Chapter 3

Memory-Safety Enforcer (MSE)

In the C memory abstraction, storage is described in terms of variables and heap-allocated blocks rather than registers and memory addresses. Conceptually, a program’s memory space consists of a set of disjoint objects each with a well-defined size and lifetime. Each memory access is bound by the language semantics to a specific intended target object, which, roughly speaking, is the object whose address was used to compute the address of the accessed location. Memory safety is the condition in which every memory access falls within the bounds and lifetime of its intended target.

There are two classes of memory-safety violations: a spatial access error occurs when a memory access falls outside the bounds of its intended target, while a temporal access error occurs if the intended target of a memory access has been deallocated. In this work, the term invalid access will be used to refer to either class of memory-safety violation, and the terms invalid read and invalid write will refer to, respectively, a read and a write that violates memory safety.

An invalid access can give rise to unexpected behavior because it may access an arbitrary location in memory; such an access may be a security vulnerability because it may be exploited to access a specific sensitive location in memory it was not intended to access. The control-transfer attacks described in Chapter 2 all require a memory-safety violation to succeed; specifically, they require an invalid write into a sensitive location. Therefore,
enforcing memory safety at runtime is an effective way to protect programs from control-transfer attacks, or other less severe attacks (such as denial-of-service attacks that simply corrupt data).

This chapter describes the underlying idea behind the Memory Safety Enforcer (MSE), an approach to detect memory safety violations for C programs at runtime. The MSE can be used as a security tool to prevent attacks from exploiting invalid accesses, and also as a debugging tool for finding errors that lead to memory-safety violations. We begin by describing a memory-safety model that satisfies the implicit requirements of the C language. We then describe the most frequently used approach to enforce memory safety, called fat pointers, and discuss some of its weaknesses. We then describe the MSE, which uses the tagged memory approach to enforce memory safety.

3.1 Memory-Safety Model

Memory accesses in C can be classified into direct and indirect accesses. Direct accesses, or accesses via directly-named non-array variables, are guaranteed by the language to be memory-safe. Indirect accesses — which we will collectively refer to by the term dereferences — involve either an array index ($a[i]$) or a pointer indirection (*$e$), and may be combined with a structure field or union member selector (e.g., $e.mem[i]$, $e->mem$), or be multi-level (e.g., **$e$, $e->m->n$). Indirect function calls (via a function pointer) are not considered dereferences. For simplicity of presentation, we consider only dereferences of the form *$p$ or $p->mem$ where $p$ is a pointer variable, and assume that all dereferences in the program have been normalized to one of these forms. For example, an array index $A[i]$ can be treated as *$tmp$ where $tmp$ has been assigned the value $A+i$, and $A$ is treated as a pointer containing the address of the array.

Conceptually, at any given moment during program execution, each pointer value is associated with zero or one intended target. The intended target of a pointer value is the object (if any) whose address was used to compute the pointer’s value. For example, after any of the assignments $p = &x$, $p = &x[i]$, and $p = &x[i] + j$, the intended target of $p$ is the
object \( x \). When \( p \) is dereferenced, if the value of \( p \) falls outside the bounds or lifetime of \( x \) (e.g., because the values of \( i \) or \( j \) are too large, or because \( x \) has been deallocated) then a memory-safety violation occurs.

One decision that needs to be made concerns the granularity of the memory-safety model to enforce. We adopt the *outermost-object model*, in which a dereference with a given intended target is allowed to access any part of the *outermost object* containing the target. For example, after the assignment \( p = &x.m \), \( p \)’s intended target is any part of the object \( x \) (not just the \( m \) field). While this model is a good fit for the flexibility of the C language, it can give rise to some peculiarities with respect to structures.

Consider the following structure definition:

```c
struct S {
    int a[10];
    int b;
} s;
```

With the outermost-object model, the dereference \( s.a[i] \) would be allowed to access any part of the structure \( s \), even if it were to overrun the bounds of the array \( s.a \) and access the field \( s.b \) (which would probably happen if \( i = 10 \)).

Given this example, it may seem desirable to adopt a stricter memory-safety model that distinguishes between fields of structures, and restricts the ability of a pointer to one field to access another field. However, since the `offsetof` macro permits one to portably compute the address of one field in a structure based on another field, the stricter memory-safety model is likely to give false positives (i.e., report a deliberately designed dereference as a memory-safety violation). For example, since C requires that the address of the first field of a structure be the same as the address of the structure itself, the following two snippets (operating on the structure defined above) are equivalent in behavior:

<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int * p = (int *) &amp;s;</code></td>
<td><code>int * p = &amp;s.a[0];</code></td>
</tr>
<tr>
<td><code>*(p+i) = ...;</code></td>
<td><code>*(p+i) = ...;</code></td>
</tr>
</tbody>
</table>
With a stricter memory-safety model, if \( i \) contained the value 10 (perhaps computed with the \texttt{offsetof} macro), the dereference in (a) would be permitted while the dereference in (b) would be a memory-safety violation.

Another example that is common in practice is to define a structure with an “open” array as its last field, by declaring the array to have size one:

```c
struct T {
    ...
    int array[1];
} * p;
```

The structure would be allocated dynamically (with \texttt{malloc}) so that \( p->array \) can be treated as an array with more than one element. The stricter memory-safety model would disallow the access \( p->array[i] \) for \( i \geq 1 \).

### 3.2 Fat Pointers Approach

A natural approach to enforcing memory safety for pointers is to record information about the intended target with the pointer; this approach is sometimes called \textit{fat pointers}. A fat pointer [Jim+02] (a.k.a., smart pointer [Ros86], safe pointer [Aus+94], bounded pointer [McG97], augmented pointer [Kee+02b], or sequential pointer [Nec+02]) is a triple, \( \langle \text{ptr}, \text{base}, \text{size} \rangle \), where \( \text{ptr} \) is the pointer value, while \( \text{base} \) and \( \text{size} \) describe the intended target of the pointer. Figure 3.1 gives some example C statements and their equivalents when the pointers (\( p \) and \( q \)) are converted to fat pointers. To enforce memory safety, each dereference \( *p \) is checked to ensure that \( \text{p.base} \leq \text{p.ptr} < \text{p.base} + \text{p.size} \).

There are a number of drawbacks to using fat pointers. Firstly, they do not detect temporal access errors. There are two main solutions that have been proposed to address this problem. One solution is to tag every stack frame and heap object with a unique

---

\(^1\)Technically, this formula is only valid if \( *p \) accesses one byte of memory. To handle the general case, the fat pointer triple can be defined such that the \( \text{ptr} \) field has the same pointer type as the original pointer, the \( \text{base} \) field has type \( \text{char*} \), and the \( \text{size} \) field records the number of bytes in the intended target. Then, since the dereference \( *p \) will access \texttt{sizeof(*p)} bytes of memory, the memory-safety check of \( *p \) must ensure that \( \text{p.base} \leq \text{p.ptr} \leq \text{p.base} + \text{p.size} - \text{sizeof(*p.ptr)}. \)
identifier, and to augment the fat pointer to include the identifier of its intended target, but this approach incurs a high overhead [Aus+94, Kee+02b]. Another solution is to use a garbage collector or a different memory- allocation scheme, like regions [Jim+02], but this solution constrains the programmer’s control over memory management. Furthermore, using a garbage collector may introduce pauses that are unacceptable in certain environments (such as real-time applications), and may restrict the flexibility of the C language (e.g., CCured [Nec+02] does not allow pointers to stack objects to be stored in the heap).

Another problem with fat pointers is that, due to the change in pointer representation, it is difficult to interface with modules and libraries that do not use fat pointers. This is a major obstacle to their widespread use in existing systems or in legacy code. One solution is to translate fat pointers to regular pointers at function-call interfaces to modules that do not use fat pointers; a difficulty of this approach is knowing which function calls need to have their arguments translated, especially in the presence of function pointers. Another solution is to decouple the information about the intended target from the pointer itself: Jones and Kelly [Jon+97] store the information about the intended target in a separate data structure (a splay tree), but incur a high overhead to look up this information.

<table>
<thead>
<tr>
<th>Normal Pointers</th>
<th>Fat Pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>p = &amp;x;</code></td>
<td><code>p.ptr = &amp;x;</code></td>
</tr>
<tr>
<td></td>
<td><code>p.base = &amp;x;</code></td>
</tr>
<tr>
<td></td>
<td><code>p.size = sizeof(x);</code></td>
</tr>
<tr>
<td><code>p = q+1;</code></td>
<td><code>p.ptr = q.ptr+1;</code></td>
</tr>
<tr>
<td></td>
<td><code>p.base = q.base;</code></td>
</tr>
<tr>
<td></td>
<td><code>p.size = q.size;</code></td>
</tr>
<tr>
<td><code>p = &amp;a[i].m;</code></td>
<td><code>p.ptr = &amp;a[i].m;</code></td>
</tr>
<tr>
<td></td>
<td><code>p.base = &amp;a;</code></td>
</tr>
<tr>
<td></td>
<td><code>p.size = sizeof(a);</code></td>
</tr>
<tr>
<td><code>p = malloc(j);</code></td>
<td><code>p.ptr = malloc(j);</code></td>
</tr>
<tr>
<td></td>
<td><code>p.base = p.ptr;</code></td>
</tr>
<tr>
<td></td>
<td><code>p.size = (p.ptr==NULL)?0:j;</code></td>
</tr>
</tbody>
</table>

Figure 3.1 Fat Pointers
A third drawback to using fat pointers is that it is difficult to support certain unorthodox operations, or to support them efficiently. Examples of such practices are dereferencing a pointer that has been cast to an integer and back, using the difference of two pointer values to jump between objects, and declaring open arrays in structures to have size 1. Handling such idiosyncrasies with fat pointers would incur additional overhead and complication, so existing fat pointer approaches either restrict their use, do not handle them safely, or require programmer-added annotations to account for them.

3.3 Tagged Memory Approach

Instead of using fat pointers, our Memory-Safety Enforcer (MSE) adopts an alternative approach based on tagged memory to enforce memory safety at runtime. The underlying idea is to associate extra information with the pointed-to locations, rather than with the pointers. Specifically, each byte\(^2\) of memory is tagged with one bit to indicate whether it is part of the valid target of some dereference. This bit is set to valid when the location is allocated, and invalid when it is deallocated (when exiting a scope or calling free). Prior to each dereference, the tag of the target location is checked: if it is tagged invalid, then a memory-safety violation is reported. A call to free is also checked to see if the target location had been previously allocated with malloc.\(^3\)

In our implementation, the tags are maintained in a “mirror” of memory that is \(\frac{1}{8}\) the size of addressable space, with a tag value of 1 representing valid and 0 representing invalid. Initially, the tags of all memory must be marked invalid, so the entire mirror must be zero-initialized at program startup. On systems that support demand-zero paging, the initial “allocation” and zero-initialization of a large mirror consumes negligible time and resources. The advantage of using a mirror is that looking up the tag of a byte \(B\) takes constant time, since the location of \(B\)’s tag in the mirror can be computed directly from \(B\)’s address. A

\(^2\)We tag each byte of memory because C requires memory to be byte-addressable; in a language and platform with coarser addressability, we need only use one tag bit for each addressable unit of memory.

\(^3\)For brevity, we use malloc throughout this thesis to refer to any of the heap-allocation functions: malloc, calloc, realloc, valloc, memalign.
disadvantage of the mirror implementation is that accessing the tags for \( n \) bytes of memory takes time linear in \( n \). Thus, a big part of the runtime overhead is due to the setting and clearing of tags when allocating and deallocating large blocks of memory.

The tagged-memory approach can detect both spatial and temporal access errors efficiently without restricting the flexibility of C. The programmer can make arbitrary \texttt{malloc} and \texttt{free} calls, rather than be forced to use a specific memory manager or a garbage collector. The programmer is also allowed to manipulate pointers in an arbitrary fashion, including performing unorthodox pointer arithmetic or casting between pointers and integers; when such a pointer is dereferenced, the pointer value is used to look up the tag in the mirror to determine whether the access is valid.

Another benefit of the tagged-memory approach is that the tags are tamper-resistant; that is, an attacker cannot modify the tag of a given location to circumvent the MSE checks. This is because each byte of the mirror is itself tagged \textit{invalid} within the mirror; thus, any attempt to write to the mirror (via a dereference) would be detected as an invalid write.

The tagged-memory approach also allows instrumented modules to interface cleanly with uninstrumented ones. Since there is no change in the pointer representation, there is no need to translate pointer values when calling uninstrumented functions. This is an important feature for facilitating deployment in large systems, where certain components may not have source code readily available, or may not even be written in the same language; it also allows the approach to be applied to a confined subcomponent of the system. Naturally, a memory-safety violation that occurs within an uninstrumented module would not be detected. Further, if an uninstrumented function returns a pointer to a location that is not tagged \textit{valid}, checking a dereference of that pointer may cause a false positive to be reported. To properly account for this, a wrapper function can be written to adjust the necessary tags, or even to perform memory-safety checks. Our MSE implementation includes wrappers for standard C library functions that are vulnerable to memory-safety violations, such as \texttt{fgets}, \texttt{strcpy} and \texttt{bzero}, and functions that return pointers to objects declared within the library, such as \texttt{stat} and \texttt{ctime}. 
To handle \texttt{free} calls precisely, we also maintain a hash table containing information about each \texttt{malloc}-allocated block, including its size. This allows us to check that only \texttt{malloc}-allocated blocks are freed, and tells us how many bytes of memory are being deallocated. This hash table is used for one further optimization: when dereferencing into a large block of memory (e.g., dereferencing a pointer to a structure or writing via a library function like \texttt{bzero}), if we can find that block of memory in the hash table, then the check operation associated with that dereference can take constant time instead of linear time. Note that this only improves performance when the starting address of the checked block is the starting address of a \texttt{malloc}-allocated block of memory.

The main drawback of using tagged memory is that, although it is guaranteed to detect a large class of vulnerabilities (including stack smashing), it is not guaranteed to detect all invalid accesses. However, the use of aggressive static analysis can improve the likelihood of detecting an invalid access.

### 3.4 MSE Classification Framework

For a given program \( P \), we associate each program point with a unique identifier \( i \). Let \( \text{derefs}(P) \) be the set of dereferences in \( P \), with elements of the form \((i,e)\) to represent a dereference \( e \) at program point \( i \). (A call to \texttt{free(}e\texttt{)} is treated as a dereference \( *e \).) Let \( \text{locs}(P) \) be the set of abstract user-defined locations in \( P \), where each abstract location is a static representative of one or more concrete locations at runtime. We define \( \text{locs}(P) \) to include the following:

- \( v \), for each variable \( v \) (including formals) in \( P \).
- \( \text{MALLOC}_i \), for each \texttt{malloc} callsite in \( P \): \( \text{MALLOC}_i \) represents all heap objects allocated at program point \( i \).
- \( \text{STRLIT}_i \), for each string literal occurring in \( P \): \( \text{STRLIT}_i \) represents the string literal declared at program point \( i \).
As defined, $\text{locs}(P)$ is the set of “outermost” objects declared in the program, consistent with the memory-safety model we have adopted.

The tagged-memory approach consists of two phases, classification and instrumentation:

**Classification** : Given a program $P$, compute the following two sets:

- Checked dereferences, $\text{checked-derefs}(P) \subseteq \text{derefs}(P)$, are dereferences that will be checked for memory safety at runtime; dereferences not in $\text{checked-derefs}(P)$ will not be checked.

- Tracked locations, $\text{tracked-locs}(P) \subseteq \text{locs}(P)$, are locations that will be tagged *valid* when allocated and *invalid* when deallocated at runtime. Untracked locations (locations not in $\text{tracked-locs}(P)$) will always be tagged *invalid*.

**Instrumentation** : Given a program $P$ and the sets $\text{checked-derefs}(P)$ and $\text{tracked-locs}(P)$, instrument $P$ so that the following happens at runtime:

- Initially, all locations are tagged *invalid*.

- For each tracked location that is a global variable, set its tag(s) to *valid* prior to the top-level call to `main`.

- When a tracked location is allocated on the stack or the heap, change its tag(s) from *invalid* to *valid*; a tracked static variable is marked as *valid* the first time its declaration is encountered at runtime.

- When a tracked location is deallocated (when exiting a scope or calling `free`), set its tag(s) to *invalid*.

- For each checked dereference, if any part of the target location is tagged *invalid*, report an error and halt the program.

Programs are instrumented via a C source-to-source transformation (using the Ckit front end [Ckit]). Instrumenting at the source level makes our tool portable, as an instrumented source file can be compiled on any platform that supports C.
### 3.4.1 Naive Classification

The classification phase determines the amount of instrumentation to add. We begin by naively choosing $\text{checked-derefs}(P) = \text{derefs}(P)$ and $\text{tracked-locs}(P) = \text{locs}(P)$, so that all dereferences will be checked and all user locations will be tagged $\text{valid}$ when allocated. This means that a memory-safety violation will be detected if and only if it accesses memory outside of user-defined space (such as the return address in the activation record, or memory that has been deallocated).

Figure 3.2 shows the buffer-overrun example described earlier in Section 2.1, with column (c) depicting the runtime tags associated with the naive classification scheme (where each of the user-defined locations $\text{src}$, $\text{fp}$, $\text{buf}$, and $\text{dst}$ is tagged $\text{valid}$). While a stack-smashing attack that overwrites the return address (ret) would be detected, an attack that overwrites the function pointer $\text{fp}$ would not. The goal of static analysis is to develop a better classification scheme, in particular, one that can identify $\text{tracked-locs}(P) = \{\text{buf}\}$, giving the tag layout of column (d).
3.4.2 Read-Write vs. Write-Only Checking

Let write-derefs($P$) ⊆ derefs($P$) be the set of dereferences that are writes — i.e., that are on the left-hand-side of an assignment (or the operand of the pre- or post-fix operator $++/--$). One high-level policy we can make is to check only writes via dereferences, that is, to choose checked-derefs($P$) = write-derefs($P$). As was seen in the examples in Chapter 2, control-transfer attacks usually need to exploit an invalid write to succeed. Since reads occur more frequently in programs than writes, checking only writes can let us gain significant improvements in performance without sacrificing too much coverage (where “coverage” means the likelihood of detecting an attack). In fact, because a smaller checked-derefs($P$) can lead to a smaller tracked-locs($P$) — as will be shown in Section 4.1 — checking only writes can improve coverage as well. However, read-write checking may still be desirable in certain high-security settings to prevent an attacker from reading confidential data, or may be useful for debugging.

![Percentage Slowdown](image)

**Figure 3.3 Naive Classification Runtime Overhead**

Figure 3.3 gives the runtime overhead of MSE using the naive classification scheme. The benchmarks (see Appendix A for details) are sorted in increasing order of size (lines of code), to highlight any dependence on program size (in this case, the overheads are essentially
independent of program size). The percentage slowdown is computed as \( \frac{t_{\text{inst}} - t_{\text{orig}}}{t_{\text{orig}}} \times 100\% \), where \( t_{\text{inst}} \) is the running time of the instrumented executable, and \( t_{\text{orig}} \) is the running time of the original (uninstrumented) executable. The average slowdown is 318\% in read-write checking mode, and 160\% in write-only checking mode. Not surprisingly, the overhead of write-only checking is significantly better, confirming our earlier assertion that write-only checking can result in significant gains in performance.

These runtime overheads are quite high, and are probably not acceptable for use in deployed code. They serve as a baseline case, against which improvements due to static analysis, to be presented in the following chapters, can be measured.

### 3.5 Improving MSE

To improve both the performance and the coverage of the Memory-Safety Enforcer, classification schemes based on several static analyses are used to identify smaller \textit{checked-derefs} and \textit{tracked-locs} sets. These are described in the following chapters:

- Chapter 4: Points-To Analysis
- Chapter 5: Redundant Checks Analysis
- Chapter 6: Pointer-Range Analysis

Each static analysis may be applied in either read-write checking mode or in write-only checking mode.

The static-analysis classifications are constrained to preserve coverage and soundness; that is, an invalid access that is detected by the naive classification must be reported by the improved classification, and no false positives may be reported. At a high level, the idea behind the static analyses is first to classify each dereference as either \textit{definitely safe} or \textit{potentially unsafe}. A dereference that is definitely safe is one that can be statically determined to never cause an invalid access, and so can be removed from \textit{checked-derefs}(P); only potentially unsafe dereferences will be checked. Next, a location is identified as tracked if it may be
legitimately accessed by a potentially unsafe dereference; an untracked location is one that is either never accessed by a dereference, or is only accessed by definitely safe dereferences. The goal of static analysis is to identify as few potentially unsafe dereferences as possible, which leads to a smaller set of tracked locations. Having fewer tracked locations (which are tagged as valid at runtime) increases the likelihood that a memory-safety violation will access a location tagged invalid, and thus be detected. Fewer potentially unsafe dereferences and tracked locations also leads to fewer runtime checks, which results in improved runtime performance.