# Hardening Binaries against More Memory Errors

Gregory J. Duck
Department of Computer Science
National University of Singapore
Singapore
gregory@comp.nus.edu.sg

Yuntong Zhang
Department of Computer Science
National University of Singapore
Singapore
yuntong@comp.nus.edu.sg

Roland H. C. Yap
Department of Computer Science
National University of Singapore
Singapore
ryap@comp.nus.edu.sg

## Abstract

Memory errors, such as buffer overflows and use-after-free, remain the root cause of many security vulnerabilities in modern software. The use of closed source software further exacerbates the problem, as source-based memory error mitigation cannot be applied. While many memory error detection tools exist, most are based on a single error detection methodology with resulting known limitations, such as incomplete memory error detection (redzones) or false error detections (low-fat pointers). In this paper we introduce RedFat, a memory error hardening tool for stripped binaries that is fast, practical and scalable. The core idea behind RedFat is to combine complementary error detection methodologies—redzones and low-fat pointers—in order to detect more memory errors that can be detected by each individual methodology alone. However, complementary error detection also inherits the limitations of each approach, such as false error detections from low-fat pointers. To mitigate this, we introduce a profile-based analysis that automatically determines the strongest memory error protection possible without negative side effects.

We implement RedFat on top of a scalable binary rewriting framework, and demonstrate low overheads compared to the current state-of-the-art. We show RedFat to be language agnostic on C/C++/Fortran binaries with minimal requirements, and works with stripped binaries for both position independent/dependent code. We also show that the RedFat instrumentation can scale to very large/complex binaries, such as Google Chrome.

***CCS Concepts:*** • **Software and its engineering**; • **Security and privacy → Software and application security**; **Systems security**;

***Keywords:*** Static binary rewriting, binary instrumentation, binary hardening, memory safety, memory errors, buffer overflows, use-after-free, low-fat pointers, redzones



*EuroSys '22, April 5–8, 2022, RENNES, France*

ACM ISBN 978-1-4503-9162-7/22/04.
https://doi.org/10.1145/3492321.3519580

**ACM Reference Format:**
Gregory J. Duck, Yuntong Zhang, and Roland H. C. Yap. 2022. Hardening Binaries against More Memory Errors. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3492321.3519580

## 1 Introduction

The problem of memory errors, such as buffer overflows and use-after-free, date back to the 1970s [33]. Unfortunately, memory errors are still the major source of security vulnerabilities in modern software, with Google and Microsoft reporting that "*70% of our serious security bugs are memory safety problems*" [7, 22]. One mitigation is to *harden* the program against memory errors using a suitable *sanitizer* [32], such as AddressSanitizer (ASAN) [29] or LowFat [10]. Such sanitizers work by *instrumenting* the program with explicit memory safety checks, thereby detecting memory errors before they occur. However, most sanitizers are implemented as compiler transformations, meaning that the availability of source code is taken for granted.

For closed-sourced or *Commercial Off-The-Shelf* (COTS) software, source-code-based sanitizers are not applicable. However, binary-only software can be just as vulnerable to memory errors and 0-day attack, meaning that hardening against exploitation may still be necessary. Rather than source-level instrumentation, we can attempt to instrument the binary code directly. However, directly instrumenting binary code is known to be a challenging problem. For example, (i) binaries are usually "stripped", meaning that source-level information, such as symbols and types, are not available; (ii) instrumenting (modifying) the binary is difficult to do in a reliable fashion; and (iii) the overheads of binary instrumentation are generally much higher than source-level instrumentation. As such, most existing binary instrumentation tools [5, 8, 25] have one or more limitations, such as high overheads, limited programming language support (e.g., no C++ binaries), or only support specific types of binaries (e.g., *Position-Independent-Code* (PIC) only).

Furthermore, due to the lack of type information at the binary-level, existing tools are limited to specific classes of memory error. For example, tools such as [5, 8, 25] detect out-of-bounds errors (e.g., buffer overflows) by wrapping objects with inaccessible *poisoned redzones* (or just *redzones* for short). However, a well-known limitation of this method is the inability to detect out-of-bounds errors that

"skip" redzones to access adjacent valid objects. For example, given an array A of length $n$, then A[$n$] will access the redzone and be detected as an overflow. However, the "bigger" overflow of A[$n$+16] will "skip" a redzone (of size 16) altogether and access the next object (say B) in memory. The "bigger" overflow will not be detected with redzone-based tools, meaning that the instrumented binary may still be vulnerable. Fixing the problem is non-trivial, since the lack of type information means that it is difficult to distinguish valid from invalid memory access. For example, the "invalid" access A[$n$+16] may be indistinguishable from "valid" access (B−$n$−16)[$n$+16] at the binary level, since both can be compiled into the same instruction.

In this paper, our objective is to harden binaries, including COTS binaries, against memory errors with minimal assumptions for broader applicability. We want to be language and compiler agnostic, as well as to avoid making special assumptions such as specific languages or PIC, which may not hold for COTS binaries. We want to scale to large binaries (e.g., Google Chrome at >100$MB$), all with a reasonable performance overhead. Finally, we want to detect "more" memory errors, i.e., detect as many memory errors as is practically possible, including out-of-bounds errors that skip over redzones. The latter is especially important for the binary hardening use case, as opposed to testing and bug detection, since the aim is to prevent the binary from being exploited against an adaptable attacker. For hardening, it is important to detect as many classes of memory error as is practically possible.

In order to increase error detection coverage, our approach is to harden binaries using "complementary" memory error detection methodologies—specifically, we augment redzones with pointer arithmetic checking in the form of *low-fat pointers* [10]. The basic idea is that low-fat pointers can detect some classes of memory error that are missed by redzones alone, and vice versa, leading to a synergistic combination.

However, adapting pointer arithmetic checking to binary code faces several challenges. As noted above, one main challenge is that both valid and invalid memory access may be compiled into identical instructions, making it impossible to distinguish between the two cases by inspection alone. Without this distinction, naïve pointer arithmetic checking may incorrectly flag some legitimate memory access as an out-of-bounds error, i.e., a *false detection* (a.k.a., a *false positive*). To address this challenge, our philosophy is *opportunistic hardening*—i.e., harden the binary as much as possible without compromising on scalability and negative side effects. For example, to address the problem of false positives, we introduce a *profile-based* analysis that can use dynamic information to detect problematic memory access operations for which pointer arithmetic checking may not be accurate. For such operations, the instrumentation can be adapted accordingly, thereby avoiding the problem of false positives in production code. Finally, we show that affected operations are relatively rare in practice, meaning that the majority of memory access can still enjoy full redzone and pointer arithmetic checking.

*Contributions.* In this paper, we present RedFat—a binary instrumentation tool that is primarily designed towards the *hardening* use case, rather than testing/debugging. As such, RedFat aims to detect "more" memory errors that other tools can miss, including memory errors that could be used by an attacker to bypass existing state-of-the-art tools. In summary, the main contributions of this paper are:

- We present a *complementary* memory error protection scheme for binaries, combining strengths of *redzone* and *low-fat pointer* memory error detection. As such, RedFat can detect more classes of memory error than the current state-of-the-art tools that are based on redzones-only. This will also be shown by our evaluation.
- We adapt pointer arithmetic checking, in the form of *low-fat pointers*, to binaries. Specifically, we present a solution to the problem of *false positives* using automatic allow-list generation based on a profile-based analysis. We show this approach is pragmatic.
- We present several optimizations which significantly reduce overheads introduced by binary rewriting.
- We have implemented our approach in the form of the RedFat tool. We evaluate RedFat on standard benchmarks as well as a web browser. The overhead is 1.55× for the full SPEC CPU2006 benchmark for write protection, which is sufficient to harden the binary against common attacks and is usable for many (COTS) binaries. For protecting both reads and writes, the overhead increases to 3.81× (compared to 11.76× overhead for the current state-of-the-art tool: Valgrind Memcheck [25, 30]). We show that the RedFat instrumentation can scale to large and complex binaries, such as the Chrome browser, which is particularly challenging for binary instrumentation tools.

**Open Source Release**
https://github.com/GJDuck/RedFat

## 2 Background and Motivation

Our high-level aim is to harden binary programs against memory errors, such as out-of-bounds errors (e.g., buffer overflows) and use-after-free. Here, we summarize the current memory error instrumentation technologies.

### 2.1 Memory Error Checking

Many different memory error checking methodologies exist. These are broadly classified into object-based and pointer-based methods. Object-based methodologies work by attaching meta information to allocated objects, allowing for certain classes of memory errors to be detected when the object is accessed. Pointer-based methodologies attach meta information to pointers, allowing for invalid pointer arithmetic

to be detected. Each approach has its own set of advantages and disadvantages.

*Redzones.* One popular object-based methodology for detecting memory errors is *poisoned redzones*—an approach used by state-of-art memory safety tools such as AddressSanitizer (ASAN) [29] and Memcheck [30]. The basic idea is to surround each allocated object (e.g., as returned by `malloc`) with a small block of unused memory, i.e., the *redzone*. The redzone memory is not meant to be accessed, and thus any malicious or unintentional access at runtime will be deemed an out-of-bounds error.

A common method for implementing redzones is to map the program's virtual address space into a special *shadow memory* region that tracks the state of corresponding memory. A basic implementation will track at least two states: Allocated and Redzone. Only access to memory marked as Allocated is allowed, and access to Redzone memory will be detected as a memory error. This implements a form of *memory poisoning*, i.e., the redzone memory is effectively "poisoned" meaning that it cannot be accessed. Some implementations also implement a third state (Free) that marks `free`'ed objects, allowing for *use-after-free* error detection.

Memory error detection is implemented using *instrumentation*, i.e., additional code inserted into the program to detect whether the accessed memory is poisoned or not. To do so, a runtime system provides a $state(ptr)$ operation that maps a pointer *ptr* to the state of the corresponding memory (e.g., stored in the shadow memory). The program is then instrumented using the following schema that explicitly checks for poisoned memory before each read or write operation:

```
if (state(ptr) ≠ Allocated)
    error();
*ptr = val;      or      val = *ptr;
```

(Redzone)

Only access to Allocated memory is allowed. Access to Free memory is detected as a use-after-free, and Redzone memory is detected as an out-of-bounds error.

While redzones are effective at detecting many out-of-bounds and use-after-free errors, not all errors can be detected [29]. For example, consider the following pair of vulnerable code snippets:

```
int n = input();
for (i = 0; i < n; i++)
    array[i] = val;
```

(a) incremental

```
int i = input();
array[i] = val;
```

(b) non-incremental

An attacker that controls the input can induce an out-of-bounds error in both snippets (a) and (b). Snippet (a) is *incremental*—the code accesses memory sequentially. Snippet (b) is *non-incremental*— `array[`$i$`]` can be accessed without constraint. A redzone-based defense will always detect the overflow in the incremental snippet (a) when the buffer overflow runs into the redzone. However, for the non-incremental snippet (b), the attacker can craft a value for `i` that can "skip-over" one (or more) redzones and into other objects. As no redzone memory is accessed, no memory error is detected. This leads to our first problem statement:

```
1  static int
2  channelised_fill_sdh_g707_format(
3      sdh_g707_format_t* in_fmt,
4      guint16 bit_flds,
5      guint8 vc_size,
6      guint8 speed)
7  {
8      int i = 0;
9      if (0 == vc_size)
10         return -1;
11     in_fmt->m_vc_size = vc_size;
12     in_fmt->m_sdh_line_rate = speed;
13     memset(&(in_fmt->m_vc_index_array[0]),0xff,
14            DECHAN_MAX_AUG_INDEX);
15     in_fmt->m_vc_index_array[speed - 1] = 0;
16     // ...
17     return 0;
18 }
```

**Figure 1.** Code snippet for CVE-2012-4295. Line 15 has a non-incremental out-of-bounds error.

> ***Problem #1****: Poisoned memory/redzones are effective at detecting incremental out-of-bounds and use-after-free errors, but ineffective at detecting non-incremental out-of-bounds errors.*

**Example 1** (Non-incremental Overflow)**.** We illustrate a non-incremental memory error with a real world example. The `wireshark` program contains a non-incremental bug (CVE-2012-4295) as shown in Figure 1. The vulnerable statement is at line 15. Here, `im_fmt` is a heap-allocated struct, and `im_fmt->m_vc_index_array` is an array sub-object with length `DECHAN_MAX_AUG_INDEX` (5) `char` elements. The input `speed` parameter can be controlled by an attacker using a specially crafted packet, either sent through the network or loaded through a malicious *Packet Capture* (PCAP) file. Valgrind Memcheck [25, 30], a state-of-the-art memory error detection tools for binaries, uses a 16 byte redzone by default, meaning that a large enough value for `speed` (e.g., 200) is sufficient to "skip-over" the `im_fmt` struct's redzone and bypass detection. This allows the attacker to zero bytes in adjacent heap objects or `malloc` headers, which can be used to change pointer or data values, and this may be used as the basis for an attack. □

*Low-fat pointers.* One alternative to redzone-based memory error detection is *low-fat pointers* [10, 13]—a method for encoding bounds meta information (e.g., object size and base) directly into the pointer representation itself. Conceptually,

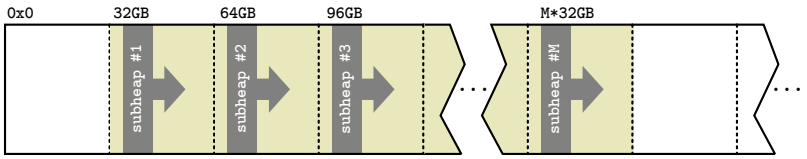


**Figure 2.** The low-fat allocator memory layout. Here, the virtual address space of the program is partitioned into several large (32GB) regions. Regions #1-#M are managed by the low-fat allocator, and each contains a *subheap* that services memory allocations of a specific size range.

low-fat pointers could be implemented using the following simple example encoding:

```
union {
    void *ptr;
    struct {
        uintptr_t size:10;    /* MSB */
        uintptr_t unused:54;
    };
};
```

Here, the *object size* can be directly retrieved using the `size` field, i.e., (*size*(*ptr*)=*ptr*`.size`). Furthermore, by ensuring that all allocated objects are `size`-aligned, the *object base address* can also be retrieved by rounding-down `ptr` to the nearest `size`-multiple, i.e.:

$$base(ptr) = ptr\texttt{.ptr} - (ptr\texttt{.ptr} \bmod ptr\texttt{.size})$$

Finally, the pointer value itself (*ptr*`.ptr`) must be *dereferencable*, i.e., a native `x86_64` machine pointer. The latter differentiates low-fat pointers from traditional "*fat pointers*", such as those implemented by CCured [24], which are incompatible with existing binary code. By replacing the default memory allocator (e.g., `malloc`) with a "low-fat" allocator (e.g., `lowfat_malloc`), the base/size of allocated objects can be easily tracked. For most binaries, the default memory allocator can be replaced using the `LD_PRELOAD` trick.

The simplified low-fat pointer encoding (above) is difficult to implement in practice, since it imposes strong constraints on object placement in the program's virtual address. Instead, we use the more sophisticated low-fat pointer encoding of [10, 13]. This encoding works by partitioning the program's virtual address space into several large (e.g., 32GB) equally-sized *regions* as illustrated by Figure 2. There are two main types of regions: *low-fat regions* #1-#*M* (for some predefined *M*) that contain objects managed by the low-fat allocator, and *non-fat regions* that contain everything else (stack, globals, code, etc.). Each low-fat region will contain a *subheap* that services allocations of a predetermined size range. For example, under the default configuration of [13], region #1 handles allocations of sizes 1-16 bytes, region #2 handles allocations of sizes 17-32 bytes, region #3 handles sizes of 33-48 bytes, etc. As with the simplified encoding, allocated objects are *aligned* to a multiple of the allocation size. Under this design, we can define efficient *size*(*ptr*) and *base*(*ptr*) operations as follows:

$$size(ptr) = \texttt{SIZES}[ptr \;/\; 32GB]$$
$$base(ptr) = ptr - (ptr \bmod size(ptr))$$

Here, $32GB$ is the region size, and `SIZES` is a predefined table mapping region indices to allocation sizes. The *size*(*ptr*) and *base*(*ptr*) operations are designed to be very fast, and can be implemented in a few `x86_64` instructions. Finally, by defining `SIZES`[*i*]=`SIZE_MAX` for non-fat regions #i, we see that *size*(*ptr*)=`SIZE_MAX` and *base*(*ptr*)=`NULL`, meaning that "over approximate" (but otherwise valid) bounds will be returned for non-fat pointers (*ptr*). This ensures that non-fat pointers will always be considered "in bounds".

The low-fat allocator of [10, 13] has been implemented as part of the LowFat tool [20]. Compared to `glibc malloc`, the overheads of the low-fat allocator are low (~1% performance [11] and ~3% memory [10]). The low-level implementation details of the low-fat allocator are complex, so we refer the reader to [10] for more information.

The primary application of low-fat pointers is bounds checking. Specifically, low-fat pointers can be used to check whether *pointer arithmetic* is within the bounds of the allocated object using the following instrumentation schema:

```
const void *lb = ptr + i;
const void *ub = ptr + i + sizeof(*ptr);
if (lb < base(ptr))
    error();
if (ub > base(ptr) + size(ptr))
    error();

ptr[i] = val;      or      val = ptr[i];
```

(LowFat)

Given a dereference of the form (*ptr*`[`*i*`]`), the (LowFat) check enforces the property that the pointer arithmetic (*ptr* + *i*) is valid w.r.t. the *ptr*. Since the (LowFat) check is protecting the pointer arithmetic itself, any out-of-bounds error that "skips-over" the space between objects can be detected as an error. For example, in Example 1, a sufficiently large value for the array index (`speed`) can be used to bypass the (Redzone) check. However, given *ptr*=`&in_fmt->m_vc_index_array` and *i*=`speed`−1, the same overflow will be detected by (LowFat) check, regardless of the value of `speed`. As such, low-fat pointers are useful for hardening applications where the attacker is aware of the defense.

Although detecting non-incremental overflows is an advantage, low-fat pointers also have disadvantages. One disadvantage is that low-fat pointers may report *false positives* in the case where the programmer (or compiler) *intentionally* creates "out-of-bounds pointers". For example, one known `C` anti-idiom is to create an array indexed from a non-zero value $K$, i.e., `array[K]` is the first element, `array[K+1]` is the second, etc. This can be implemented by subtracting $K$ from the array pointer before access, as illustrated by the snippet:

```
array -= K;
int i = input2();
array[i] = val;
```

(c) false positive

For the sake of an example, we shall assume that `input2()` always returns a valid index. Since the resulting array access (`array[i]`) always targets the intended object, this snippet will not be flagged by poisoned redzones instrumentation. However, for low-fat pointers, this snippet appears to be an out-of-bounds error overflowing whatever object happens to be pointed to by `array` into the object pointed to by `array+i`. The problem is also exacerbated at the binary level, where the compiler can optimize the generated code to use out-of-bounds pointers, even for cases where such pointers do not exist at the source-code level.

It is tempting to use static binary analysis to distinguish intentional and unintentional out-of-bounds memory access. However, the problem is undecidable in the general case. For example, assuming the compiler chooses the following register assignment `array=%rax`, `i=%rbx` and `val=%rcx`, then the assignment in snippets (b) and (c) can be plausibly compiled into the same `x86_64` instruction:

```
mov %rcx, (%rax,%rbx,4)   # array[i] = val
```

Thus, to determine if this instruction is a false positive or not, the context of the instruction must be analyzed. However, even the sub-problem of determining all context instructions is generally undecidable. This leads to the second problem statement:

> ***Problem #2**: Low-fat pointers cannot distinguish between intentional and unintentional out-of-bounds pointer arithmetic, possibly leading to false positives.*

It should be noted that Problem #2 also applies to source-level low-fat pointers [10, 13]. However, the deliberate creation of out-of-bounds pointers is *undefined behavior* under the `C`/`C++` semantics, so such detection can still be justified. However, this rationale cannot be applied to binary hardening, since high-level `C`/`C++` language semantics are not relevant at the `x86_64` binary-level.

Besides the potential for false positives, low-fat pointers also have some additional challenges/limitations, namely:

- *No use-after-free detection*: Since the metadata is stored in the pointer rather than the object, low-fat pointers do not protect against use-after-free errors.
- *Out-of-bounds detection granularity*: A typical `malloc` implementation will *pad* allocations in order to satisfy minimum size and alignment constraints. For example, an allocation of size 13 bytes will be "rounded-up" to 16 bytes by adding 3 bytes of padding. The low-fat pointer instrumentation of [10, 13] will not detect overflows into such padding.
- *Detecting pointer arithmetic*: The detection of pointer arithmetic at the binary-level can be difficult. For example, the instruction (`addq %rax, %rbx`) may be used for both *pointer* or *integer* arithmetic, and it may not be possible to differentiate the two cases through static analysis.

## 2.2 Binary Rewriting

Our overall aim is to harden stripped binary code by inserting *instrumentation* to check for memory errors. To do so, we need some method for *rewriting* the original binary into a new version with the instrumentation inserted. For this we will use a form of *binary rewriting* [36].

One approach is *Dynamic Binary Instrumentation* (DBI) which rewrites the binary code as the program executes, essentially inserting the instrumentation on top of *Just-in-Time* (JIT) execution. Well known DBI systems include PIN [21], DynamoRIO [4] and Valgrind [25]. The main disadvantage of DBI is the cost of the JIT as well as requiring a heavyweight infrastructure. Valgrind is one such tool designed for heavyweight DBI and applications such as Memcheck [30]. Another approach is *static binary rewriting*, which aims to rewrite binary files "on disk". This approach does not require a heavyweight infrastructure, but can also be difficult to get right. For example, inserting new code can modify jump/call targets, and this must be corrected in the output binary. Some tools attempt to statically recover control flow information (i.e., jump/call targets), usually resorting to heuristics/assumptions about the input binary. For example, Egalito [37] and RetroWrite [8] assume that the binary is compiled from `C` and is *Position Independent Code* (PIC). As such, these tools cannot rewrite binaries compiled from `C++`.

An alternative is *trampoline*-based static binary rewriting. This approach attempts to minimize the number of moved instructions, thereby preserving the original control-flow as much as possible. To do so, the instrumentation is placed in a *trampoline* that is placed at an (otherwise-unused) virtual address. At the instrumentation point, the corresponding instruction is replaced by a jump that redirects control-flow to the trampoline. The trampoline executes (1) the instrumentation (2) the displaced instruction, and (3) a *jump* back to the next instruction after the insertion point. This approach can scale better without making assumptions on the source language, compiler or PIC requirements. Recent

advances in trampoline-based rewriting for the `x86_64`—namely, the E9Patch tool [9]—have made it possible to insert jumps-to-trampolines at any location, regardless of instruction size. This approach has proven to be very scalable, and is able to rewrite very large/complex programs including web browsers (Google Chrome and FireFox). The details are very technical, so we refer the reader to [6, 9] for more information. In this paper, we employ E9Patch as the underlying binary rewriter, which allows RedFat to be language (C/C++/Fortran), compiler (`gcc`/`clang`), and PIC agnostic.

## 3 Design

Both the redzone and low-fat pointer-based approaches have pros and cons. The redzone approach can detect use-after-free errors and incremental out-of-bounds errors, but may not protect against non-incremental out-of-bounds errors. The low-fat pointer approach can detect non-incremental out-of-bounds errors, but not use-after-free errors and may suffer from false positives. For security hardening, we argue that neither approach offers satisfactory protection. For example, an attacker may exploit a non-incremental out-of-bounds error to bypass the redzone check (i.e., Problem #1) or a use-after-free to bypass the low-fat pointer check. Instead of choosing one approach over the other, our core idea is to combine the (Redzone)+(LowFat) instrumentation together for "complementary" protection, meaning that an attack missed by one defense can be detected by the other. As such, complementary protection offers an overall stronger defense than each individual protection can offer alone.

However, protecting all memory access with combined (Redzone)+(LowFat) checking exhibits some practical difficulties. For example, in addition to *false positives* (see Section 2, Problem #2), it is generally difficult to differentiate pointer and integer arithmetic at the binary level. To address these issues, our general design philosophy will be *opportunistic hardening*, meaning that we should use the stronger (Redzone)+(LowFat) checking as much as possible, and only fall back to (Redzone)-only checking when the stronger check cannot be justified. As will be shown by our experimental results (Section 7), most memory access can be protected by the stronger (Redzone)+(LowFat) check. The process of merging instrumentation methodologies also introduces technical challenges, such as managing performance overheads.

We now summarize the main design decisions with corresponding justification. Our overall aim is to build a practical binary hardening tool that maximizes protection, provided that correctness and scalability can be maintained.

*Scalability and performance.* We aim to build a usable binary hardening tool that can scale to large/complex binaries, as well as being language/compiler/PIE agnostic. Generally we prioritize scalability over performance, and choose a trampoline-based binary rewriting methodology (E9Patch) as the foundation of the tool. To reduce the performance impact as much as possible, we introduce several optimizations, including check *elimination*, *batching* and *merging*.

Another concern is the design of the instrumented check itself. Simply concatenating complementary protections together effectively doubles the number of checks, and this can be another source of overhead. Existing redzone implementations have little synergy with existing low-fat pointer implementations, meaning that the necessary object-tracking bookkeeping is essentially doubled. One of our aims is to design a hybrid (LowFat)+(Redzone) check so that as much functionality overlaps as possible, minimizing the performance impact and implementation effort. In summary:

> *Use a scalable trampoline-based binary rewriting methodology, and optimize as much as possible.*

*Pointer arithmetic.* The (LowFat) instrumentation checks whether a pointer arithmetic operation (`ptr+i`) is within the bounds of the object. However, `x86_64` binary code lacks type information, making it difficult to differentiate pointer and integer arithmetic. One exception is `x86_64` *memory operands*, which combine pointer arithmetic and memory access into a single instruction. For example, consider the instruction ⟨`movq %rax, (%rbx,%rcx,4)`⟩ with the memory operand highlighted. This instruction is a valid compilation of the statement `*(ptr+i)=val` with `ptr=%rbx`, `i=%rcx`, and `val=%rax`. Here, the pointer arithmetic is unambiguous, meaning that the full (LowFat)+(Redzone) check ought to be applied under the philosophy of opportunistic hardening. In summary:

> *Use a conservative redzone-based policy by default, otherwise opportunistically harden with lowfat+redzone if pointer arithmetic is unambiguous.*

*False positives.* The (LowFat) instrumentation may suffer from *false positives* caused by the program deliberately using out-of-bounds pointers. On one hand, false positives impact on the utility of the defense. On the other hand, pure redzone-based defenses may miss non-incremental out-of-bounds errors, and this could be exploited by an attacker. Under the philosophy of opportunistic hardening, we aim to use the combined (Redzone)+(LowFat) for cases where false positives are deemed unlikely to occur, and to fall back to (Redzone) otherwise. In summary:

> *Use a conservative redzone-based policy by default, and use a combined lowfat+redzone policy when deemed safe from false positives.*

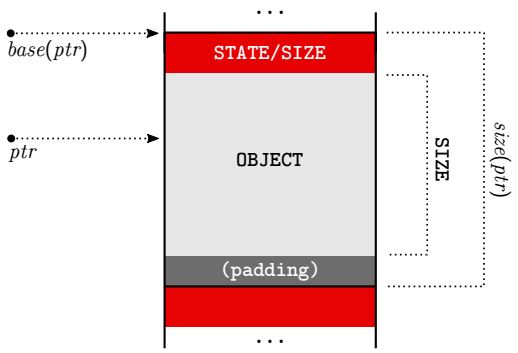


**Figure 3.** Illustration of the layout of an `OBJECT` allocated by the replacement `malloc(SIZE)`. In this illustration we assume that addresses grow down. Here, each object is *prepended* with a (16 byte) *redzone* that also stores state/size metadata. Given a pointer (*ptr*) into the object, the metadata can be quickly retrieved using the low-fat *base*(*ptr*) operation. The low-fat allocator may also `(pad)` the allocation in order to satisfy size/alignment constraints.

The static detection of false positives is undecidable in the general case. Instead, we propose a dynamic (profile-based) analysis that can automatically generate an *allow-list* for all memory access deemed to be (LowFat)-safe (see Section 5). Members of the allow-list will therefore be instrumented with the stronger (Redzone)+(LowFat) check, otherwise the default (Redzone)-only check will be used. Although dynamic analysis has known limitations, the approach is nevertheless justified under opportunistic hardening.

## 4 Memory Error Detection

Our aim is to enforce the combined (Redzone)+(LowFat) check whenever possible. A naïve approach would be to simply concatenate the two instrumentation schemas, essentially doubling the instrumentation. Instead, we propose a more optimized approach, by redesigning each schema so that as much functionality can be shared as possible.

### 4.1 Redzone and Low-fat Instrumentation

*Redzone*-based memory error detection works by (1) padding objects with *redzone* memory (e.g., 16 bytes), and (2) maintaining a *shadow state* mapping virtual addresses to an Allocated, Free and Redzone state. The runtime system provides a *state*(*ptr*) operation that maps the given virtual address *ptr* to the corresponding state. To pass the (Redzone) check, the state must always be Allocated. AddressSanitizer (ASAN) [29], implements the *state*(*ptr*) operation using a *shadow memory*—a reserved memory region storing the state value of each corresponding 8 byte word. The *state*(*ptr*) operation can be implemented as:

$$state_{shadow}(ptr) = *(\texttt{SHADOW\_MAP} + (ptr \div 8))$$

The division can be efficiently implemented as a left shift.

We present an alternative implementation of redzones by leveraging the low-fat memory allocator and low-fat pointers. The idea is to store metadata inside the redzone itself, since redzone memory is otherwise unused, thereby eliminating the need for a separate (large) shadow memory. To do so, we implement a replacement memory allocator (`malloc`) that is implemented as a wrapper over the underlying low-fat allocator (`lowfat_malloc`) introduced in Section 2. The wrapper transparently prepends a 16-byte redzone region to the front of each allocated object, conceptually as follows:

```
malloc(SIZE) = lowfat_malloc(SIZE+16)+16
```

This region serves the dual purposes of: (1) the *redzone* memory, and (2) *shadow* memory to store object state/size meta information. Under this design, the state can be quickly retrieved at runtime using the low-fat pointer *base*(*ptr*) operation. Invalid access to redzone memory itself can be detected by calculating the distance from the accessed pointer and base address. This leads to the following alternative definition for *state*(*ptr*):

$$state_{lowfat}(ptr) = (ptr - base(ptr) < 16 \text{? Redzone} : *base(ptr))$$

The modified `OBJECT` layout for an allocation of `SIZE` bytes is illustrated in Figure 3. Here, a 16-byte redzone region is prepended to the `OBJECT`, which also serves as shadow memory for the `STATE/SIZE`. The $state_{lowfat}$ definition can be used to implement the (Redzone) check. For our application, the main advantage of $state_{lowfat}$ is that it implements redzones only using low-fat pointer operations, meaning that the *base*(*ptr*) operation can be shared between the (Redzone) and (LowFat) checks.

Finally, we note that a redzone region is only ever prepended to the start of objects, and never to the end. However, the redzone at the start of the *next* object in memory serves as a redzone at the end of the current object, as illustrated in Figure 3. This works even if the next object is unallocated.

*Implementing low-fat pointers at the binary level.* As we work at the binary level, the (LowFat) instrumentation is applied to *memory operands*. The general form of a memory operand is the highlighted portion of the following `mov` instruction (in AT&T syntax):

`mov %rax,` *seg*:*disp*(*base*,*idx*,*scale*)

A memory operand is essentially a 5-tuple, comprising:

1. A *segment register* (*seg*)
2. A 32bit signed integer *displacement* (*disp*)
3. A *base register* (*base*)
4. An *index register* (*idx*)
5. An integer $\{1, 2, 4, 8\}$ *scale* (*scale*)

Semantically, a memory operand represents the expression:

$$seg + disp + base + idx \times scale$$

One or more components may be omitted, in which case the following default values are used: zero (for *seg*, *disp*, *base*,

```
1  void check(void *ptr,
2             size_t i,
3             size_t len)
4  {
5      // STEP (1): Calculate memory access range
6      void *LB = ptr+i;
7      void *UB = ptr+i+len;
8
9      // STEP (2): Calculate object base address
10     size_t *BASE = NULL;
11     if (BASE == NULL)
12         BASE = base(ptr);// (LowFat) base
13     if (BASE == NULL)
14         BASE = base(LB); // (Redzone) base
15     if (BASE == NULL)
16         return;          // Non-fat pointer
17
18     // STEP (3): Get object state & size
19     size_t STATE = state(BASE);
20     size_t  SIZE = objectSize(BASE);
21
22     // STEP (4): Check for errors
23     if (SIZE > size(BASE)-16)
24         error();         // Corrupted metadata
25     if (STATE == FREE)
26         error();         // UaF check failed
27     if (LB < BASE+16)
28         error();         // LB check failed
29     if (UB > BASE+16+SIZE)
30         error();         // UB check failed
31 }
```

**Figure 4.** Pseudo-code for memory hardening checking. Here, the instrumented check takes a pointer (`ptr`), an index (`i`), and an access size (`len`) as input. The code on lines 11-12,23-24 is *optional*, and the code on lines 19-20,25-30 is *mergeable*. Here, we assume a redzone size of 16 bytes.

and *idx*) or one (for *scale*). We can implement the (LowFat) instrumentation for memory operands by using the following substitution: *ptr*=*base* and *i*=*disp*+*scale*×*idx*.

### 4.2 Complementary Instrumentation

The pseudo-code for the combined (Redzone)+(LowFat) check is shown in Figure 4. Here, we represent the check as a function, `check()`, that takes three arguments: a pointer `ptr` (representing an address, i.e. 64-bit address), an index/offset `i`, and a memory access size `len`. The `error()` function will be called if a memory error is detected. The `error()` function will abort execution (for hardening applications) or log the error (for bug finding applications).

The `check()` function is divided into several steps. Step (1), in lines 6-7, calculates the lower/upper bound of the memory access (LB/UB). Step (2), in lines 10-16, calculates the base address of the accessed object using the low-fat *base*() operation. This calculation is done in two steps: first, we attempt to calculate the base address from `ptr`, as in (LowFat). If that fails because `ptr` is a non-fat pointer, we calculate the base address from the accessed address itself LB, as in (Redzone). In essence, this implements a form of "fallback" checking: if `ptr` is a non-fat pointer, then we use LB to calculate the base pointer in order to enforce redzone-based protection. This covers the subtle case where a non-fat pointer overflows into the heap. If the resulting address LB is also a non-fat pointer, the `check()` function simply returns (line 16).

Step (3), in lines 19-20, retrieves the metadata that is stored at the base of the object, including the object's allocation `STATE` (Allocated/Free returned by `state()`) and the object's `malloc` `SIZE` (returned by `objectSize()`). Step (4), in lines 23-30, implements the various checks, including:

1. lines 23-24: validating `SIZE` against the low-fat pointer size *size*(`BASE`) for metadata hardening (see below);
2. lines 25-26: the *use-after-free* check;
3. lines 27-28: the *lower-bound* check;
4. lines 29-30: the *upper-bound* check;

Access to redzone memory is protected by the lower and upper bound checks. Note that *both* the lower and upper bounds of the access are checked. Other schemes, such as [10] only check the lower bound for efficiency reasons. However, this can lead to undetected overflows, which is especially dangerous when the metadata is stored in the redzone memory itself. In effect, the Figure 4 instrumentation schema implements accurate checking of both the lower and upper bounds against the original `malloc` size (`SIZE`). This means that overflows into padding (see `(padding)` from Figure 3) will also be detected.

*Metadata hardening.* Under the object layout of Figure 3, metadata is directly stored in the redzone memory prepended to the start of each object. This protects the metadata from direct modification by instrumented code. However, an attacker may still attempt to modify the metadata using a memory error in unprotected code, e.g., from an uninstrumented library. To mitigate this threat, lines 23-24 validates the stored `SIZE` against the (immutable) low-fat *size*(`BASE`) value, which limits potential secondary overflows to within the allocation padding.

*Optional code.* Some code is optional (at the user's discretion). For example, we can exclusively use LB to calculate the base pointer by removing the code on lines 11-12. This effectively disables (LowFat) checking in place of exclusive (Redzone) checking. Similarly, the check on lines 23-24 can be removed to disable metadata hardening, trading some security for better performance.

*Mergeable code.* The instrumentation can be made smaller by merging code. One idea is to combine the representation of the object's state and size. For example, we can encode the Allocated state by (`SIZE` > 0), and the Free by (`SIZE` = 0). This eliminates the need for a separate representation of both values and thereby simplifying the code on lines 19-20. Another optimization merges several checks into one. Since the Free state is represented by (`SIZE` = 0), the *use-after-free*

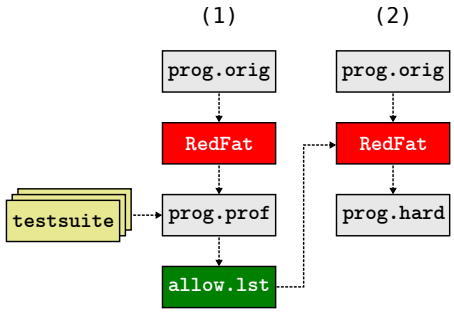


**Figure 5.** Illustration of profile-based false positive elimination. Step (1) is the *profiling phase*. Here, the original binary (`prog.orig`) is instrumented with a modified Figure 4 instrumentation that is designed to detect all memory access that passes the (LowFat) check. Next, the instrumented `prog.prof` binary is then run against a test suite to generate an allow-list (`allow.lst`) of all passing memory access that are unlikely to trigger false positives. Step (2) is the *production phase*. Here, the final hardened binary (`prog.hard`) is generated using the `allow.lst`. For this, all memory access from `allow.lst` is instrumented with the full (Redzone)+(LowFat) check, and all other memory access with the (Redzone)-only check.

check on lines 25-26 can be directly merged into the out-of-bounds check on lines 27-30. In other words, if `SIZE=0`, the bounds check will always fail, eliminating the need for a separate UaF check. Finally, the lower bound check on lines 27-28 can be merged into the upper bound check on lines 29-30 by taking advantage of integer underflow. The basic idea is to use the following alternative definition for `UB`:

```
void *UB =
    (void *)(uint32_t)(LB - (BASE+16))
        + BASE+16+len;
```

If the `LB` is within bounds, the calculated `UB` will be the same as the original value. Otherwise, an integer underflow will occur, resulting in a large value for `UB` which will fail the upper check on lines 29-30. Although this definition of `UB` requires more operations, in our experience the removal of an additional branch is worthwhile.

## 5 False Positive Mitigation

Complementary memory error protection inherits the benefits of both redzones and low-fat pointers, but can also inherit the limitations. Specifically, the low-fat pointer component of the Figure 4 schema may still detect false positives for *intentional* out-of-bounds pointers that are created by the programmer or compiler, see Problem #2 from Section 2.

Since static analysis is impractical, we instead propose a *profile-based* solution that attempts to find potential false positives using *dynamic analysis*. The basic idea is to first profile the binary in order to gather information on which memory operations are likely to be false positives. Next, this information is used to harden the binary for production. The analysis itself is heuristic-based, and is based on the following simple hypothesis:

> Each memory operation is *always* a false positive or *never* a false positive.

This hypothesis is based on common `C` anti-idioms that trigger false positives. For example, consider the anti-idiom `(array-K)[i]` to access array elements for positive constant `K` (e.g., snippet (c) from Section 2). Then the base pointer `array-K` will always be out-of-bounds of the `array` object, for all possible executions. Thus, the above memory access will always fail the (LowFat) check, regardless of the program input or execution path. Conversely, idiomatic array access `array[i]` will never trigger a false positive.

We can use this hypothesis to minimize false positives by using a simple two-phase workflow, as illustrated in Figure 5:

1. *Profiling Phase*: A *test binary* is generated by instrumenting the input binary with a variant of the Figure 4 check. The test binary is then run against a given *test suite* that exercises normal program behavior. Only the memory operations that are observed to always pass the (LowFat) check are added to an *allow-list*.
2. *Production Phase*: A hardened *production binary* is generated by instrumenting the input binary with (Redzone)+(LowFat) for memory operations in the *allow-list*, or (Redzone)-only otherwise. The production binary can then be deployed accordingly.

For example, the anti-idiom `(array-K)[i]` memory access will fail the (LowFat) check during the profiling phase, hence, will not be included in the generated *allow-list*. As will be discussed in Section 7, this anti-idiom accounts for most of the false positives observed in the SPEC benchmarks. All other observed normal (i.e., idiomatic) memory access will be added to the *allow-list*. In the production binary, the normal memory access will enjoy the full (Redzone)+(LowFat) protection, whereas anti-idiom memory access will be protected by the baseline (Redzone) check.

The allow-list generation step assumes that actual memory errors are rare. If an actual memory error is encountered during profiling, it may be incorrectly classified as a false positive. This risk is mitigated by the testing process itself, where unusual program behavior is more likely to be noticed by the user. Conversely, a given memory operation may only trigger a false positive sporadically (i.e., violating the hypothesis), and this may be missed during testing. However, our aim is to minimize false positives even if they cannot be completely eliminated. We will also validate our approach experimentally in Section 7.

As our approach is based on profiling, the quality of the generated allow-list depends on the quality of the test suite, especially in relation to *coverage*. This is an inherent limitation of profile-based analysis. Nevertheless, under the philosophy of opportunistic hardening, even partial coverage is

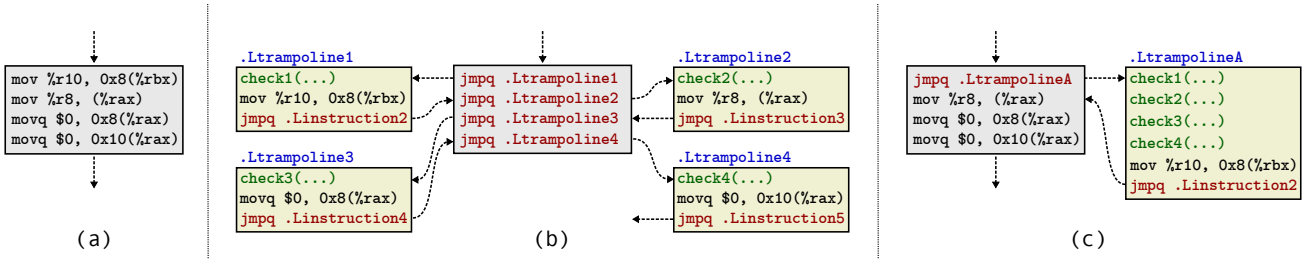


**Figure 6.** Illustration of the *check batching* optimization. Here, (a) is the original basic block, (b) instruments each instruction with its own trampoline, and (c) batches all instrumented checks into a single trampoline.

better than none. Furthermore, automated coverage-guided testing tools, such as the *American Fuzzy Lop* (AFL) [1] over binaries [16], can be used to boost coverage. Finally, we note that all memory access will still be protected by the (REDZONE) check, regardless of the allow-list. Thus, even an incomplete allow-list can offer a stronger protection than the current state-of-the-art.

## 6 Optimization

Another design goal is to minimize the performance impact of the instrumented binary. We can mitigate much of the overhead by designing three main optimizations: *check elimination*, *batching* and *merging*.

*Check elimination.* The *check elimination* optimization removes instrumentation on instructions that will never fail the Figure 4 check. For example, consider the instruction: ⟨`movq %rax,0x601000`⟩. Under the low-fat `malloc`, the pointer `0x601000` will always be non-fat, and thus the instruction can never fail Figure 4. We can generalize this to any memory operand that provably cannot reach heap memory. This includes any memory operand:

1. With no *index register*; and
2. With no *base register*, or a base register not within $\pm 2GB$ (minimum/maximum *displacement*) from heap memory.

Assuming the default locations for executable and stack memory (at least $\pm 2GB$ from heap memory), the second rule can also exclude memory operands using the *instruction pointer* (`%rip`) or *stack pointer* (`%rsp`) as the base register. This eliminates many instrumented checks for instructions that access the global or stack memory.

*Check batching.* Trampoline-based binary rewriting is highly scalable, but also introduces runtime overheads in the form of jumps to/from trampolines. One direct way to reduce overhead is to minimize the number of trampolines and jumps. This can be achieved by "batching" multiple checks for consecutive memory operations into a single trampoline whenever possible.

**Example 2** (Check Batching)**.** Consider the following instruction sequence which is part of the same basic block:

```
.Linstruction1: mov %r10,0x8(%rbx)
.Linstruction2: mov %r8,(%rax)
.Linstruction3: movq $0x0,0x8(%rax)
.Linstruction4: movq $0x0,0x10(%rax)
```

The instruction sequence is illustrated in Figure 6(a). Under the baseline instrumentation schema, each memory operation will need to be checked separately, leading to the spaghetti-like code illustrated in Figure 6 (b). This fragments the executable code over 1+4 = 5 non-contiguous locations (the main program + four trampolines), and uses a total of 4+4 = 8 jumps. The loss of locality and additional control-flow redirections can result in a non-trivial runtime overhead. An optimized version is shown in Figure 6 (c). The optimized version uses a single trampoline (`.LtrampolineA`) that is invoked only once at first memory operation in the basic block. This version uses a total of 1+1 = 2 locations and 1+1 = 2 jumps. □

In order to apply the *check batching* optimization, memory access instructions are grouped into sequences ⟨`I1`, ..., `In`⟩ satisfying the following properties:

1. The *ordering* of ⟨`I1`, ..., `In`⟩ is the same as the program;
2. Each `I`$i$ belongs to the *same basic block* as `I1`; and
3. Each `I`$i$ can be *reordered* to position `I1` without affecting the memory access location.

If satisfied, the instrumented checks for ⟨`I1`, ..., `In`⟩ can be combined into a single trampoline which is invoked exactly once before `I1`, as illustrated by Figure 6 (c).

A static binary analysis is necessary to find sequences of memory access instructions that can be batched. The *reordering* property can be established with a simple static register usage analysis. The *same basic block* property can be established using a control-flow (basic block) recovery analysis. Although precise control-flow information is not perfectly recoverable in the general case, an *over* approximation will be sufficient for the purpose of optimization. For example, suppose an instruction `I`$i$ is misidentified as a jump/call target, then two smaller batches (⟨`I1`, ..., `I`$i-1$⟩, ⟨`I`$i$, ..., `In`⟩) will be used instead of the optimal sequence ⟨`I1`, ..., `In`⟩. In other

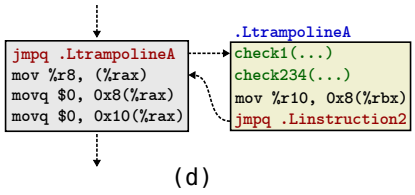


**Figure 7.** Illustration of the *check merging* optimization.

words, an over approximation may result in suboptimal batch sizes, but otherwise does not affect correctness. Similarly, if the analysis were to yield an *under* approximation (i.e., missed targets due to anti-idiomatic calls/jumps), then the coverage of the protection may be reduced (i.e., missed instrumented checks). However, this case also does not affect correctness, and our implementation uses a conservative control-flow recovery analysis that errs on the side of over approximation rather than under.

*Check merging.* We can further improve the performance of batching by merging instrumented checks for sufficiently similar memory operations. For example, considering the instruction sequence from Example 2. The instructions at `.Linstruction{2,3,4}` all use a memory operand of the same basic form:

$$disp\texttt{(\%rax)} \text{ where } disp \in \{\texttt{0x0}, \texttt{0x8}, \texttt{0x10}\} \quad \text{(MemOps)}$$

Using *check merging*, we can treat the instructions as a single "merged" memory access over the combined address range.

We can generalize this idea as follows. For the purpose of the instrumentation (Figure 4), the *memory access range* *LB*...*UB* can be defined as a pair of memory operands:

$$\begin{aligned} LB &= seg : disp(base, idx, scale) \\ UB &= seg : disp{+}len(base, idx, scale) \end{aligned}$$

Here, *len* is the size of the memory access. The displacement effectively defines a *partial ordering* as follows:

$$seg : disp_1(base, idx, scale) < seg : disp_2(base, idx, scale) \text{ iff } disp_1 < disp_2$$

Using this partial ordering, we can merge the checks for memory operands that share the same segment, base, index and scale. For example, in (MemOps) the memory operands only differ by the *displacement*. Thus, we can define the following *merged bounds* over all three memory access operations:

$$\begin{aligned} LB_{merged} &= \texttt{0x0(\%rax)} \\ UB_{merged} &= \texttt{0x10+8(\%rax)} \end{aligned}$$

The merged bounds $LB_{merged}...UB_{merged}$ need only be checked once rather than three times. The *check merging*-optimized version of Example 2 is illustrated in Figure 7 (d).

*Additional low-level optimizations.* The trampoline implementations of the (LowFat) and (Redzone) checks are implemented using optimized `x86_64` assembly code. In general, the trampoline code needs 3-4 scratch registers and must also preserve the flags register (`%eflags`). It is also common for some registers to be *clobbered* after the instrumentation point, in which case the corresponding values need not be preserved. Our implementation uses a simple static analysis to determine which registers (if any) are *clobbered*, and specializes the trampoline code accordingly.

## 7 Experiments

We have implemented RedFat on top of the E9Patch binary rewriting tool. RedFat takes a binary file as input, then outputs a hardened binary with all relevant memory operations instrumented. The hardened binary can then be used as a drop-in replacement for the original. To enable hardening, a `libredfat.so` runtime library (which includes the low-fat `malloc` implementation) must be `LD_PRELOAD`'ed. Our implementation supports all of the optimizations described in Section 6. The experiments are run on an Intel Xeon Silver 4114 CPU (clocked at 2.20GHz) with 32GB of RAM. Each benchmark/program is compiled using `gcc`/`g++`/`gfortran` compiler version 5.4.0.

### 7.1 Performance

We have evaluated the performance of RedFat against the full SPEC CPU2006 benchmark suite [18], including `C`, `C++`, and Fortran programs. All benchmarks are compiled with the `-O2` optimization level, with the exception of `gamess`, which is compiled with `-O1`. This is to work around an existing and documented miscompare (incorrect output with `-O2`) with the uninstrumented `gamess` benchmark at higher optimization levels.[1] To evaluate the performance of RedFat, we compare against the original uninstrumented binaries as well as Valgrind Memcheck [25]—a scalable, state-of-the-art memory error detection tool for binary code. We note that Memcheck is based on heavyweight dynamic binary instrumentation rather than static rewriting, and primarily intended for the testing/debugging use case rather than hardening. To make the comparison as fair as possible, we disable Memcheck's *leak checking* (`-leak-check=no`) and *uninitialized read checking* (`-undef-value-errors=no`). To validate the false positive minimization workflow (Section 5), the benchmarks were instrumented using an *allow-list* generated from the SPEC `train` workload used as the test suite.

The experimental results for the SPEC `ref` workload are shown in Table 1. Here, the *unoptimized* column represents the performance of RedFat-instrumentation without any optimization enabled. The *+elim*, *+batch*, *+merge* columns progressively enable the *check elimination*, *batching* and *merging* optimizations respectively. The RedFat tool also supports the removal of some checks for better performance. The *-size* and *-reads* columns remove the size metadata hardening and read checking respectively. The latter can be justified

[1] See https://gcc.gnu.org/gcc-4.8/changes.html

**Table 1.** Performance of RedFat and Memcheck on the SPEC CPU2006 benchmark suite. The C / C++ / Fortran benchmarks are highlighted with different colors. Legend: *coverage* denotes the percentage of memory operands covered by full (Redzone)+(LowFat) check (others are covered by (Redzone)-only). Baseline denotes the original binaries with no instrumentation; *+elim* enables check elimination; *+batch* enables check batching; *+merge* enables check merging; *-size* disables size metadata hardening; *-reads* disables read instrumentation; NR=not run due to known issues. Baseline column lists the timing in seconds; other columns (except *coverage* column) list the slow-down as a factor compared to the Baseline. Memcheck is invoked with `-leak-check=no` and `-undef-value-errors=no`. The main results are highlighted in **bold**.

| Binary | *coverage* | Baseline *(seconds)* | RedFat *unoptimized* | *+elim* | *+batch* | RedFat *fully optimized* | | | Memcheck |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | *+merge* | *-size* | *-reads* | |
| *perlbench* | 88.9% | 286 | 12.83× | 9.82× | 8.26× | 7.46× | 6.75× | 2.26× | 29.22× |
| *bzip2* | 97.0% | 452 | 7.38× | 6.52× | 5.99× | 5.52× | 4.75× | 1.98× | 7.36× |
| *gcc* | 66.0% | 242 | 5.34× | 4.49× | 4.21× | 3.92× | 3.52× | 1.70× | 14.32× |
| *mcf* | 98.7% | 280 | 3.69× | 3.64× | 3.33× | 2.86× | 2.67× | 1.13× | 4.74× |
| *gobmk* | 90.7% | 441 | 6.83× | 4.62× | 3.92× | 3.75× | 3.58× | 1.56× | 19.84× |
| *hmmer* | 48.0% | 341 | 17.88× | 15.66× | 12.94× | 10.67× | 9.52× | 2.20× | 12.07× |
| *sjeng* | 98.6% | 496 | 7.48× | 5.84× | 4.94× | 4.75× | 4.57× | 1.51× | 20.59× |
| *libquantum* | 100.0% | 309 | 3.32× | 3.33× | 3.39× | 3.38× | 2.80× | 1.80× | 4.73× |
| *h264ref* | 20.0% | 456 | 11.54× | 8.87× | 7.58× | 7.19× | 6.34× | 1.52× | 21.71× |
| *omnetpp* | 62.8% | 306 | 3.56× | 3.42× | 3.00× | 2.89× | 2.62× | 1.40× | 12.40× |
| *astar* | 99.7% | 389 | 4.84× | 4.06× | 3.75× | 3.52× | 3.23× | 1.25× | 7.82× |
| *xalancbmk* | 78.9% | 195 | 7.28× | 6.47× | 6.14× | 6.02× | 5.03× | 1.13× | 22.34× |
| *milc* | 99.4% | 456 | 3.98× | 3.60× | 3.59× | 1.91× | 1.80× | 1.15× | 4.68× |
| *lbm* | 98.8% | 236 | 5.44× | 4.42× | 3.79× | 1.31× | 1.23× | 1.05× | 7.15× |
| *sphinx3* | 99.5% | 502 | 7.36× | 7.06× | 6.86× | 6.60× | 5.91× | 1.20× | 12.85× |
| *namd* | 100.0% | 349 | 7.19× | 5.95× | 5.29× | 2.63× | 2.44× | 1.28× | 7.77× |
| *dealII* | 81.7% | 282 | 7.70× | 6.70× | 6.45× | 5.70× | 4.93× | 1.71× | *NR* |
| *soplex* | 96.4% | 212 | 5.00× | 4.83× | 4.57× | 4.09× | 3.68× | 1.59× | 6.24× |
| *povray* | 99.9% | 139 | 10.91× | 8.86× | 7.12× | 5.35× | 4.88× | 1.81× | 36.96× |
| *bwaves* | 85.2% | 344 | 7.54× | 6.47× | 6.25× | 6.10× | 5.57× | 1.26× | 10.87× |
| *gamess* | 43.0% | 680 | 9.04× | 6.17× | 5.40× | 4.34× | 4.31× | 1.98× | 15.41× |
| *zeusmp* | 23.2% | 319 | 4.85× | 3.89× | 3.42× | 2.41× | 2.42× | 1.50× | *NR* |
| *gromacs* | 83.3% | 270 | 7.40× | 3.76× | 3.50× | 2.28× | 2.07× | 1.27× | 12.72× |
| *cactusADM* | 99.9% | 460 | 8.97× | 2.70× | 2.56× | 2.30× | 2.11× | 1.13× | 14.43× |
| *leslie3d* | 100.0% | 262 | 9.38× | 8.99× | 8.63× | 7.86× | 7.00× | 2.66× | 11.23× |
| *calculix* | 28.7% | 760 | 4.74× | 4.47× | 5.09× | 5.08× | 4.68× | 1.24× | 10.83× |
| *GemsFDTD* | 98.7% | 331 | 7.27× | 6.67× | 6.39× | 5.36× | 4.93× | 2.13× | 8.35× |
| *tonto* | 95.0% | 454 | 5.85× | 4.03× | 3.92× | 3.27× | 2.90× | 1.61× | 14.81× |
| *wrf* | 27.0% | 420 | 8.54× | 8.07× | 7.82× | 6.93× | 6.19× | 2.38× | 13.98× |
| Geometric mean | 72.6% | 345 | 6.78× | 5.50× | 5.06× | 4.18× | **3.81**× | **1.55**× | 11.76× |

for hardening applications where many common exploits require a write operation as part of the attack sequence. The *-size* column represents the performance of RedFat that most closely aligns with Valgrind Memcheck (last column). On average, RedFat has less overhead than Memcheck, with a 3.81× slowdown (the *-size* column) for RedFat, versus a 11.76× slowdown for Memcheck. For *-reads*, the performance is further reduced to 1.55×, which is fast enough for many practical applications while also protecting the binary against many common attacks.

Aside from lower performance overheads, RedFat is also able to instrument more binaries. For Memcheck, the `dealII` and `zeusmp` benchmarks failed to run because of known issues documented in the Valgrind repository.[2] These failures are due to Valgrind being unable to handle large data segments, as well as only supporting 64-bit floating-point values (instead of the full 80-bits supported by `x87` instructions). RedFat does not have such limitations, and is able to instrument and run the full SPEC suite, including all C/C++/Fortran benchmarks.

*Coverage from profile-based hardening.* The *coverage* column shows the percentage of dynamically reachable memory access instructions that were covered by the (Redzone)+

[2] See `docs/internals/SPEC-notes.txt`

(LowFat) check, versus those covered by (Redzone)-only. The coverage is based on the profile-based false positive minimization workflow as explained in Section 5, and by using the SPEC `train` workload to generate the allow-list. Overall, the mean coverage is 72.6% for full (Redzone)+(LowFat) protection. In contrast, the current state-of-art memory error detection tools for binaries are based on (Redzone)-only checking, and therefore have an effective 0% coverage. Finally, we note that the coverage result could be further improved by running additional tests during the profiling phase (beyond the `train` workload). Nevertheless, we present the results "as-is" for a consistent methodology.

*Detected errors.* Both the RedFat and Valgrind Memcheck tools detect out-of-bounds read errors in the `calculix` and `wrf` Fortran benchmarks. The `calculix` benchmark has 4 read underflows in the `main()` function, all of which are of the form `array[-1]`. The `wrf` benchmark has a read overflow in the `interp_fcn()` function.

*Optimizations.* The experimental results in Table 1 show that each optimization from Section 6 is significant. Specifically, check elimination (+*elim*) improves the overhead from 6.78× to 5.50×, check batching (+*batch*) further improves the overhead to 5.06×, and check merging (+*merge*) further improves the overhead to 4.18×. Overheads can be further improved by removing checks, to 3.81× and 1.55× with (-*size*) and (-*reads*) respectively.

*False positives.* We also reran the SPEC benchmarks with full (Redzone)+(LowFat) checking enabled on all memory access (i.e., no profile-based allow-list). In this case, some false positives were reported, including: 1 from `perlbench`, 14 from `gcc`, 1 from `gobmk`, 1 from `povray`, 5 from `bwaves`, 3 from `gromacs`, 32 from `GemsFDTD`, 26 from `wrf`, and 2 from `calculix`. Some false positives are known and have been reported previously [10]. Most of the false positives were caused by accessing an array base pointer of the form (`array-K`) for some `K`. Although this is an anti-idiom and *undefined behavior* in C/C++, it is natively supported in Fortran where the base index of an array need not be zero. For example, the following is a false positive from the `wrf` benchmark:

```
REAL,  DIMENSION(its:ite, kts:kte, 2) :: fqy
  ...
fqy(i, k, jp1) = ...
```

The code defines a Fortran multidimensional array with the index ranges `its...ite`, `kts...kte`, and 1..2. The `gfortran` compiler essentially normalizes the array base pointer to be `fqy-K`, creating an intentional out-of-bounds pointer. Such an out-of-bounds pointer will be detected as a false positive when accessed using the (LowFat) check. False positives of this form can be readily detected during the profile-based analysis.

| CVE entry | Memcheck | RedFat |
|---|---|---|
| CVE-2007-3476 (php) | 0/1 (0%) | 1/1 (100%) |
| CVE-2016-1903 (php) | 0/1 (0%) | 1/1 (100%) |
| CVE-2012-4295 (wireshark) | 0/1 (0%) | 1/1 (100%) |
| CVE-2016-2335 (7zip) | 0/1 (0%) | 1/1 (100%) |
| CWE-122-Heap-Buffer (Juliet) | 0/480 (0%) | 480/480 (100%) |

**Table 2.** CVEs/CWEs for non-incremental bounds errors.

### 7.2 Non-incremental Overflows

The main advantage of the (LowFat) check is the ability to detect non-incremental out-of-bounds errors. To evaluate the effectiveness of our approach, we create a test suite that includes (1) four real-world *Common Vulnerabilities and Exposures* (CVEs) with non-incremental overflows, and (2) the subset of Juliet test suite [26] with non-incremental overflows. The CVEs originate from vulnerable versions of `php` (a general-purpose scripting language), `wireshark` (a network protocol analyzer), and `7zip` (a file archiver). For each test case, we model an offset that is controlled by the attacker, meaning that an adjacent heap object will be accessed. We tested both RedFat and Valgrind Memcheck.

A summary of the results is shown in Table 2. Our experiments show that RedFat is able to detect all non-incremental errors from all test cases, which is the expected result under RedFat's design. On the other hand, Valgrind Memcheck, which uses (Redzone)-only checking, cannot detect these errors for suitably crafted offsets. Our results show that non-incremental errors do exist in practice, and can be used to bypass (Redzone)-only checking for an attacker that is aware of the defense. Thus, the stronger protection of (Redzone)+(LowFat) is justified in these cases.

### 7.3 Scalability

Scalability is important for practical tools, especially for hardening applications, where the tool should mitigate existing bugs and not introduce new bugs. To evaluate scalability, we use RedFat to instrument the Google Chrome [17] web browser version 80.0.3987.132 with (Redzone)+(LowFat) checking for all write operations. The closed-source Chrome binary is ~149MB, which is much larger than the SPEC2006 binaries combined, and is considered challenging for binary instrumentation. To measure the performance, we use the Kraken browser benchmark [19] and the results are shown in Figure 8. Here, RedFat is able to successfully instrument Chrome and the resulting binary runs stable. Overall, the instrumented binary exhibits a 1.28× overhead (geometric mean) under the Kraken benchmarks.

### 7.4 Limitations

RedFat is designed to detect out-of-bounds and use-after-free errors. However, RedFat cannot detect other kinds of errors such as *sub-object bounds errors* (i.e., overflow an array

member of a `struct`) or *type confusion* (i.e., bad casts). The detection of these kinds of error depends on type information (e.g., `struct` layouts) which is generally unavailable at the binary-level. Similarly, the layout of global and stack objects is also generally unavailable at the binary-level, meaning that RedFat is focused on protecting heap objects only. That said, most modern binaries partly mitigate stack write overflows using *stack canaries*.

RedFat is designed to be compatible with most off-the-shelf Linux ELF binaries. However, since RedFat is based on E9Patch [9], RedFat inherits the limitations of the latter, such as restrictions with self-modifying code. RedFat also depends on replacing the default `glibc malloc` implementation in the instrumented binary. However, this approach may not be effective for binaries which use their own *Custom Memory Allocators* (CMAs), or are otherwise designed to minimize heap allocation altogether (e.g., by using stack-/global memory). Finally, since RedFat is based on static binary rewriting, only the binaries that are explicitly instrumented by RedFat will be protected. For example, if the main program (e.g., `main`) is instrumented by RedFat, but a dynamic library dependency (e.g., `libdependency.so`) is not, then only the former will enjoy memory error protection at runtime. RedFat supports both ELF executables and shared objects, meaning that it is possible to separately instrument both the main program and any dynamic library dependency as required.

## 8 Related Work

Many different source-level and binary-level memory safety tools have been proposed [32]. Most [5, 8, 15, 25, 29] are based on some variant of *redzone* protection, but others are based on (low) fat pointers [10, 13, 24] or some other metadata tracking scheme [2, 12, 14, 23, 24, 38].

Heavyweight *Dynamic Binary Instrumentation* tools, such as DrMemory [5] and Valgrind Memcheck [25, 30], compete with RedFat in terms of scalability. For example, Memcheck is able to run most SPEC2006 benchmarks, and was able to load Chrome to the start page under our testing. However, scalability is not perfect. Furthermore, these tools only support a redzone-based checking. Memcheck also has high overheads, at 11.76× even with some features disabled, which is too slow for most applications.

Like RedFat, RetroWrite's Binary ASAN [8] also implements memory error detection using static binary rewriting. RetroWrite's static binary rewriting methodology is based on the notion of *reassemblable disassembly* [34, 35]. This is feasible for some PIC binaries compiled from `C` code (`C++` binaries are not supported). Unlike E9Patch, RetroWrite can insert the instrumentation *inline* because of the assumptions made. When applicable, this approach is generally faster than trampoline-based binary rewriting. However, RetroWrite's Binary ASAN only implements redzone-only protection and the motivating application is fuzz testing rather than hardening.

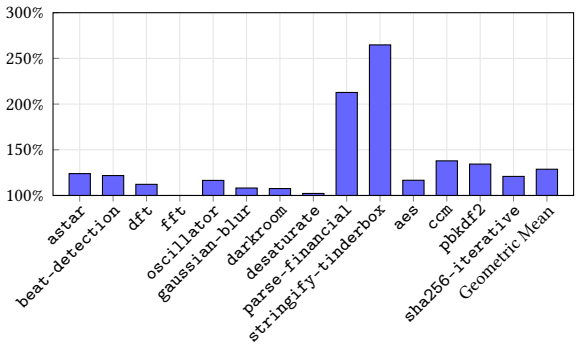


**Figure 8.** Chrome overhead using the Kraken benchmarks.

Tools such as DieHard [3], DieHarder [27], FreeGuard [31], ASLR [28] are based on randomization rather than binary rewriting and instrumentation. Such tools offer a probabilistic defense, and are designed to make attacks more difficult rather than impossible. While our current implementation also incorporates basic heap randomization, a full integration is orthogonal to the ideas here and left as future work.

## 9 Conclusion

In this paper, we have presented a binary instrumentation system for detecting memory errors such as buffer overflows and use-after-free. Our approach is primarily designed towards the *hardening* use cases, rather than testing/debugging. As such, we use a stronger memory error checking—based on complementary redzone and low-fat pointer protection—allowing for the detection of "more" memory errors compared to each individual protection used alone. However, adapting low-fat pointers to the binary level may result in false positives due to the ambiguous nature of memory access without type/symbol information. We present a solution, in the form of a profile-based analysis, that can mitigate most potential false positives in practice.

We have implemented our approach in the form of the RedFat tool, which is scalable, language agnostic (e.g., `C`/`C++`/Fortran binaries), applicable (e.g., PIC versus non-PIC binaries), and achieves a much lower overhead compared to the current state-of-the-art; all while offering a stronger defense against memory errors.

## Acknowledgements

This work was partially supported by the National Satellite of Excellence in Trustworthy Software Systems, funded by the National Research Foundation (NRF) Singapore under the National Cybersecurity R&D (NCR) programme; and the Ministry of Education, Singapore (Grant No. MOE2018-T2-1-142). We also thank our shepherd, Valerio Schiavoni, as well as the anonymous referees for their comments.

## References


[1] AFL. 2022. https://github.com/google/AFL.

[2] P. Akritidis, M. Costa, M. Castro, and S. Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense Against Out-of-Bounds Errors. In *USENIX Security Symposium*. USENIX.

[3] E. Berger and B. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Programming Language Design and Implementation*. ACM.

[4] D. Bruening, T. Garnett, and S. Amarasinghe. 2003. An Infrastructure for Adaptive Dynamic Optimization. In *Code Generation and Optimization*. IEEE.

[5] D. Bruening and Q. Zhao. 2011. Practical Memory Checking with Dr. Memory. In *Code Generation and Optimization*. IEEE.

[6] B. Chamith, B. Svensson, L. Dalessandro, and R. Newton. 2017. Instruction Punning: Lightweight Instrumentation for x86-64. In *Program Design and Implementation*. ACM.

[7] Chromium. 2022. https://www.chromium.org/Home/chromium-security/memory-safety.

[8] S. Dinesh, N. Burow, D. Xu, , and M. Payer. 2020. RetroWrite : Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *Security and Privacy*. IEEE.

[9] G. Duck, X. Gao, and A. Roychoudhury. 2020. Binary Rewriting without Control Flow Recovery. In *Programming Language Design and Implementation*. ACM.

[10] G. Duck and R. Yap. 2016. Heap Bounds Protection with Low Fat Pointers. In *Compiler Construction*. ACM.

[11] G. Duck and R. Yap. 2018. An Extended Low Fat Allocator API and Applications. https://arxiv.org/abs/1804.04812.

[12] G. Duck and R. Yap. 2018. EffectiveSan: Type and Memory Error Detection using Dynamically Typed C/C++. In *Programming Language Design and Implementation*. ACM.

[13] G. Duck, R. Yap, and L. Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers. In *Network and Distributed System Security Symposium*. Internet Society.

[14] F. Eigler. 2003. Mudflap: Pointer Use Checking for C/C++. In *GCC Developer's Summit*.

[15] A. Fioraldi, D. D'Elia, and L. Querzoni. 2020. Fuzzing Binaries for Memory Safety Errors with QASan. In *Secure Development*. IEEE.

[16] X. Gao, G. Duck, and A. Roychoudhury. 2021. Scalable Fuzzing of Program Binaries with E9AFL. In *Automated Software Engineering*. IEEE.

[17] Google Chrome 2022. Google Chrome Web Browser. https://www.google.com/chrome/.

[18] J. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *Computer Architecture News* 34, 4 (2006).

[19] Kraken. 2022. https://wiki.mozilla.org/Kraken.

[20] LowFat 2022. https://github.com/GJDuck/LowFat.

[21] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Programming Language Design and Implementation*. ACM.

[22] Microsoft. 2019. Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape. https://github.com/Microsoft/MSRC-Security-Research/.

[23] S. Nagarakatte, Z. Santosh, M. Jianzhou, M. Martin, and S. Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Programming Language Design and Implementation*. ACM.

[24] G. Necula, S. McPeak, and W. Weimer. 2002. CCured: Type-Safe Retrofitting of Legacy Code. In *Principles of Programming Languages*. ACM.

[25] N. Nethercote and J. Seward. 2007. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *Programming Language Design and Implementation*. ACM.

[26] NIST. 2022. *Juliet Test Suite for C/C++ v1.3*. https://samate.nist.gov/SARD/testsuite.php

[27] G. Novark and E. Berger. 2010. DieHarder: Securing the Heap. In *Computer and Communications Security*. ACM.

[28] Pax. 2022. Address Space Layout Randomization. http://pax.grsecurity.net/docs/aslr.txt.

[29] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Annual Technical Conference*. USENIX.

[30] J. Seward and N. Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *Annual Technical Conference*. USENIX.

[31] S. Silvestro, H. Liu, C. Crosser, Z. Lin, and T. Liu. 2017. FreeGuard: A Faster Secure Heap Allocator. In *Computer and Communications Security*. ACM.

[32] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. 2019. SoK: Sanitizing for Security. In *Security and Privacy*. IEEE.

[33] V. Veen, N. Sharma, L. Cavallaro, and H. Bos. 2012. Memory Errors: The Past, the Present, and the Future. In *Research in Attacks, Intrusions, and Defenses*.

[34] S. Wang, P. Wang, and D. Wu. 2015. Reassembleable Disassembling. In *Security Symposium*. USENIX.

[35] S. Wang, P. Wang, and D. Wu. 2016. UROBOROS: Instrumenting Stripped Binaries with Static Reassembling. In *Software Analysis, Evolution, and Reengineering*. IEEE.

[36] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl. 2019. From Hack to Elaborate Technique - A Survey on Binary Rewriting. *Computing Surveys* 52, 3 (2019).

[37] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. Wu, J. Yang, and V. Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. ACM.

[38] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and Wouter W. Joosen. 2010. PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs. In *Information, Computer and Communications Security*. ACM.