MESH: A Memory-Efficient Safe Heap for C/C++

Emanuel Q. Vintila
emanuel.vintila@gmail.com
Fraunhofer AISEC
Garching, near Munich, Germany

Philipp Zieris
philipp.zieris@aisec.fraunhofer.de
Fraunhofer AISEC
Garching, near Munich, Germany

Julian Horsch
julian.horsch@aisec.fraunhofer.de
Fraunhofer AISEC
Garching, near Munich, Germany

ABSTRACT

While memory corruption bugs stemming from the use of unsafe programming languages are an old and well-researched problem, the resulting vulnerabilities still dominate real-world exploitation today. Various mitigations have been proposed to alleviate the problem, mainly in the form of language dialects, static program analysis, and code or binary instrumentation. Solutions like Adress-Sanitizer (ASan) and Softbound/CETS have proven that the latter approach is very promising, being able to achieve memory safety without requiring manual source code adaptions, albeit suffering substantial performance and memory overheads. While performance overhead can be seen as a flexible constraint, extensive memory overheads can be prohibitive for the use of such solutions in memory-constrained environments. To address this problem, we propose MESH, a highly memory-efficient safe heap for C/C++. With its constant, very small memory overhead (configurable up to 2 MB on x86-64) and constant complexity for pointer access checking, MESH offers efficient, byte-precise spatial and temporal memory safety for memory-constrained scenarios. Without jeopardizing the security of safe heap objects, MESH is fully compatible with existing code and uninstrumented libraries, making it practical to use in heterogeneous environments. We show the feasibility of our approach with a full LLVM-based prototype supporting both major architectures, i.e., x86-64 and ARM64, in a Linux runtime environment. Our prototype evaluation shows that, compared to ASan and Softbound/CETS, MESH can achieve huge memory savings while preserving similar execution performance.

KEYWORDS

memory safety, unsafe programming languages, buffer overflows, pointer tagging, dangling pointers, use-after-free

1 INTRODUCTION

The C and C++ programming languages are preferred for system programming because they are efficient and offer low-level control. These advantages come at the cost of memory safety, requiring the programmer to manually manage heap memory and ensure the legality of memory accesses. Incorrect handling by the programmer can lead to serious issues [43] such as *buffer overflows* or *use-after-free* bugs, which can be exploited by attackers to hijack the control-flow (causing the program to inadvertently change its course of execution) [6, 11, 35, 39] or to leak information [18, 40]. Even though memory vulnerabilities are long-known issues, they are still very common in mainstream programs: Out of all the vulnerabilities reported in the Common Vulnerabilities and Exposures (CVE) database, 14.7% are marked as overflows and 4.3% are marked as memory corruptions [13]. C/C++ memory corruption bugs can be divided into two categories:

Temporal memory bugs access an object outside of its lifetime, i.e., before its *allocation* or after its *deallocation.*

Spatial memory bugs access an object outside of its bounds, i.e., using addresses *lower than the start* or *higher than the end* of the object.

Several mechanisms have been proposed to mitigate the effects of memory bugs. A widely used mitigation mechanism are stack canaries [14], detecting overflows on the stack before they can affect the control flow via the stack-based return address. Another mitigation mechanism is Address Space Layout Randomization (ASLR) [24], randomizing the location of program parts, making it harder for the attacker to correctly guess valid code addresses. Finally, in recent years, a plethora of Control-Flow Integrity (CFI) approaches have been proposed [7, 9], trying to protect indirect jumps, calls, and function returns using various policies and techniques. Unfortunately, trading security for performance typically leaves mitigation techniques vulnerable to exploitation, as shown in various attacks on CFI [10, 12, 17] and ASLR [4, 33, 40].

A more complete way to counter memory vulnerabilities is to detect and mitigate the memory bugs themselves—and not just their effects—by imposing *memory safety*. Several ways of enforcing memory safety in C/C++ have been proposed, among which are *language dialects*, *static analysis*, and, more relevant to our work, *code or binary instrumentation*. Language dialects remove unsafe features from the languages, offering alternatives, for example, in the form of new pointers types [2, 21, 30]. While they can be effective and efficient (fewer run-time checks), dialects of C/C++ lack compatibility with existing code and are less likely to be accepted in practice. Static analysis techniques [3, 20] detect bugs without running the code, and hence do not impose any run-time overheads, but are less effective. Finally, code or binary instrumentation can be used for detecting memory bugs through dynamic analysis

[25, 41, 42]. They work on existing code and allow the programmer to write new code as usual, without any knowledge about the underlying instrumentation.

The most prominent instrumentation-based memory safety mechanism is AddressSanitizer (ASan) [36]. ASan approximates memory safety by inserting protected memory regions between objects and delaying reuse of freed memory. A more precise solution is offered by SoftBound/CETS [28, 29], which ties pointers to their objects and achieves strong temporal and spatial memory safety. Both approaches, as well as other similar solutions [8, 16, 37], impose substantial run-time and memory overheads. While a large performance overhead might be impractical, it presents a flexible constraint and does not prevent a solution from being applicable at all. In contrast, extensive memory overheads can be prohibitive for applying a solution in memory-constrained devices. Furthermore, most of the approaches [2, 8, 16, 21, 28–30] do not offer a concept for (secure) compatibility with uninstrumented code, hindering their real-world applicability, e.g., in situations where external libraries cannot be instrumented.

To tackle those problems, we present MESH, a novel, highly memory-efficient heap safety mechanism, which uses a configurable but constant-sized lookup table, the *MESH table*, for storing bounds and the validity of objects in memory. MESH makes use of the unused bits in a pointer (by default, on Linux, only 47 bits of a pointer are used on x86-64[1], and 48 on ARM64[2]) to link pointers to their objects' metadata entries in the MESH table. Using a constant lookup algorithm, MESH ensures that only legal pointer uses are permitted at run-time. Since stack corruptions have become less important in recent years while heap corruptions are still very common and often critical [27], MESH focuses on heap safety. Nonetheless, MESH offers not only memory safety for heap objects, but also protects its safe heap against other accesses, e.g., using stack-based or external pointers.

Compared to other approaches of similar preciseness [2, 8, 21, 29, 30], MESH's constant maximum memory overhead of only 2 MB on systems with 17 unused pointer bits (2^{17} table entries with 16 bytes each) is almost negligible. MESH is fully compatible with uninstrumented code, enabling the safe use and creation of external pointers in MESH-instrumented code without manual code modifications.

In summary, we make the following contributions:

- We propose MESH, a memory-efficient safe heap using code instrumentation for spatial and temporal memory safety.
- We present an LLVM-based prototype implementation protecting the heap of C/C++ applications in a Linux runtime environment for the x86-64 and ARM64 architectures.[3]
- We show the feasibility of our approach in a detailed evaluation, measuring and comparing MESH's performance and memory overhead to existing solutions.

The remainder of the paper is organized as follows. In Section 2, we define design goals for MESH. Section 3 details MESH's actual design based on our design goals. We describe our prototype in Section 4 before discussing evaluation results in Section 5. Finally, we examine related work in Section 6 and conclude in Section 7.

---

[1]https://www.kernel.org/doc/html/v5.8/x86/x86_64/mm.html
[2]https://www.kernel.org/doc/html/v5.8/arm64/memory.html
[3]Source code available under: https://github.com/Fraunhofer-AISEC/mesh

2 DESIGN GOALS

For the design of MESH, we assume an attacker who can perform memory corruption attacks on objects in arbitrary data regions of a program. Further, we assume that the attacker cannot corrupt the code itself (typically guaranteed by non-writable text segments) or modify the program's data regions from outside the instrumented code, e.g., using special hardware access.

Based on these assumptions, we define six design goals for MESH to ensure heap memory safety for C/C++ while maintaining high compatibility and memory efficiency. Our first three design goals aim to *ensure memory safety* for the MESH heap:

❶ **Detect temporal bugs:** Detect the usage of heap pointers to heap objects that are no longer allocated (i.e., dangling pointers).

❷ **Detect spatial bugs:** Detect any access (read/write) using a heap pointer pointing outside the allocation bounds of its object (i.e., overflows or underflows) with byte precision.

❸ **Detect unprotected pointer accesses:** Detect accesses to protected objects using unprotected pointers (e.g., stack or external pointers).

The next design goals target MESH's *compatibility*. Our defense mechanism must not change the functionality of the software it protects and developers should not have to be aware of the underlying protection. In other words, MESH has to work on any existing C/C++ code without requiring modifications. We formalize these goals as follows:

❹ **Support standard C/C++:** Every language feature in the C/C++ standards must be supported. Moreover, language features should not be limited or extended.

❺ **Support unprotected code:** Code that is instrumented must be compatible with uninstrumented code, i.e., calling external functions or using pointers created in external code must not break the functionality of the program.

Finally, the last design goal aims to enable MESH's *application in memory-constrained environments*:

❻ **Impose virtually no memory overhead:** The memory overhead imposed on the protected software must be constant and insignificant in the software's overall memory consumption.

3 MESH DESIGN

In C and C++, pointers are simple types that hold addresses without any information about the memory object pointed to. Hence, programming errors involving pointers can easily lead to violations of temporal or spatial memory safety. To ensure memory safety, pointers must be augmented with additional *metadata* that can be used to check if an access is legal at run-time. MESH keeps its metadata in a table disjoint from pointers and objects, and uses pointer tagging for linking pointers to metadata.

In the following, we first introduce the MESH metadata structure. Then, we discuss the management and placement of our safe heap. Finally, we discuss how the MESH heap achieves our goals in terms of memory safety and compatibility.

3.1 Objects and Metadata

To discuss details of MESH's design, we first identify all operations on objects that must be considered when implementing memory safety. The life-cycle of heap objects in C/C++ has three phases:

Allocation. The allocation of objects in the heap is done by calling the corresponding functions (e.g., *malloc* or *new*). The allocator then reserves enough bytes of memory for the object and returns a pointer to the start of the object.

Access. Read or write accesses must happen during this phase. Accesses are only legal on allocated objects and must be within object bounds. Otherwise, they present spatial memory bugs, e.g., *buffer overflows and underflows*.

Deallocation. The deallocation of heap objects is done manually using the respective allocator functions (e.g., *free* or *delete*). Only one deallocation per allocated object is allowed. Deallocating an object more than once, a *double-free* bug, or accessing a deallocated object, a *use-after-free* bug, can directly lead to memory corruptions or information leaks.

3.1.1 MESH Metadata Structure. To ensure memory safety, MESH requires metadata to describe the spatial and temporal characteristics of objects. This metadata can be represented as follows:

$$m_{obj} := \Big((lower\_bound_{obj}, upper\_bound_{obj}), validity\_flag_{obj} \Big)$$

For spatial memory safety, all pointer *accesses* must be within the bounds of the object they are pointing to. Memory associated with objects is contiguous, starting from a lower bound and ending at an upper bound. These bounds are part of the object metadata and are used for checking each pointer access. For ensuring temporal safety, all pointer *accesses* must happen on objects that have been allocated, but not yet deallocated. Therefore, the metadata must contain information about the temporal validity of the object. For this purpose, a metadata entry contains a validity flag, which is set during allocation and cleared as soon as the object is deallocated.

MESH stores all metadata disjoint from pointers and objects in the MESH table, as illustrated in Figure 1. Each row in the table uniquely corresponds to one object obj containing its metadata m_{obj}. The rows are indexed using the global MESH table index I. Figure 1 additionally shows a pointer c to object obj at address N. During the *allocation* of obj, a new entry is generated at the current index I. The entry contains the object's base address N as the $lower\_bound_{obj}$ and the result from adding the size of obj to its base address as the $upper\_bound_{obj}$. The $validity\_flag_{obj}$ is also set and cleared once the object is *deallocated.*

3.1.2 MESH Metadata Access. MESH uses unused pointer bits to store an index into the MESH table, linking a pointer to its corresponding object's metadata entry. This is illustrated in Figure 1, showing a system in which T pointer bits are unused. By storing the index I in those unused pointer bits, MESH links the pointer c to its object obj, represented by entry I in the MESH table. T also determines the size of the MESH table (i.e., 2^T possible entries), limiting the number of objects that can be protected simultaneously.

In a sequence of allocations, I always contains the MESH table index used for the next object. At program start, I is initialized to the maximum value of $2^T - 1$ and the MESH table is initialized to contain only invalid entries. During program execution, I is
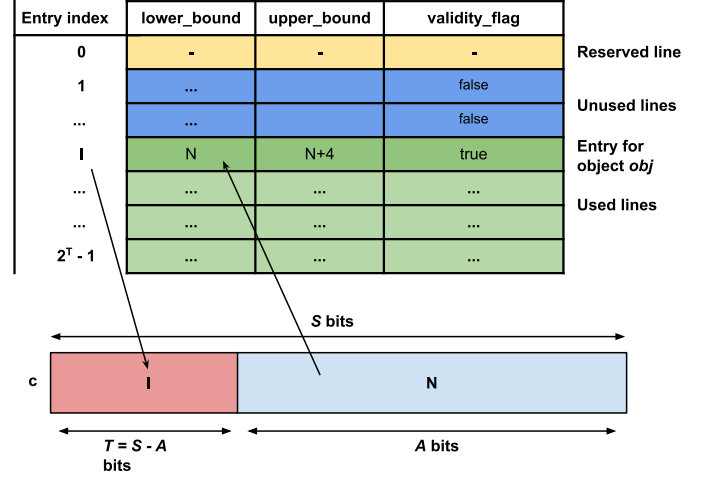

Figure 1: MESH table after a 4-byte allocation at address N, and its resulting tagged pointer

decremented after each allocation. If I reaches its minimum value of 1, MESH will either wrap around I and start re-using table slots freed in the meantime or will terminate on the next allocation. This is discussed further in Section 3.5.

3.2 Heap Separation

For MESH to effectively protect its heap objects, we must not only guarantee their memory safety by maintaining the MESH table and tagging pointers, but also secure access to those objects and the MESH table from unprotected, i.e., *untagged*, pointers. Unprotected pointers might origin from within the program itself, i.e., from global, stack, or memory-mapped memory, or from a call to an external function that returns a pointer to an object allocated in the unprotected heap or memory-mapped memory of the program.

To achieve this protection, we separate the MESH table and all MESH-protected heap objects from other data segments in a dedicated memory segment, the *safe heap*. The safe heap is managed by its own allocator, enabling programs to allocate protected objects with tagged pointers on the safe heap, while simultaneously supporting the allocation of unprotected objects using the conventional allocator and generating untagged, i.e., unmodified, pointers. To protect safe heap objects from accesses through untagged pointers, we prevent any access to the safe heap using an untagged pointer. As a consequence, the MESH index $I = 0$ is reserved for untagged pointers and cannot be used to store MESH metadata.

3.3 Memory Safety Enforcement

To show how MESH enforces memory safety, we examine possible operations on pointers. A pointer is generated whenever a new heap object is allocated and its memory address is *assigned* to the pointer. Additionally, a pointer may also be *derived* from another, already assigned pointer. During the object's life span, the pointer can be *compared* to other pointers or *dereferenced* to access the underlying object. Once the object is deallocated, the pointer is *invalidated* and no further use should be permitted. When a pointer is derived

---

Algorithm 1 Metadata generation during allocation

1: **globals**
2:     *Table*: MESH table
3:     *I*: MESH table index
4: **procedure** MESH_MALLOC(*size*)
5:     *address* ← **safe_malloc**(*size*) ▷ Allocate safe heap object
6:     **if** *pointer* = *NULL* **then**
7:         **return** *NULL*
8:     **if** *I* = 0 **then**   ▷ Check for end of MESH table
9:         **fail**("Metadata exhaustion")
10:     *i* ← *I*   ▷ Derive new index
11:     *I* ← *I* − 1   ▷ Decrement index
12:     *Table*[*i*].*lower_bound* ← *address* ▷ Set spatial metadata
13:     *Table*[*i*].*upper_bound* ← *address* + *size*
14:     *Table*[*i*].*validity_flag* ← *true* ▷ Set temporal metadata
15:     *address* ← *i* | *address*   ▷ Tag pointer
16:     **return** *address*

---

from another pointer, an implicit invalidation of the overwritten pointer occurs. To achieve our design goals from Section 2, the MESH design must consider all five pointer operations.

3.3.1 Assignment. With MESH, objects can be allocated in one of two coexisting memory areas: the MESH-protected safe heap and the remaining memory, including allocations on the stack, the normal heap, or in memory-mapped memory. In general, MESH automatically redirects all heap allocations inside the program to the safe heap, while other allocations remain unaffected. In particular, the latter includes allocations in external, uninstrumented code, such as linked libraries, which are unchanged by MESH, and still reserve memory in the normal heap and return untagged pointers. Using this separation, MESH does not interfere with any C/C++ feature nor restricts compatibility with uninstrumented code, fulfilling our design goals ❹ and ❺.

For allocations on the safe heap, MESH must generate a new metadata entry and tag the new pointer with its MESH table index. For this, we wrap the safe heap allocator (safe_malloc) within a dedicated metadata generation routine, as shown in Algorithm 1. The routine takes the allocation size as input and first tries to allocate the object through the safe heap allocator. If the allocation fails, the routine aborts and returns NULL letting the application handle the failed allocation. Next, the routine derives the table index *i* for the newly allocated object from the global MESH table index *I*. The operations on *I* are arranged in a way that the allocation algorithm can be made thread-safe as discussed in Section 4.4. Finally, the routine initializes the bounds and validity metadata, and returns the tagged address as the final pointer to the object.

3.3.2 Derivation. Pointers can be derived from other pointers by copying or performing pointer arithmetic. Pointers can also be cast to different types, changing the way they are dereferenced. Since the derived pointer must retain the same restrictions as the pointer it derived from, MESH must propagate metadata from source to destination pointers.

As MESH uses tagged pointers, metadata is an implicit part of the value used during the pointer derivation, trivially achieving

---

Algorithm 2 Temporal safety check

1: **globals**
2:     *Table*: MESH table
3: **procedure** TEMPORAL_SAFETY_CHECK(*pointer*)
4:     *tag* | *address* ← *pointer*   ▷ Split pointer
5:     **if** *tag* = 0 **then**   ▷ No check for untagged pointers
6:         **return**
    ▷ Perform temporal check
7:     **if** *Table*[*tag*].*validity_flag* = *false* **then**
8:         **fail**("use-after-free detected")

---

design goal ❹ without instrumentation. For copying and casting, the tag is implicitly copied with the pointer value. Similarly, pointer arithmetic directly performs calculations on the tagged pointer and generates a new pointer containing the same tag. As desired, untagged pointers remain untagged through pointer derivation. During pointer arithmetic, address values might theoretically overflow into the MESH tag bits, potentially corrupting an existing tag or creating a tag in an untagged pointer. It is very unlikely that the result of such overflows is usable, since the pointer value must match the bounds associated with the corrupted or crafted tag.

3.3.3 Comparison. In C and C++, pointers of the same object can be compared for equality or relative ordering. With MESH, pointers of the same object share the same tag, leaving comparisons for equality or relative ordering intact without instrumentation. C and C++ do not define comparing pointers of different objects and comparing pointers with different tags in MESH might yield different results than comparing untagged pointers without MESH.

3.3.4 Dereferencing. An object is accessed by dereferencing the corresponding pointer. For safe heap pointers, accesses must only be allowed within the bounds of the object and during the life span of the object, i.e., while it is still allocated. Both constraints are enforced in MESH by calling the temporal and spatial safety check routines shown in Algorithms 2 and 3 before dereferencing a pointer. Both routines first separate the tag from the checked pointer and then use the metadata from the MESH table entry indexed by the tag to verify the legality of the access. A temporal memory safety violation is detected if the pointer's object has been deallocated, indicated by an unset validity bit. A spatial violation is detected if the access is outside the object's bounds. The spatial check takes the pointer type's size into account, also detecting overflows that occur due to casting pointers to larger sizes. The MESH table is initialized with unset validity flags and invalid bounds, so that tag bits indexing an unused metadata entry will always lead to a violation. Pointers not derived from an allocation in the safe heap should never be allowed to access the safe heap. Hence, for those untagged pointers, the spatial safety check must additionally ensure that access to the safe heap memory is prohibited.

After the temporal and spatial validity of a pointer has been verified, the pointer can be dereferenced. However, because MESH uses tagged pointers, we cannot directly take the pointer's value to access the underlying memory. Hence, MESH *strips* pointers, i.e., removes their tag bits, before dereferencing.

Algorithm 3 Spatial safety check

1: **globals**
2:     $Table$: MESH table
3: **procedure** SPATIAL_SAFETY_CHECK($pointer$)
4:     $tag \mid address \leftarrow pointer$ ▷ Split pointer
5:     **if** $tag = 0$ **then**
6:         **if** $address \in$ safe heap **then**
7:             **fail**("Illegal access to safe heap detected")
8:         **else**
9:             **return**
        ▷ Perform spatial checks
10:     **if** $address < Table[tag].lower\_bound$ **then**
11:         **fail**("Buffer underflow detected")
12:     $size \leftarrow$ **sizeof**(**typeof**($pointer$))
13:     **if** ($address + size) \geq Table[tag].upper\_bound$ **then**
14:         **fail**("Buffer overflow detected")

In summary, by calling both the temporal and spatial safety check routines before dereferencing a pointer, MESH fulfills the design goals ❶, ❷, and ❸. In addition, by stripping the pointer after validation but before dereferencing, design goal ❹ is also met.

3.3.5 Invalidation. At the end of their lifetime, safe heap objects are deallocated using our dedicated allocator, while objects on the normal heap are freed as regular. To prohibit any further access to a deallocated object in the safe heap and to achieve goal ❶, MESH must invalidate the corresponding metadata entry. Additionally, MESH must ensure that normal heap objects can still be freed using untagged pointers and, to achieve goal ❸, that this does not affect the safe heap. Similarly to safe heap allocations, deallocations are realized by wrapping the safe heap deallocator (`safe_free`) within a metadata invalidation routine.

The metadata invalidation routine, shown in Algorithm 4, takes the pointer to be freed as input. Typically, heap objects allocated by external, uninstrumented code are also deallocated externally. Nonetheless, since this is not always the case, the routine must be able to deallocate normal heap objects with untagged pointers. If the given pointer is untagged, the corresponding object is freed using the system-provided heap deallocation function, after checking that the pointer does not point into the safe heap region. For tagged pointers, the routine performs a temporal safety check before invalidating and freeing the object.

3.4 Compatibility

As discussed in the previous section, MESH's pointer tagging naturally interferes with C/C++ language assumptions. However, because MESH strips pointers before dereferencing or calling any deallocation functions, design goal ❹ is still fulfilled. For implementing MESH, the only essential requirement are unused bits in pointers to store the MESH table index I. MESH does not make any other assumptions about the underlying platform implementation nor modifies it in any way. In particular, while MESH moves heap allocations into the safe heap, it does not change the behavior or semantics of a program's normal heap and its memory management. Further, pointers are not "fattened" by MESH using pointer structures and pointers to globals, stack objects, or other unprotected

---

Algorithm 4 Metadata invalidation during deallocation

1: **globals**
2:     $Table$: MESH table
3: **procedure** MESH_FREE($pointer$)
4:     $tag \mid address \leftarrow pointer$ ▷ Split pointer
5:     **if** $tag = 0$ **then**
6:         **if** $address \in$ safe heap **then**
7:             **fail**("Illegal free in safe heap detected")
8:         **else**
9:             **free**($address$)   ▷ Deallocate normal heap object
10:             **return**
        ▷ Perform temporal check
11:     **if** $Table[tag].validity\_flag = false$ **then**
12:         **fail**("double-free detected")
13:     $Table[tag].lower\_bound \leftarrow 0$ ▷ Invalidate entry
14:     $Table[tag].upper\_bound \leftarrow 0$
15:     $Table[tag].validity\_flag \leftarrow false$
16:     **safe_free**($address$)   ▷ Deallocate safe heap object

---

memory are left untouched. Moreover, the most notable sources of language incompatibility for memory safety mechanisms are modifications to function call conventions, casts between different pointer types, or casts between pointers and integers. MESH handles these cases by encoding the index to the metadata into the value of the pointer itself: Since pointers are passed to functions by value, their tags are passed implicitly. Likewise, casts are done by value, only having their interpretation changed.

For compatibility with external code, we have to consider untagged pointers that originate from uninstrumented code, and tagged pointers that are passed to uninstrumented code. Since the normal heap is still available alongside the safe heap, uninstrumented code can still use the system-provided heap management functions, allocating and deallocating objects with untagged pointers. If those untagged pointers are returned to MESH-instrumented code, MESH is able to use them as-is without breaking any functionality and without compromising the security of the safe heap, as those untagged pointers can never access the safe heap.

When passing tagged pointers as arguments to external functions, the pointers are checked and stripped by MESH before issuing the function call, as they are unusable for uninstrumented code otherwise. For functions returning one of its pointer arguments, MESH additionally wraps the call to store the stripped tag temporarily and reapply it to the returned pointer argument after the function returned. To identify functions external to MESH-instrumented code, we perform instrumentation during Link Time Optimization (LTO). LTO generates an inter-modular representation of the entire application, enabling our instrumentation to detect truly external functions that will not be instrumented. As a result, MESH achieves complete compatibility with any external, uninstrumented library, and therefore meets design goal ❺.

3.5 MESH Limitations

The limitations of MESH result from the number of unused bits in a pointer. As those bits store the MESH table index I, they directly

determine the maximum number of objects that can be represented in the MESH table.

3.5.1 Heap-Only Protection. In general, we designed MESH to avoid the complete filling of its table. As a consequence, we are bound to protecting a maximum of $2^T - 1$ objects, when T pointer bits are unused. In practice, as shown in Section 5.3, restricting the MESH protection to the heap alleviates this problem, as there are generally much fewer, but more long-lived heap objects than stack objects in typical programs.

However, limiting the MESH protection to the heap only, leaves stack objects unprotected and stack pointers untagged. An unprotected stack does not pose a direct security threat to MESH, as untagged pointers are prohibited from accessing the safe heap. But, with a stack vulnerability present, an attacker might be able to craft a tagged pointer to access the safe heap regardless. To circumvent this possibility, it is possible to combine MESH with a separate stack protection mechanism, such as SafeStack [23]. With SafeStack, potentially exploitable stack objects are moved to MESH's safe heap, while provable safe objects remain on the stack. Consequently, stack vulnerabilities no longer pose a security threat as well.

3.5.2 Architecture Support. To reduce the limiting effects of unused pointer bits available for storing the MESH table index I, MESH is best supported on 64-bit architectures. On our target architectures x86-64 and ARM64, MESH natively supports storage of at least $2^{17}-1$ and $2^{16}-1$ metadata entries, respectively. Most applications—especially in a memory-constrained environment—only use up to a few thousand heap objects per process (as shown in more detail in Section 5.3). Moreover, by modifying the kernel's virtual address space size (i.e., decreasing the number of used address bits in a pointer) or by restricting the heap memory space on an application level, the bits available for storing I can be increased.

3.5.3 MESH Table Wrap-Around. If more than the supported $2^T - 1$ allocations are required by an application, once the MESH table index I reaches its minimum value of 1, MESH can optionally perform a wrap-around of I and reuse entries of objects freed in the meantime. If enabled, this wrap-around is an extension to MESH's metadata generation routine (see Algorithm 1) that searches for entries in the MESH table whose validity flag is unset. Such an entry is then reused by the routine to store the metadata of the newly allocated object. While this wrap-around avoids the limitation on the total number of allocations, MESH is still limited to support at most $2^T - 1$ alive safe heap objects at the same time. The limit on simultaneously alive objects—with or without the optional wrap-around—is similar in nature to other common resource exhaustion problems, such as stack overflows or heap exhaustion. However, the optional wrap-around degrades MESH's temporal safety guarantees from precise to probabilistic: A *dangling* pointer to an already deallocated object might get usable again when the metadata entry corresponding to its tag gets reused due to the wrap-around mechanism. Nevertheless, for rogue memory accesses to succeed with such dangling pointers, the bounds of the new object have to overlap with the bounds of the past object, as otherwise a spatial safety violation is detected. Considering this, we can assume that most memory safety violations are still detected and MESH can,

with its wrap-around mechanism, offer an attractive trade-off for larger applications.

4 IMPLEMENTATION

To show the feasibility of MESH, we implemented a prototype for the x86-64 and ARM64 architectures as an extension to the LLVM compiler framework (version 11). Our prototype currently supports the ELF binary format for Linux runtime environments.

As discussed in Section 3.3, MESH has to handle pointer assignments, dereferences, and invalidations to enforce the memory safety of heap objects. Our prototype provides a dedicated runtime support library for assigning and invalidating pointers, and directly instruments *load* and *store* instructions in LLVM's Intermediate Representation (IR) with the temporal and spatial safety checks for pointer dereferences. Additionally, our prototype optimizes performance by removing checks statically proven to be safe and ensures that all changes to the program are thread-safe.

4.1 Runtime Support Library

We built our prototype's runtime support based on the LLVM compiler runtime (compiler-rt) library. The runtime support initializes and manages the safe heap, maintains the MESH table, and handles memory safety violations.

The runtime support adds a constructor to protected binaries, which is executed by the dynamic linker at load-time. This constructor first allocates 1/8 of available memory for the safe heap using mmap. Within the safe heap, it then allocates the MESH table and initializes the entire table to invalid memory addresses (i.e., the value 0xFF...FF for both columns *lower_bound* and *upper_bound*). As discussed in Section 3.1, the MESH table has a constant size of 2^T entries, depending on the architecture and its number of unused bits T in a pointer. The constructor initializes the MESH table index I to the table's uppermost entry, i.e., $I = 2^T - 1$. The index is also allocated in the safe heap, making it solely accessible to our metadata handling routines. The MESH table and its index are independent for each process so that there are no inter-process conflicts.

The runtime support also provides our dedicated safe heap allocator and routines wrapping the allocator functions to handle MESH metadata. The safe heap allocator is based on a simple heap allocator implementation[4] modified by us to be thread-safe and support 64-bit systems. The implementation of the metadata routines, as described in Section 3.3, straightforwardly follows Algorithm 1 for allocating and Algorithm 4 for deallocating. The MESH instrumentation detects calls to the GNU C and C++ standard library's allocation and deallocation functions and redirects them to our routines.[5] In addition, if enabled through a compiler flag, the allocation routines also perform the optional MESH table wrap-around, as detailed in Section 3.5.

4.2 IR Instrumentation

To enforce temporal and spatial memory safety, our prototype has to instrument each load and store instruction. Because loads and stores are very frequent, we use an optimized check that combines

---

[4]https://github.com/CCareaga/heap_allocator
[5]MESH provides custom routines for malloc, free, new, delete, calloc, realloc, and memalign.

Algorithm 5 Spatial and temporal safety check

1: **globals**
2:     *Table*: MESH table
3: **procedure** SAFETY_CHECK(*pointer*)
4:     *tag* | *address* ← *pointer*   ▷ Split pointer
5:     **if** *tag* = 0 **then**
           ▷ Optimized access to safe heap check
6:         **if** *address* ≤ safe heap upper bound **then**
7:             **fail**("Illegal access to safe heap detected")
8:         **else**
9:             **return**
    ▷ Combined temporal and lower bound spatial check
10:     **if** *address* < *Table*[*tag*].*lower_bound* **then**
11:         **fail**("use-after-free or buffer underflow detected")
    ▷ Upper bound spatial check
12:     *size* ← **sizeof**(**typeof**(*pointer*))
13:     **if** (*address* + *size*) ≥ *Table*[*tag*].*upper_bound* **then**
14:         **fail**("Buffer overflow detected")

---

the temporal and spatial checks (Algorithm 2 and Algorithm 3) presented in Section 3.3. This optimized check routine, shown in Algorithm 5, requires a specific placement for the safe heap and an adaption of the MESH table, both described in the following.

First, to protect the safe heap against spatial violations through untagged pointers, our combined memory safety check must verify a pointer's address against the bounds of the safe heap (lines 5 to 9). This verification can be implemented using a *single* comparison by placing the safe heap at a low address in memory before *any other* data segment. Hence, we allocate our safe heap at the lowest possible memory page, i.e., the second page on Linux, requiring us only to check if the address of untagged pointers is greater than our safe heap's upper bound.

Then, the combined memory safety check must verify the temporal and spatial validity of tagged pointers using the metadata stored in the MESH table (lines 10 to 14). This verification can be implemented using only *two* comparisons by optimizing the MESH table to encode a heap object's validity flag within its lower bound: If the lower bound has a value of 0xFF...FF, i.e., the highest though invalid memory address, the entire MESH table entry can be considered invalid and any access to the corresponding object is prohibited. Hence, revisiting Figure 1, since the column *validity_flag* is not required by the implementation, we can slim the MESH table to the two columns *lower_bound* and *upper_bound*. The resulting combined check is identical to the spatial safety check from Algorithm 3, except that the check in line 10 can now indicate both a temporal or spatial violation. Both types of violations can still be differentiated afterwards by examining the lower bound: A lower bound of 0xFF...FF indicates a use-after-free while other values indicate a buffer underflow.

4.3 Check Removal

By default, the MESH prototype instruments every pointer dereference. This also includes dereferences loading from or storing to objects that are not allocated on the heap and also do not have compound types. Even though pointers to such objects are not tagged, our instrumentation still requires three additional instructions to verify their tags are zero. Hence, in order to reduce the performance impact of handling untagged pointers, we apply a simple optimization to our instrumentation.

For every pointer dereference, we perform an intra-procedural analysis of pointer origins. If we infer that a pointer originates from an allocation on the stack (i.e., resulting from an *alloca* instruction) or from a global IR variable, we omit the instrumentation of the corresponding pointer dereference if the accessed address is the one allocated and not otherwise derived. The optimization is very conservative, only omitting instrumentation if a pointer's origin can be determined reliably. For example, checks for accesses in which the pointer is going through a PHI node are only omitted if it can be guaranteed that *all* PHI sources are stack allocations which have not been further derived. This ensures that the optimization does not result in MESH missing tagged pointers. In an example case protecting the nginx web server with MESH, we were able to reduce the number of instrumented pointer dereferences by about 5% using our optimization.

4.4 Multithreading Support

Our MESH prototype is fully compatible with user-level and kernel-level multithreading. In the following, we discuss the considerations taken into account for supporting multithreading and present how MESH achieves thread-safety.

4.4.1 Concurrent Allocations. When allocating objects on the safe heap, unique MESH table entries are required to store the objects' metadata. To allow for concurrent allocations, our runtime support library must ensure that the MESH table index I is not modified concurrently. To this end, our heap allocation routines use a global mutex to provide atomic access to the retrieval of a new tag (i.e., the modification of the index). Thread-safety for the actual memory allocation is guaranteed by our underlying heap allocator itself, as our runtime support library merely wraps the heap allocation functions.

4.4.2 Concurrent Deallocations. Deallocating the same object from two different threads is an application-level issue and must be handled by the application programmer. In other words, without application-level thread-safety, this behavior is undefined with or without MESH. Pointers of different objects are associated with different tags, hence, correspond to different MESH table entries so that accesses to invalidate the metadata do not collide.

4.4.3 Concurrent Allocations and Deallocations. For the default configuration of MESH, concurrent allocations and deallocations already handled, as our allocation routines are thread-safe and our deallocation routines do not modify the MESH table index I. However, if the MESH table wrap-around (see Section 3.5) is active, concurrent allocations and deallocations become problematic as unused MESH table entries—which might concurrently be invalidated—can be reused for new allocations. To solve this problem, we protect the invalidation of a MESH table entry with the same global mutex that is used by the allocation routines, making all allocations and deallocations atomic in regard to each other.

Table 1: Memory overhead of memory safety mechanisms

Defense Mechanism	Memory Overhead (approx.)
MESH	$\leq$ 2 MB (x86-64) / $\leq$ 1 MB (ARM64)*
CUP	$\leq$ 32 GB ($2^{31} \cdot 16$)* [8]
ASan	> 200% [42]
LFP	3–11%** [16]
SoftBound (hash table)	87% [29]
SoftBound (shadow)	64% [29]

* Cannot be measured in percent because it is constant
** Depending on precision

5 EVALUATION

To evaluate MESH, we first compare its memory overhead to those of similar memory safety solutions (as discussed in Section 6). Then, we evaluate the performance of MESH on an artificial benchmark and a widely used real-world program. This evaluation is performed on the x86-64 and ARM64 architectures. Finally, we count the number of heap allocations in a mainstream application to determine the maximum number of objects alive at the same time, as well as the total number of allocations performed during the lifetime of the program. We use this to validate our assumption that the MESH table is large enough for small to mid-size programs that typically run on resource-constrained devices.

5.1 Memory Overhead

Table 1 shows a comparison of MESH's memory overhead with the memory overheads incurred by other memory safety solutions. MESH's and CUP's [8] overheads are given as calculated constant maximums, while the others are variable and measured at runtime. The comparison shows that, except for programs with little memory usage, MESH achieves a much smaller memory overhead than the other approaches. Especially solutions based on shadow memory, namely Softbound [29] and ASan [36], typically perform much worse than the approaches based on pointer tagging, namely LFP [16], CUP, and MESH. Although LFP achieves a very low memory overhead, in contrast to MESH, it only provides spatial safety and does not detect memory corruptions with byte-precision, as discussed further in Section 6.

5.2 Performance Overhead

For evaluating MESH's performance, we chose CoreMark[6], a CPU benchmark designed for embedded systems, as a synthetic test and the Nginx[7] web engine in conjunction with the ApacheBench HTTP benchmark[8] as a real-world application test. To evaluate the performance with Nginx, we let ApacheBench generate 10,000 HTTP requests measuring the average response time. We repeated each test five times to exclude outside factors as much as possible. With our two benchmarks, we compared MESH and ASan [36] against a baseline without instrumentation. Since MESH is a heap-only protection, we configured ASan to also protect only the heap.

---

[6]https://www.eembc.org/coremark/
[7]https://www.nginx.com/
[8]http://httpd.apache.org/docs/current/programs/ab.html

Table 2: Performance overhead of MESH and ASan

Program	MESH	ASan (Heap-only)
CoreMark for x86-64	170%	51%
CoreMark for ARM64	111%	30%
Nginx for x86-64	5.6%	8.2%
Nginx for ARM64	3.1%	4.6%

Table 2 summarizes the results of our evaluation. It shows that for CoreMark, MESH is noticeably slower than ASan. This is mainly due to the more complex checking required for the byte-precision and the space-saving MESH table, which, in this case, has a heavy impact as CoreMark involves a large number of memory accesses. However, in case of Nginx's real-world application test, MESH outperforms ASan and causes almost no overhead at all. We explain this difference with the better caching properties of the small MESH table in comparison to ASan's shadow metadata. While CoreMark repeatedly accesses the same memory regions, for which the metadata can be cached well for both solutions, Nginx accesses a larger variety of memory regions, making the checks hard to cache for ASan, while remaining easy for MESH.

Summarizing, depending on the test case, MESH provides performance that is comparable to ASan. However, MESH provides an increased precision, as discussed further in Section 6, and has a lower memory footprint, as shown in Section 5.1.

5.3 Other Metrics

Finally, to validate our assumption that the MESH table has enough entries for practical use, we analyzed the number of allocated objects in Nginx and CoreMark. For Nginx, we counted the allocations for the same ApacheBench test procedure used in the performance evaluation. Since Nginx is a multi-process application and each process uses an independent MESH table, we simply took the maximum number of allocations out of all processes.

The results show that the number of allocated heap objects varies substantially between the two test programs. While CoreMark only allocated *one* heap object over its full execution in our test, Nginx allocated 5211 objects, of which only a maximum of 151 were alive at the same time. In other words, only 4% and 8% of the maximum supported entries of the MESH table were filled for x86-64 and ARM64, respectively. Furthermore, only at most 0.1% and 0.2% of entires were used at the same time. The test results confirm that MESH can be used to effectively protect real-world applications, especially in memory-constrained devices.

6 RELATED WORK

In the following, we analyze how MESH compares to other memory safety approaches. We are mainly interested in solutions that meet at least some of the MESH design goals presented in Section 2.

Fat pointer approaches [2, 21, 30] store an object's metadata alongside its pointers (e.g., using struct-like pointers). Because metadata is stored directly with the protected pointers, such approaches usually only tackle spatial memory safety and do not offer temporal memory safety (❶): If an object is deallocated, all its

Table 3: Comparison of MESH and related memory safety solutions

Goal	MESH	LFP [16]	ASan [36]	HWASan [37]	SoftBound/CETS [28, 29]	CUP [8]
❶ Detect temporal bugs	Yes	No	Imprecise	Probabilistic	Yes	Probabilistic
❷ Detect spatial bugs	Yes	Imprecise	Imprecise	Probabilistic	Yes	Yes
❸ Detect unprotected pointer accesses	Yes	No	Imprecise	Yes	Yes	Yes
❹ Support standard C/C++	Yes	Yes	Yes	Yes	Yes	Yes
❺ Support unprotected code	Yes	Yes	Yes	Yes	No	No
❻ Impose low memory overhead	Yes	Yes	No	No	No	No

pointers must be found and invalidated, which is not trivial without additional data structures. Moreover, replacing normal pointers with fat pointers can cause incompatibilities with language features and uninstrumented libraries, thus failing to meet design goals ❹ and ❺. To tackle this problem, LFP [16] encodes the metadata (still only spatial) inside the value of the pointer itself to protect the heap of 64-bit systems. Similarly to MESH, LFP instruments LLVM IR to tag pointers and associate them with their object's spatial metadata. But since LFP optimizes performance overheads by size-aligning objects to a set of predefined sizes, it loses precision for its spatial checks compared to MESH, as overflows up to the alignment size are not detected (❷). In addition, LFP does not restrict access of non-heap and external pointers, failing to protect against unprotected pointers (❸). However, while not constant, LFP achieves a low memory overhead of 3-11% (❻).

Another approach to memory safety is the use of *shadow memory*. *Location-based* solutions shadow a portion of the addressable memory to store certain attributes (e.g., accessible or non-accessible) about the shadowed memory. Using this shadow, objects are surrounded by *red-zones* [5, 19, 36, 38] or interleaved by *guard pages* [15, 26, 32] to detect spatial memory violations. The most prominent location-based solution is ASan [36], which shadows 8 byte blocks with an 8-bit tag to insert red-zones between memory allocations. ASan performs compile-time instrumentation of pointer dereferences to detect errors when accessing the red-zones. ASan is not byte-precise (8-byte alignment) and only able to detect buffer under- and overflows, but not arbitrary reads and writes skipping red-zones. Hence, unlike MESH, ASan only partially achieves spatial memory safety (❷ and ❸). Furthermore, and also in contrast to MESH, ASan only approximates temporal safety by delaying the reuse of freed memory (❶) and introduces a substantial memory overhead overhead of 237% [36]. But, ASan supports standard C/C++ (❹) and is compatible with uninstrumented code (❺).

An alternative to ASan is Hardware-assisted AddressSanitizer (HWASan) [37], which aims to reduce memory and performance overheads by using specific hardware features. HWASan implements a typical *memory tagging* approach, in which objects in memory and their pointers are tagged with tags that must match for a memory access to be allowed. Similarly to ASan, HWASan uses shadow memory to tag objects, and similarly to MESH, uses unused pointer bits to store its tags. Since HWASan, in its main form, relies on an ARM64-specific feature that allows the processor to ignore the top eight bits in pointers, only 8-bit sized tags are used. While not having to strip pointers before dereferencing significantly increases HWASan's performance, its small tag size makes both spatial and temporal checks probabilistic (❶ and ❷), with a chance of 0.39% [37] for missing bugs even without an attacker

targeting the mechanism. Furthermore, using shadow memory, similarly to ASan, HWASan is less memory-efficient than MESH (❻). However, HWASan supports standard C/C++ language features (❹) and untagged pointers do not interfere with its instrumentation (❺) nor jeopardize the security of tagged objects (❸).

Another group of approaches using shadow memory are *identity-based* solutions, which track object bounds for each pointer. Using these bounds, pointer dereferences are precisely checked to detect spatial memory violations. These solutions either use *per-object bounds tracking* [1, 22, 34, 44], where pointers to the same object share the same bounds, or *per-pointer bounds tracking* [8, 28, 29, 31], where each pointer tracks its own object bounds. Per-pointer bounds tracking is usually more precise, as per-object bounds have to be aligned to powers of two. SoftBound [29], the most prominent identity-based solution, uses per-pointer bounds tracking for spatial memory safety (❷). Additionally, it ensures temporal memory safety with its CETS [28] extension that invalidates pointers on object deallocation (❶). SoftBound/CETS supports standard C/C++ (❹), but is incompatible with pointers originating from uninstrumented code (❺), for which it aborts due to missing metadata (❸). SoftBound/CETS shadows 8 byte blocks (i.e., pointers) with 32 byte of metadata. To reduce memory overhead, a variant of SoftBound/CETS uses a disjoint table to store metadata instead of shadow memory. While this metadata store is similar to MESH, instead of tagging pointers, SoftBound/CETS derives a unique hash from each pointer to access its metadata. On average, with an overhead of 87% for its hash table and 64% for its shadow memory [29], SoftBound/CETS has a significantly higher memory footprint than MESH (❻).

CUP [8] is a recent approach that is similar to MESH design-wise and aims to achieve temporal and spatial memory safety. As MESH, CUP uses a constant metadata table to facilitate per-pointer bounds tracking. But in contrast to MESH, CUP focuses on completeness, partly sacrificing memory-efficiency and modularity in comparison. CUP also leverages the unused bits in pointers on 64-bit architectures to store a link into the metadata table. Pointers in CUP are completely replaced with an index into the metadata table and the pointer's offset into the object. While this frees up additional index bits in pointers, it comes at a performance loss, as, in comparison to MESH, additional steps are required to prepare a tagged pointer for an actual memory access. CUP achieves probabilistic temporal and precise spatial memory safety (❶ and ❷), but the resulting CUP metadata table is very large (❻). Further, while CUP supports standard C/C++ language features (❹), by design, it does not strip pointers for uninstrumented code and mandates the use of a modified `libc` library that is capable of allocating protected objects and handling tagged pointers. Hence, uninstrumented code is not able

to allocate unprotected objects with untagged pointers (❸) nor able to dereference tagged pointers outside of `libc` (❺).

Table 3 summarizes MESH's comparison to other approaches. None of the other approaches is able to achieve all our design goals, confirming the necessity for MESH as a memory-efficient and highly compatible alternative for memory-safe heap solutions.

7 CONCLUSION

We presented MESH, a simple yet efficient safe heap design for C and C++ programs. MESH's metadata store and pointer tagging mechanism ensure constant memory overheads and metadata lookups. We showed the feasibility of our concept with a full LLVM-based implementation for both major 64-bit architectures, x86-64 and ARM64. Our practical evaluation using the Nginx web server and the CoreMark CPU benchmark shows that while the performance overhead imposed by MESH is similar to those of existing memory safety solutions, MESH typically requires magnitudes less memory, causing an insignificant and constant memory overhead, such as 2 MB on x86-64. Its performance and memory efficiency, together with its increased precision and complete compatibility with external code make MESH a viable solution for heap memory safety, especially in resource-constrained scenarios.

ACKNOWLEDGMENTS

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*. USENIX Association.
[2] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. 1994. Efficient Detection of All Pointer and Array Access Errors. In *PLDI*. ACM.
[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *Comput. Surveys* 51, 3 (2018).
[4] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking Blind. In *S&P*. IEEE.
[5] Derek Bruening and Qin Zhao. 2011. Practical Memory Checking with Dr. Memory. In *CGO*. ACM.
[6] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. 2008. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *CCS*. ACM.
[7] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *Comput. Surveys* 50, 1 (2017).
[8] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. 2018. CUP: Comprehensive User-Space Protection for C/C++. In *ASIACCS*. ACM.
[9] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining Light on Shadow Stacks. In *S&P*. IEEE.
[10] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-flow Bending: On the Effectiveness of Control-flow Integrity. In *USENIX Security*. USENIX Association.
[11] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-Oriented Programming Without Returns. In *CCS*. ACM.
[12] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. Losing Control: On the Effectiveness of Control-Flow Integrity Under Stack Attacks. In *CCS*. ACM.
[13] MITRE Corporation. [n.d.]. CVE Vulnerabilities By Type. https://www.cvedetails.com/vulnerabilities-by-types.php. Accessed: 2020-07-12.
[14] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security*. USENIX Association.

[15] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2017. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In *USENIX Security*. USENIX Association.
[16] Gregory J. Duck and Roland H. C. Yap. 2016. Heap Bounds Protection with Low Fat Pointers. In *CC*. ACM.
[17] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *S&P*. IEEE.
[18] Enes Göktaş, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. 2016. Undermining Information Hiding (And What to do About it). In *USENIX Security*. USENIX Association.
[19] Reed Hastings and Bob Joyce. 1991. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter 1992 USENIX Conference*. USENIX Association.
[20] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011).
[21] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *USENIX ATC*. USENIX Association.
[22] Richard W M Jones and Paul H J Kelly. 1997. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In *AADEBUG*. Linköping University Electronic Press.
[23] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *OSDI*. USENIX Association.
[24] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated Software Diversity. In *S&P*. IEEE.
[25] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2019. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* (2019).
[26] Microsoft Corp. 2017. GFlags and PageHeap. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap.
[27] Matt Miller. 2019. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. In *BlueHat IL 2019*. Microsoft.
[28] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *ISMM*. ACM.
[29] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *PLDI*. ACM.
[30] George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-Safe retrofitting of Legacy Code. In *POPL*. ACM.
[31] Harish Patil and Charles Fischer. 1997. Low-Cost, Concurrent Checking of Pointer and Array Accesses in C Programs. *Software: Practice and Experience* 27, 1 (1997).
[32] Bruce Perens. [n.d.]. Electric Fence Malloc Debugger. https://linux.die.net/man/3/libefence.
[33] Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, Ahmad-Reza Sadeghi, and Hamed Okhravi. 2017. Address-oblivious Code Reuse: On the Effectiveness of Leakage-resilient Diversity. In *NDSS*. Internet Society.
[34] Olatunji Ruwase and Monica S. Lam. 2004. A Practical Dynamic Buffer Overflow Detector. In *NDSS*. Internet Society.
[35] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-Oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *S&P*. IEEE.
[36] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC*. USENIX Association.
[37] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. 2018. *Memory Tagging and How it Improves C/C++ Memory Safety*. arXiv. Google LLC.
[38] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *USENIX ATC*. USENIX Association.
[39] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *CCS*. ACM.
[40] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2010. On the Effectiveness of Address-Space Randomization. In *CCS*. ACM.
[41] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. (State of) The Art of War: Offensive Techniques in Binary Analysis. In *S&P*. IEEE.
[42] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for Security. In *S&P*. IEEE.
[43] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *S&P*. IEEE.
[44] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. 2010. PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs. In *ASIACCS*. ACM.