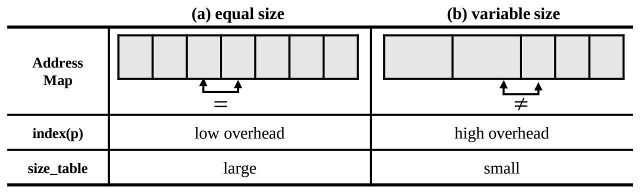

Figure 1: The comparison in adapting low-fat pointer to embedded systems.

4 Threat Model

We consider an embedded system equipped with an ARMv8-M processor. In micro-fat pointer, we assume that the attacker is aware of heap memory vulnerabilities within the embedded system, allowing them to potentially exploit it for software attacks, including data manipulation attacks. We presume that critical system resources, such as MPU's memory-mapped registers, are securely protected using privilege separation techniques [10, 17] which are orthogonal to and compatible with micro-fat pointer.

5 Motivation

In this section, we first describe how low-fat pointer [12] manages the bounds metadata. Then, we discuss design challenges when the low-fat pointer is applied to embedded systems.

In low-fat pointer, it defines *base* and *size* as bounds metadata. *base* denotes the base address of the memory object referred to by the pointer and *size* represents the allocation size of the referent object. Since they partition the virtual address space into several regions of equal size (*i.e.*, 4 GB), the upper 32 bits of 64-bit pointer indicate the region number, called the *index*. The following `index(p)` operation is for obtaining *index* for pointer `p`.

```
index(p) = p » 32
```

Then, by using *index*, they retrieve the assigned allocation size for the corresponding region from *size_table*, which is a data structure for storing the allocation size of each region. The following shows `size(p)` operation to retrieve the allocation size corresponding to `index(p)`.

```
size(p) = size_table[index(p)]
```

Finally, `base(p)` operation calculates the base address. Since all heap objects are allocation size-aligned, the base address can be obtained by just applying the combination of integer division and multiplication.

```
base(p) = (p / size(p)) * size(p)
```

It is noteworthy that, in low-fat pointer, the pointer itself contains *index* which can derive the allocation size, and it can be calculated by simply applying the right shift operation to the pointer. Also, for `size_table` in low-fat pointer, typically most entries correspond to the unused regions and they have the same value. For example, 48-bit address space requires 65536 entries. To reduce the memory overhead, low-fat pointer maps virtual pages holding those entries to the same physical page.

In this way, low-fat pointer effectively ensures spatial memory safety in a 64-bit machine that supports virtual memory. However, we found that low-fat pointer is not sufficiently effective in embedded systems. The challenges are examined through two illustrative cases.

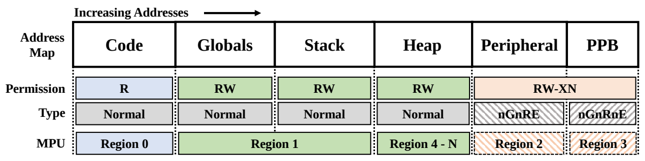

Figure 2: MPU Region Definition.

In the first case, we consider low-fat pointer in embedded systems with an equal region size (Figure 1 (a)). The main challenge is the limited SRAM, which restricts region size. For example, in a 256 KB SRAM system with a 4 GB memory map, considering segments like globals, heap, and stack in SRAM, a realistic approach is to partition memory into regions of 50 KB. However, this requires about 83,886 size_table entries. With each entry being 4 bytes, the total size_table would need around 328 KB of memory. Additionally, without virtual memory support, storing size_table entries for unused regions in the same physical memory becomes impossible. As a result, it leads to a higher memory overhead. Nevertheless, the performance impact remains comparatively low, as the `index(p)` operation is still adoptable.

The second case explores an alternative where memory regions in embedded systems have variable sizes, as shown in Figure 1 (b). This case helps in reducing the number of entries required in the size_table. However, it results in increased performance overhead. The problem is that the `index(p)` operation of low-fat pointer cannot be used because the regions are not aligned with equal sizes, making a simple shift operation practically impossible to determine the index. Designing conditional statements to distinguish between regions of different sizes is necessary. Also, defining different operations tailored to each region size is also required. However, these measures may result in additional performance overhead.

6 Design

In this section, we describe the design of micro-fat pointer. As mentioned before, micro-fat pointer defines MPU regions (in § 6.1). Based on this definition, micro-fat pointer queries the MPU region number and efficiently retrieves the index using the TT instruction. micro-fat pointer then utilizes this index to calculate bounds metadata. In micro-fat pointer, bounds metadata are defined as *base* and *size* for bounds checks. To facilitate these bounds checks, micro-fat pointer calculates bounds metadata and propagates the *base* metadata (in § 6.2). Also, bounds checks are instrumented for pointer operations (in § 6.3).

6.1 Memory Management

MPU Region Definition. micro-fat pointer defines MPU regions for heap memory safety as illustrated in Figure 2. It considers different memory access permissions and types for each memory segment. The code segment, with `read` permission, is assigned its own MPU region. In contrast, globals and stack segments, both having `read/write` permissions and identical memory types, are assigned to one MPU region. Peripherals and PPB are assigned individual MPU regions, despite having the same memory access permissions and memory types, due to distinct memory attributes such as nGnRE or nGnRnE. The remaining MPU regions are allocated to the heap segment, allowing it to be divided into multiple heap regions.

The size_map Configuration. The *size* metadata is defined in a table called `size_map`. We define the `size_map` as follows:

$$\texttt{size\_map[] = \{size}_0\texttt{, size}_1\texttt{, ... , size}_N\texttt{\}}$$

In micro-fat pointer, the *size* metadata is categorized into two types: *segment size* and *object size*. The segment size refers to the total size of the corresponding MPU region, while the object size indicates the size of the object to be allocated within that MPU region. Essentially, each element in the `size_map` is assigned either the segment size or the object size. Therefore, accessing the `size_map` with a non-heap region number retrieves the segment size, and with a heap region number retrieves the object size. Also, for size_1, an appropriate size is used to distinguish between globals and stack, instead of using the segment size. This is because both are allocated within the same MPU region, specifically region #1, as illustrated in Figure 2.

Micro-Fat Heap Allocator. A micro-fat heap allocator manages heap memory allocation in micro-fat pointer. As previously described, the heap segment in micro-fat pointer is divided into multiple heap regions. Contrasting with the conventional allocators that rely on a single, contiguous heap segment, the micro-fat heap allocator allocates objects across these several distinct heap regions. Notably, the allocated object size is consistently the same fixed within each region based on size_map. This ensures that the objects within each heap region are consistently aligned with the same size. When the micro-fat heap allocator receives a dynamic memory allocation request (*e.g.*, `malloc`), it adjusts the allocation size of the pointer to match the nearest object size defined for the heap region and allocates memory within the corresponding region. As a result, each heap object is allocated with the same allocation size within each region. This ensures that all base addresses within each region are aligned with the object size.

6.2 Bounds Metadata Management

The bounds metadata is calculated and propagated for pointer `p`. The calculation of the bounds metadata is necessary because it is directly used in bounds checks for pointer `p`. Furthermore, among the bounds metadata, the base is propagated because pointer `p` can alter its value due to various contexts, such as pointer arithmetic operations. In such cases, if the base address is calculated only within the bounds checks, it may no longer accurately represent the base address of pointer `p` when actual bounds checks are executed. This can result in incomplete bounds checking.

Metadata Calculation. Here, we define the operations for calculating the bounds metadata. First, the `index(p)` operation assumes the memory layout through the MPU region definition as described in Figure 2, and calculates the index as follows:

```
index(p) = TT(p) & 0x000000FF
```

The `TT(p)` operation involves assigning pointer `p` to the source register of the TT instruction, resulting in a 32-bit response from the TT instruction. The MPU region number, serving as the index, is obtained by performing a bitwise AND operation on the lowest 8 bits of this response. The `index(p)` operation differs significantly in the computation of the index from low-fat pointer which is described in § 5.

Next, the `size(p)` operation refers to the `size_map`, as defined in § 6.1, to retrieve the *size* metadata:

```
size = size_map[index]
```

Here, a `base(p)` operation calculates the *base*. As described in § 6.1, each heap object is allocated across heap regions, with all heap objects aligned according to their allocation size. This alignment is leveraged to compute the *base* using standard C 32-bit unsigned integer division and multiplication:

```
base = p / size * size
```

For non-heap objects, the *size* corresponds to the total size of their segment, and in these cases, the *base* is always the lower bound of that segment.

Metadata Propagation. micro-fat pointer propagates *base* metadata for bounds checking. Note that index and *size* are not explicitly propagated. The propagation of *base* diverges across the three primary scenarios: heap allocation, direct assignment, and pointer derivation. These three scenarios are based on the metadata propagation of low-fat pointer [12].

Heap allocation refers to a scenario where memory is dynamically allocated from the heap, such as when a pointer p is allocated from the heap. In this case, `p_base` is set to p:

```
p = malloc(size);
p_base = p;
```

Direct assignment refers to scenarios where a pointer's value is directly assigned without being derived from another pointer or allocated from the heap. The specific cases are, for instance, when the pointer is cast from an integer, loaded from memory, returned by a function, or used as a function argument. In these cases, the base pointer is explicitly determined using the *base(p)* operation:

```
p = (type *) i; or p = *r; or p = f();
  or type f(type *p) {};
p_base = base(p);
```

Pointer derivation refers to scenarios where a pointer is derived or transformed from an existing pointer, such as when the pointer is created through pointer arithmetic or when the pointer is cast from another pointer. In this case, the derived pointer inherits the base metadata of the parent pointer:

```
p = q + offset; or p = (type *)q;
p_base = q_base;
```

Notably, pointer arithmetic includes more than just basic addition and subtraction, covering operations like array and field access within data structures. Specifically, arithmetic operations involve tasks such as accessing elements in arrays and fields in structures. For example, accessing an element with `&ptr[index]` is effectively the same as `ptr+index`, and accessing a field with `&ptr->attribute` is similar to `ptr+offset`, where `offset` is the distance in bytes from the start of the object to the field `attribute`. Both array access and field access are instances of pointer arithmetic combined with a memory access operation.

6.3 Runtime Bounds Checking

In micro-fat pointer, the runtime bounds checks for pointer dereference operations and pointer arithmetic operations are inserted to prevent out-of-bounds memory access via compile-time transformation. Consequently, the modified program prevents OOB pointers from dereferencing or creating OOB pointers by pointer arithmetic through `check(p, base)` operation.

Bounds Checking. Here, we describe a `check(p, base)` operation. In this operation, the *base*, which is the *base* metadata propagated from the `base(p)` operation, is used as an argument. Also, similar to the `base(p)` operation, `check(p, base)` operation executes the `index(p)` and `size(p)`, and then compares the difference between the pointer value and the *base* against the *size*, using *unsigned integer underflow* [1, 12] to check the upper and lower bounds of an object through a single comparison, as follows:

```
(uintptr_t) p - (uintptr_t) base >= size
```

Pointer Dereference Instrumentation. micro-fat pointer instruments bounds checks before the pointer dereference operations such as `load` and `store` to prevent OOB pointer dereference:

```
if (check(p, p_base)) error();
var = *p; or *p = var;
```

Pointer Arithmetic Instrumentation. micro-fat pointer also instruments bounds checks to pointer arithmetic operations. Performing bounds checks on pointer arithmetic operations is essential for ensuring completeness, which is the ability to reliably detect all instances of OOB pointers regardless of pointer context transitions. In other words, an OOB pointer created through pointer arithmetic becomes problematic if it is subsequently cast to an integer, stored in memory, utilized as a function argument, or returned by a function:

```
p = q + offset;
if (check(p, q_base)) error();
i = (int)p; or *dst = p; or f(p); or return p;
```

The pointer arithmetic instrumentation to check only the OOB pointers passed between different contexts in micro-fat pointer is inspired by low-fat pointer [12].

6.4 Handling Non-Heap Objects

In the cases where it is unambiguous that pointers reference non-heap objects such as peripheral, globals, or the stack, micro-fat pointer does not instrument these pointer operations during compile-time transformation. This approach is straightforward as the distinction between heap and non-heap objects is evident, ensuring that such pointer operations are excluded from instrumentation.

In contrast, there are situations where it becomes ambiguous to determine which segment a pointer is referencing. This ambiguity is particularly prevalent in scenarios where pointers are passed as function parameters. Within the function, it becomes ambiguous whether these pointers point to heap or non-heap segments. This ambiguity necessitates that micro-fat pointer instruments these pointer operations. Therefore, when it is ambiguous whether pointers are referencing non-heap objects, micro-fat pointer invariably instruments on these pointer operations. Consequently, when conducting bounds checking (*i.e.*, `check(p, base)`) for pointers referencing non-heap objects, the size used is determined according to the segment size of the MPU region as defined in the `size_map`. This approach ensures that the pointer is considered safe as long as it stays within the boundaries of its MPU region. In other words, micro-fat pointer treats pointers as actual OOB when they exceed the boundary of their MPU region.

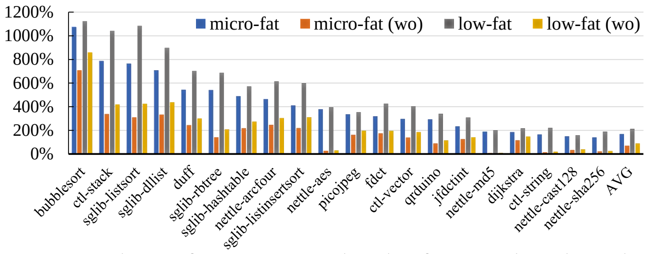

Figure 3: The performance overheads of BEEBS benchmarks

7 Evaluation

7.1 Implementation

micro-fat pointer consists of micro-fat transformations and a micro-fat runtime. The micro-fat transformations for bounds metadata management in § 6.2 and pointer instrumentation in § 6.3 are implemented in LLVM 15.0. We used the `-no-opaque-pointers` option to disable opaque pointers which are the default in LLVM 15.0 [24]. We used RIOT-OS version 2022.10 as an embedded operating system. The micro-fat runtime, which includes the micro-fat heap allocator and operations such as `base(p)` and `check(p, base)` in § 6, was implemented in RIOT-OS. For our implementation, we used the STMicroelectronics Nucleo-L552ZE-Q [34] as the target development board. This board is based on the ARMv8-M and supports eight MPU regions. Also, we configured the memory layout for the target development board through MPU configuration which is based on § 6. In addition, we partitioned the four heap regions equally with the same size of 7680 bytes. The object sizes of each heap region are 16, 64, 256, and 1024 bytes. The `size_map` is defined as follows:

```
uint32_t size_map[] = {536870912, 16384,  536870912,
                 1048576, 16, 64, 256, 1024};
```

7.2 Experimental Setup

To evaluate *micro-fat*, we compared our implementation with both a *baseline* and a *low-fat*. The baseline denotes the version where the micro-fat transformations and the micro-fat runtime are not applied. The low-fat denotes the implementation by referring to the low-fat pointer [12] and modifying it to be adapted to the embedded system as illustrated in Figure 1 (b). Note that, unlike micro-fat pointer, the low-fat does not use the MPU but implements the same size_map and maintains the same number of regions as micro-fat pointer implementation. We also evaluated a *write-only* version (wo), where instrumentation is applied solely to memory write operations. This approach is grounded in the understanding that most control and data flow attacks originate from memory writes [14]. Although this method results in incomplete bound checks, it significantly reduces the performance overhead. We used the BEEBs benchmark suite (version 2.1) [32] to evaluate performance overheads and code size overheads. We evaluated the performance overheads based on clock cycles using the DWT cycle counter. Also, we evaluated micro-fat pointer with application benchmarks [7] provided by RIOT-OS. We conducted 10 executions of each binary to compute the average clock cycles for every evaluation.

7.3 Performance Overheads

The BEEBS benchmark suite includes various benchmarks such as encryption, floating-point matrix multiplication, and sorting algorithms. This diversity made it suitable for evaluating both performance and code size overheads. We measured the performance of 62 benchmarks, selected from a total of 86, while excluding those with fewer than one million cycles or benchmarks that did not run properly. Among them, Figure 3 shows 20 benchmarks in order of greatest overhead.

As shown in Figure 3, for the micro-fat, the performance overheads compared to the baseline are 159.87% for the default version and 65.70% for the wo version. Similarly, for the low-fat, the overheads are 199.85% for the default version and 83.65% for the wo version, respectively. On average, the low-fat exhibited 41% higher overhead compared to the micro-fat. The benchmark with the largest performance difference between the micro-fat and the low-fat is the *sglib-listsort*, with a difference of 321%. According to our analysis, this is because *sglib-listsort* more frequently calls the `index(p)` than the other benchmarks. In specific, the `index(p)` of micro-fat operates with the TT instruction and a bitwise AND operation. In low-fat, however, the index cannot be obtained with shift instruction and requires a series of conditional statements and arithmetic instructions, as illustrated in Figure 1 (b).

7.4 Code Size Overheads

The code size overhead was also evaluated on the BEEBS benchmark. Compared to the baseline, the micro-fat experienced an average code size increase of 16.98%, with the wo version showing an 8.81% increase. For the low-fat implementation, there was an average code size increase of 15.65%, and the wo version exhibited a 7.24% increase. The increased code size in micro-fat is attributed to the additional code required for setting up the MPU, a step not necessary in low-fat.

7.5 Application Benchmarks

Table 1: The performance overheads of real-world examples

	psa_crypto	javascript	wasm	lua_basic
micro-fat	13.91%	24.39%	130.21%	161.01%
low-fat	23.43%	37.81%	155.74%	229.94%

We evaluate micro-fat pointer with real-world examples provided in the RIOT-OS. Specifically, we use *wasm*, *javascript*, *lua_basic*, and *psa_crypto* among them. Table 1 shows a performance overhead for micro-fat and low-fat.

Both *javascript* and *lua_basic* are script engines in common and mainly use non-heap objects (*i.e.*, global and stack objects). However, we confirmed that there is a significant difference in their performance overhead. The reason for this is that the ways of using non-heap objects were different in the two examples. In the *javascript*, the non-heap objects often are directly accessed by using the symbol name, so the `check(p, base)` and `base(p)` operations are not inserted to check them. On the other hand, in the *lua_basic*, the non-heap objects are often passed as function parameters. Then, in the callee function, the `check(p, base)` and `base(p)` operations are inevitably called due to the ambiguity of not knowing whether the object pointed to by the parameter is a heap object or a non-heap object. However, we believe that this performance overhead can be reduced if we improve the inter-procedure analysis of micro-fat pointer in the future work.

Unlike other examples, *wasm* failed to be executed with the current `size_map` configuration. Upon analyzing the cause, during the execution, most of allocation requests were between 16 and 256 bytes, but there is a single 13KB allocation request, which is larger than the maximum object size (i.e., 1024 bytes) in the `size_map`. So, for the evaluation, we adjusted the `size_map` settings to 16, 64, 256, and 13500 bytes for *wasm*.

8 Conclusion

In this paper, we introduce micro-fat pointer, a new heap memory safety solution carefully designed for embedded systems. Notably, micro-fat pointer employs an efficient bounds checking mechanism by leveraging the MPU and TT instruction provided by ARMv8-M. Our experimental results confirm that micro-fat pointer operates with much lower performance overhead than existing work. In future work, we plan to extend micro-fat pointer to a spatial memory safety solution, encompassing not only heap objects but also globals and stack objects.

Acknowledgments

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. RS-2023-00217689). Also, This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program(IITP-2024-RS-2023-00259967) supervised by the IITP(Institute for Information Communications Technology Planning Evaluation)

References

[1] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors.. In *USENIX Security Symposium*, Vol. 10. 96.

[2] Naif Almakhdhub, Abraham Anthony Clements, Saurabh Bagchi, and Mathias Payer. 2020. *uRAI: Return Address Integrity for Embedded Systems*. Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

[3] ARM. 2020. ARM Cortex-M23 Processor Technical Reference Manual. [Online]. Available: https://developer.arm.com/documentation/ddi0550/c.

[4] ARM. 2020. ARM Cortex-M33 Processor Technical Reference Manual. [Online]. Available: https://developer.arm.com/documentation/100230/0002.

[5] ARM. 2021. ARMv8-M Architecture Reference Manual. [Online]. Available: https://developer.arm.com/documentation/ddi0553/latest/.

[6] Todd M Austin, Scott E Breach, and Gurindar S Sohi. 1994. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation*. 290–301.

[7] Emmanuel Baccelli, Cenk Gündoğan, Oliver Hahm, Peter Kietzmann, Martine S. Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C. Schmidt, and Matthias Wählisch. 2018. RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT. *IEEE Internet of Things Journal* 5, 6 (March 2018). https://doi.org/10.1109/JIOT.2018.2815038

[8] Jeanelle Beeson. 2023. How Have Embedded Computers And The IoT Impacted Your Daily Life. https://robots.net/tech/how-have-embedded-computers-and-the-iot-impacted-your-daily-life/. Accessed: Nov 18, 2023.

[9] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. 2018. {ACES}: Automatic compartments for embedded systems. In *27th USENIX Security Symposium (USENIX Security 18)*. 65–82.

[10] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. 2017. Protecting bare-metal embedded systems with privilege overlays. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 289–303.

[11] Ang Cui, Michael Costello, and Salvatore Stolfo. 2013. When firmware modifications attack: A case study of embedded exploitation. (2013).

[12] Gregory J Duck and Roland HC Yap. 2016. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*. 132–142.

[13] Gregory J Duck and Roland HC Yap. 2018. An extended low fat allocator API and applications. *arXiv preprint arXiv:1804.04812* (2018).

[14] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers.. In *NDSS*, Vol. 17. 1–15.

[15] Arslan Khan, Dongyan Xu, and Dave Jing Tian. 2023. Ec: Embedded systems compartmentalization via intra-kernel isolation. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2990–3007.

[16] Arslan Khan, Dongyan Xu, and Dave Jing Tian. 2023. Low-cost privilege separation with compile time compartmentalization for embedded systems. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3008–3025.

[17] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. 2018. Securing Real-Time Microcontroller Systems through Customized Memory View Switching.. In *NDSS*.

[18] Yonghae Kim, Jaekyu Lee, and Hyesoon Kim. 2020. Hardware-based always-on heap memory safety. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1153–1166.

[19] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. 2018. Delta pointers: Buffer overflow checks without the checks. In *Proceedings of the Thirteenth EuroSys Conference*. 1–14.

[20] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. SGXBOUNDS: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*. 205–221.

[21] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 234–251.

[22] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. 2022. PACMem: Enforcing spatial and temporal memory safety via ARM pointer authentication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1901–1915.

[23] Lulu Liang, Kai Zheng, Qiankun Sheng, and Xin Huang. 2016. A denial of service attack method for an iot system. In *2016 8th international conference on Information Technology in Medicine and Education (ITME)*. IEEE, 360–364.

[24] LLVM. 2023. LLVM 15.x. https://github.com/llvm/llvm-project/tree/release/15.x. Accessed on: Nov 18, 2023.

[25] Zheyuan Ma, Xi Tan, Lukasz Ziarek, Ning Zhang, Hongxin Hu, and Ziming Zhao. 2023. Return-to-Non-Secure Vulnerabilities on ARM Cortex-M TrustZone: Attack and Defense. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[26] Alejandro Mera, Yi Hui Chen, Ruimin Sun, Engin Kirda, and Long Lu. 2022. D-Box: DMA-enabled Compartmentalization for Embedded Applications. *NDSS 2022* (2022).

[27] Daniele Midi, Mathias Payer, and Elisa Bertino. 2017. Memory safety for embedded devices with nesCheck. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 127–139.

[28] Alyssa Milburn, Erik Van Der Kouwe, and Cristiano Giuffrida. 2022. Mitigating information leakage vulnerabilities with type-based data isolation. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1049–1065.

[29] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 245–258.

[30] Thomas Nyman, Jan-Erik Ekberg, Lucas Davi, and N Asokan. 2017. CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers. In *Research in Attacks, Intrusions, and Defenses: 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18–20, 2017, Proceedings*. Springer, 259–284.

[31] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. Intel MPX explained: An empirical study of intel MPX and software-based bounds checking approaches. *arXiv preprint arXiv:1702.00719* (2017).

[32] James Pallister, Simon Hollis, and Jeremy Bennett. 2013. BEEBS: Open benchmarks for energy measurements on embedded platforms. *arXiv preprint arXiv:1308.5174* (2013).

[33] Jiwon Seo, Junseung You, Donghyun Kwon, Yeongpil Cho, and Yunheung Paek. 2023. ZOMETAG: Zone-based Memory Tagging for Fast, Deterministic Detection of Spatial Memory Violations on ARM. *IEEE Transactions on Information Forensics and Security* (2023).

[34] STMicroelectronics. 2022. STM32 NUCLEO-144 Development Board. https://www.st.com/en/evaluation-tools/nucleo-l552ze-q.html. Accessed on: Nov 18, 2023.

[35] Tong Zhang, Dongyoon Lee, and Changhee Jung. 2019. Bogo: Buy spatial memory safety, get temporal memory safety (almost) free. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 631–644.

[36] Jie Zhou, Yufei Du, Zhuojia Shen, Lele Ma, John Criswell, and Robert J Walls. 2020. Silhouette: Efficient protected shadow stacks for embedded systems. In *29th USENIX Security Symposium (USENIX Security 20)*. 1219–1236.

[37] Xia Zhou, Jiaqi Li, Wenlong Zhang, Yajin Zhou, Wenbo Shen, and Kui Ren. 2022. OPEC: operation-based security isolation for bare-metal embedded systems. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 317–333.