Delta Pointers: Buffer Overflow Checks Without the Checks

Taddeus Kroes*
Vrije Universiteit Amsterdam

Koen Koning*
Vrije Universiteit Amsterdam

Erik van der Kouwe
Leiden University

Herbert Bos
Vrije Universiteit Amsterdam

Cristiano Giuffrida
Vrije Universiteit Amsterdam

ABSTRACT

Despite decades of research, buffer overflows still rank among the most dangerous vulnerabilities in unsafe languages such as C and C++. Compared to other memory corruption vulnerabilities, buffer overflows are both common and typically easy to exploit. Yet, they have proven so challenging to detect in real-world programs that existing solutions either yield very poor performance, or introduce incompatibilities with the C/C++ language standard.

We present Delta Pointers, a new solution for buffer overflow detection based on efficient *pointer tagging*. By carefully altering the pointer representation, without violating language specifications, Delta Pointers use existing hardware features to detect both contiguous and non-contiguous overflows on dereferences, without a single check incurring extra branch or memory access operations. By focusing on buffer overflows rather than other vulnerabilities (e.g., underflows), Delta Pointers offer a unique checkless design to provide high performance while still maintaining compatibility. We show that Delta Pointers are effective in detecting arbitrary buffer overflows and, at 35% overhead on SPEC, offer much better performance than competing solutions.

CCS CONCEPTS

• **Security and privacy → Systems security**; **Software and application security**;

KEYWORDS

Bounds Checking, Memory Safety, Pointer Tagging, LLVM

ACM Reference Format:
Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. 2018. Delta Pointers: Buffer Overflow Checks Without the Checks. In *EuroSys '18: Thirteenth EuroSys Conference 2018, April 23–26, 2018, Porto, Portugal*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3190508.3190553

*Equal contribution joint first authors

EuroSys '18, April 23–26, 2018, Porto, Portugal

ACM ISBN 978-1-4503-5584-1/18/04…$15.00
https://doi.org/10.1145/3190508.3190553

1 INTRODUCTION

Almost 30 years after the Morris Worm famously used a buffer overflow bug in `fingerd`, such vulnerabilities are still rife in modern C/C++ binaries. Ever since, researchers have searched for ways to automatically detect and prevent the triggering of buffer overflows. Unfortunately, existing solutions suffer from either unacceptable overheads or poor compatibility. In this paper, we add an interesting new point in the design space to detect both contiguous and non-contiguous buffer overflows, with less overhead than comparable solutions (35% performance overhead on SPEC CPU2006 and negligible memory overhead).

Aiming for a *practical* defense against the most prevalent attacks, we limit our focus to buffer *over*flow vulnerabilities whereby attackers can overwrite memory after the end of the buffer, rather than *under*flow vulnerabilities which are far less common [28][1]. On the other hand, since non-contiguous overflows are now more common than contiguous overflows in real-world exploits [29], simply surrounding all buffers with so-called red zones (as used by AddressSanitizer [37] among others) no longer suffices, since doing so prevents only contiguous ones. Instead, we strive for a solution that prevents all types of overflow and works "out of the box" on programs written in standard C code, simply by recompiling the code with a specific compiler flag.

Bounds checking is one of the oldest and most common defenses against buffer overflows. By recording the bounds of an object with each pointer, defenses can insert run-time checks to verify that the pointer still falls in the valid range upon dereference. Sadly, bounds checking is still costly due to metadata management, but especially because of the checks. Reading the bounds data from memory, verifying the validity of the pointer, and optionally branching if the pointer is temporarily out of bounds or missing metadata due to uninstrumented code all incur significant overhead. As a consequence, researchers during the past three decades have focused on improving both the performance of such systems and their compatibility with the standard. Although the solutions have significantly improved over time, state-of-the-art systems still suffer from compatibility issues and non-practical overheads. For example, the fastest contiguous and non-contiguous buffer overflow detector today, Low-Fat Pointers [12], still incurs over 50% performance overhead (SPEC CPU2006) and yet cannot automatically support arbitrary off-by-one pointers allowed by the C/C++ standard and required by many real-world programs [7].

To address these issues, we propose Delta Pointers, a new approach to detect buffer overflows. The Delta Pointers design is different from all existing approaches in that it implicitly invalidates

[1] Anecdotally, although labels in the CVE database are not very precise, buffer overflows outnumber underflows by almost two orders of magnitude in such database: https://cve.mitre.org/

pointers upon going out of bounds (providing good performance) and revalidates them when going back in-bounds (providing good compatibility). Specifically, we show that our solution is significantly faster than existing solutions while providing compatibility with the C standard (and we discuss remaining compatibility issues in detail). We ensure that any dereference of an invalid pointer leads to an automatic crash enforced by the hardware—similar to how a dereference of a kernel pointer in user space leads to a crash. As a result, Delta Pointers *eliminate any need for additional memory accesses or branches*, pushing the overflow check to the hardware and making pointer dereferences very efficient.

Intuitively, we utilize pointer tagging and ensure that any arithmetic on the pointer that makes it point beyond the buffer's upper bound, will result in setting the most significant bit in the 64 bit address, while any arithmetic that makes it point below the upper bound, will unset this bit. Prior to a dereference, we mask out all other bits of the tag. If the most significant bit is not set, the pointer is valid, but a dereference of a pointer where this bit is set will immediately crash the application because such addresses are non-canonical, triggering a fault in the processor's memory management unit.

Our approach makes minimal assumptions and is portable across 64-bit architectures. It works with existing hardware and most C/C++ programs out of the box (and we elaborate on possible issues in later sections). While pointer tagging itself provides good compatibility, we outline some challenges faced by practical implementations which have not been detailed by previous pointer tagging-based solutions [25, 26].

To summarize, our contributions are:

- A novel, fully automated, design to detect buffer overflows based on pointer tags that automatically invalidates out-of-bounds pointers.
- An analysis of the practicality of pointer tagging for arbitrary C/C++ applications.
- An LLVM-based prototype of our design, evaluated with respect to effectiveness and performance. Our results show that Delta Pointers can detect the dominant class of spatial memory error vulnerabilities with competitive compatibility and overheads (35% slowdown on SPEC and negligible memory overhead).

2 BACKGROUND

In C and C++, a programmer can allocate a *buffer object*, which is a sequence of bytes somewhere in the address space. For instance, variables on the stack are implicitly allocated on function entry and a programmer can use `malloc` or `new` to explicitly allocate a buffer object on the heap. These objects are referenced through *pointers*, which point to the memory location of an object, or a location therein. The C standard [21] specifies that pointer arithmetic only produces defined behavior if the resulting pointer points inside the same object or at the element directly past its end. In practice, compilers produce invalid *out-of-bounds pointers* without warning. Without any security measures, such pointers can be used to access any object in the address space, regardless of which buffer the original pointer pointed to.

For example, a call such as `ptr = malloc(16)` allocates a new object on the heap and returns a pointer to its beginning. An offset of exactly 16 would yield a pointer to the end of the object, which is valid but results in undefined behavior upon dereference. Any dereference of an index smaller than 0 (e.g., `ptr[-10]`) is an underflow, and any dereference of index 16 or larger is an overflow. It is common for programs to (benignly) create pointers pointing outside the referent object [7]. However, if malicious users can lure the program into dereferencing such pointers as a result of a buffer overflow vulnerability, they may be able to leak or corrupt sensitive information, for instance to divert control flow.

Overflow detection. To detect buffer overflows during the execution, existing spatial memory safety defenses insert run-time checks in programs. Such *instrumentation* consists of one or more checks, in the form of branches, on either pointer dereferences [11, 12, 18, 26, 31, 34, 37] or pointer arithmetic (e.g., `ptr2 = &ptr[offset]`) [1, 23, 38]. Moreover, the metadata describing object bounds must be recorded, propagated and retrieved numerous times, often from memory [18, 23, 26, 31, 34, 37, 38]. Given the many extra branches and memory accesses required, the run-time checks can introduce significant runtime overhead.

As a result, the reduction of the number of branches and (especially) memory accesses has been the motivation for most prior research in spatial memory safety. Early defenses stored all their metadata in memory, for instance recording the base and size for every object in the program [23]. More recent defenses limit such in-memory metadata to only pointers that reside in memory [31] or to only the lower bounds of objects [26]. Alternative designs avoid storing metadata in memory altogether by manipulating the memory layout to implicitly encode the size and base of the object in the address of its memory location [1, 11, 12].

The need for (expensive) branching has also been a target of optimizations. Traditional spatial safety defenses require two branches: one for the lower bound and one for the upper bound of an object [31]. More recent defenses have suggested reducing pointer validation to a single branch by enforcing predictable (but wasteful) power-of-two alignment on allocated objects [1]. Besides pointer validation, existing defenses also require additional branches for compatibility with common real-world scenarios, such as branching on temporarily out-of-bounds pointers [1, 6, 38] and branching on uninstrumented pointers with NULL metadata (e.g., in the case of uninstrumented shared libraries) [26, 31].

In this paper, we focus on the most common class of spatial safety errors (*buffer overflows*) and investigate whether minimizing the number of extra branches and memory accesses (and the overhead) to detect overflows is possible. We start our analysis with a simple adaptation of state-of-the-art solutions. Specifically, we developed an adaptation of SGXBounds [26] that only stores the upper bound of an object in the high bits of its base pointer and uses a (branching) check on each memory access to compare the two parts of the pointer and detect (only) overflows. This design eliminates extra memory accesses, but still requires branching (for both pointer validation and support of uninstrumented modules) and incurs 48% (SPEC) overhead in our experiments. With Delta Pointers, we present a new design that can eliminate extra branches and memory accesses altogether, while retaining the same security guarantees

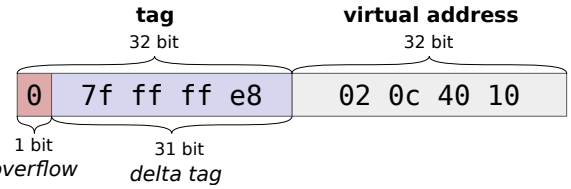

Figure 1: The encoding of Delta Pointers (tagged pointers). It points to an object on the heap of 24 bytes. The delta tag encodes the distance from the current pointer to the end of the object. The value of the delta tag is calculated as $-distance$ (here -24, in two's complement). The most significant bit is only set if the pointer is out-of-bounds.

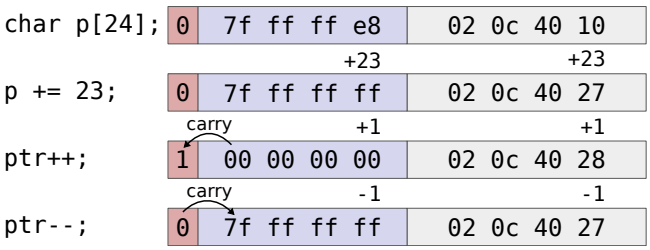

Figure 2: Examples of how the metadata inside a pointer is updated during pointer arithmetic: every operation on the address (lower bits) is also performed on the tag (higher bits). The most significant bit, indicating whether the pointer is out-of-bounds, is implicitly set and unset during these operations.

and incurring only 35% (SPEC) overhead. In other words, removing the branches reduces the overhead by roughly 25%.

3 THREAT MODEL

We consider an attacker able to exploit a buffer overflow by feeding malicious inputs to a given vulnerable user program. We assume the attacker can repeatedly interact with the program, and the program is automatically restarted in case of crashes caused by failed exploitation attempts. Our goal is to detect user-space exploitation of arbitrary buffer overflows when memory is either written to of read from, protecting both integrity and confidentiality.

4 DELTA POINTERS

Delta Pointers eliminate the need for branching checks by encoding the out-of-bounds state of a pointer in the pointer itself. Rather than relying on expensive instrumentation, we use hardware memory protection mechanisms to *implicitly* invalidate pointers that go out-of-bounds, and revalidate them when they go back in-bounds. Upon dereference, out-of-bounds pointers automatically trigger a fault. This is possible by encoding, in every pointer, a *tag* that describes the distance to the end of the memory object. This scheme aims to translate the buffer overflow detection problem into an arithmetic overflow detection problem, which can be dealt with more efficiently thanks to (i) detection offloaded to the hardware, (ii) efficient load/store and pointer arithmetic instrumentation with no branching or memory accessing instructions, (iii) no extra branches required to support common compatibility features, such as temporarily out-of-bounds pointers and uninstrumented pointers.

The absence of memory accessing instructions and out-of-band in-memory metadata provides two additional benefits other than good performance. First, the lack of in-memory metadata ensures a compact memory footprint (Delta Pointers introduce negligible memory overhead). Second, using only in-pointer metadata eliminates any need for synchronization across threads for metadata management, ensuring scalability and thread-safety in multi-threaded applications.

Pointer encoding. To enable our design, we make use of pointer tagging both to implement implicit out-of-bounds pointer invalidation and to ease propagation of metadata inside and across contexts. Specifically, each pointer is tagged with the current distance from the pointer to the end of the object, called the *delta tag*, and an *overflow bit*. As the distance from the current pointer to the end of the object changes during pointer arithmetic, the delta tag is kept consistently up to date. The overflow bit indicates whether the pointer is out-of-bounds, and is implicitly set during delta tag updates. To facilitate the implicit management of the overflow bit, the delta tag is encoded as the negated remaining distance, as we will explain later. Figure 1 illustrates our encoding scheme: the uppermost bit is the overflow bit, followed by the delta tag. This encoding allows for 32-bit tags, limiting addresses to 32 bits similar to prior tag-based schemes [26]. However, in contrast to prior schemes, in our design this division of bits can be changed arbitrarily depending on the application; the address space need not be limited to 32 bits. Instead, it allows a trade-off between the maximum object size and the address space size: as one increases, the other decreases, both by a factor of two.

The key insight exploited by our encoding scheme is that, by updating the delta tag alongside the pointer itself (which can be done efficiently), the state of the overflow bit is managed implicitly. Figure 2 shows the state of the bits during several operations on the pointer. The allocation on the first line creates a new pointer and initializes the delta tag to $-object\_size$. When adding an offset to the pointer, the addition is replicated on the delta tag, as shown on the second line. On the third line we do the same, this time causing the pointer to go out-of-bounds: the accumulated offset added to the delta tag is equal to the encoded object size. The pointer now points to the end of the object and the distance towards the end is 0. The carry bit of the addition on the delta tag sets the overflow bit implicitly. The fourth line shows that the same mechanism can bring a pointer back in bounds. When we subtract 1 from the entire upper part of the pointer (the delta tag and the overflow bit), it returns to the same state as in the second line.

Whenever the program dereferences a pointer, we first apply a bitwise AND operation to mask out the delta tag and create the regular untagged pointer the CPU expects. By maintaining the overflown state of a pointer in its upper bit we eliminate the need for an explicit (branching) check when dereferencing the pointer. Instead, we leave this bit intact by not masking it out with the delta tag as illustrated in Figure 3. Through this approach, we delegate the check to the memory management unit (MMU) of the processor: a pointer can only be used if it is in a canonical

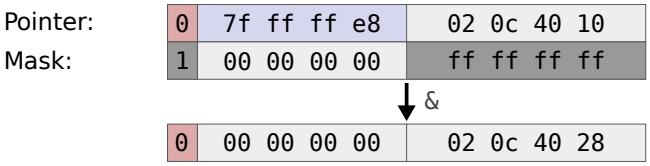

Figure 3: Before using the pointer to access memory, we have to mask out the metadata to create a valid pointer. By using a bitmask with the most significant bit, we keep the overflow bit of the pointer intact. If the pointer was overflown, and thus had its overflow bit set, this bit will still be set in the resulting pointer. This is an invalid (non-canonical) pointer, causing a run-time error.

form, where the 16 most significant bits are sign-extended [20]. For pointers in user space, this equates to 16 zero bits, since the upper bit of an 48-bit pointer is reserved for kernel space in most operating systems. A set overflow bit therefore makes the pointer uncanonical, causing the MMU to generate a fault. The underlying assumption is that bitwise masking operations, combined with arithmetic operations on pointer additions, are highly optimized on a microarchitectural level, and are therefore faster than explicit checks with branches. Even though on modern architectures the branch predictor is heavily optimized, branches still incur overhead.

Listing 1 shows the translation of these two concepts into code instrumentation. ① creates the delta tag in the correct format and places it in the upper bits of the pointer. At ② we replicate the addition that happened in the lower bits to the upper bits. Finally, ③ performs masking, leaving the pointer and the overflow bit intact. Because the delta tag is part of the pointer itself, it is automatically propagated across contexts, and will not result in any memory accesses. Moreover, ② and the pointer arithmetic can be combined to a single operation: `nptr = ptr + (user_input + (user_input << 32))`, even down in the output assembly.

NULL pointer protection. The NULL pointer constitutes a special case of a pointer that is not derived from any object, but still provides an (often neglected) attack surface for spatial errors. Exploits for such bugs typically add an attacker-controlled offset with the value of the target location to grant the attacker arbitrary memory read or write capabilities [24]. For Delta Pointers, we set a delta of 1 on all NULL pointers, thus replacing them with a value of `0x7fffffff00000000`. This will cause any dereference of a pointer derived from NULL to trigger a fault and hence detection.

Thread safety. Delta Pointers are inherently safe with respect to racy pointers. Added instrumentation on memory operations and pointer arithmetic consists of arithmetic operations that operate on registers, and no additional memory accesses are introduced. Modified pointers are written to memory in a single atomic operation (on most architectures), causing the pointer tag to always be consistent with its corresponding address. An existing race condition may be influenced by the introduction of additional operations on either end, but this is an inherent problem of racy pointers, not a race condition introduced by Delta Pointers.

```
void *foo(int user_input) {
  char *ptr = malloc(16);
  delta_tag = (1 << 31) - 16;                    ①
  ptr |= delta_tag << 32;

  char *nptr = &ptr[user_input];
  nptr += (uintptr_t)user_input << 32;           ②

  tmp = nptr & 0x80000000ffffffff;               ③
  *tmp = 'a';
}
```

Listing 1: C program instrumented with Delta Pointers. Instrumentation is shown as pseudocode on the highlighted lines. ① sets the delta tag, ② updates the delta tag alongside the pointer, and ③ strips out the metadata (but not the overflow bit) before a memory access.

Address space reduction. Since pointer tagging-based defenses use part of a pointer to encode metadata, they reduce the amount of bits left for addresses. Because this limits the addressable virtual address space, ASLR entropy is reduced as well. For instance, Low-Fat pointers [11] reserves a large virtual memory region for each object size class, and SGXBounds [26] encodes two 32-bit addresses side by side in each pointer, thus not fully utilizing the 36 bits of entropy offered by SGX. Delta Pointers similarly limit addresses to 32 bits by default (but configurable on a per-application basis). Although ASLR has a wider scope than these schemes (e.g., use-after-free bugs), it can only provide probabilistic safety at best, whereas Delta Pointers provide deterministic (spatial) memory safety guarantees on the upper bound. Because of its probabilistic nature, ASLR has proven to be easily circumventable by memory massaging [16, 33] or side channels [5, 15, 17] whereas this is not possible for deterministic defenses such as Delta Pointers. Moreover, the impact of address space reduction is limited in certain application domains. As an example, similarly to SGXBounds, Delta Pointers are well suited to run arbitrary programs in SGX enclaves, which only support 36-bit addresses currently. By encoding metadata in pointers rather than in memory, the only memory overhead of Delta Pointers is that of added code (which is negligible), even amounting to an advantage over, for example, shadow memory-based schemes in SGX where virtual memory is limited.

Our Delta Pointers prototype uses 32 bits to address a 4 GB address space. The remaining 31-bit delta tag allows for a maximum allocation size of 2 GB. This split is configurable and can be tweaked depending on the application: a program that allocates a lot of smaller objects can have a bigger address space. For instance, without any address space reduction, an application would have 47 bits of address space and 16-bit delta tags, allowing for only 64 KB objects. For Delta Pointers we did all evaluation with 32-bit addresses and 31-bit delta tags because our experimentation shows this achieves a good compatibility with a large set of complex real-world programs under realistic workloads.

5 POINTER TAGGING

Delta Pointers are an instance of pointer tagging to efficiently store metadata per pointer, yielding a design that has low overhead for lookups, negligible memory overhead and automatically works with concurrent programs. These benefits do not come for free: modifying the representation of pointer introduces various challenges regarding compatibility and performance. These challenges are not limited to Delta Pointers, but are generic in nature and must be dealt with by any defense that encodes metadata in pointers.

In this section, we discuss the challenges that need to be addressed by pointer tagging based approaches, with the aim of inspiring and assisting future research based on this increasingly popular technique. We also present the solutions to the listed problems which are implemented in Delta Pointers, showing that high compatibility with complex real-world applications is possible in the presence of tagged pointers. We discuss four challenges with pointer tagging. First, adding arbitrary metadata in pointers requires careful implementation of pointer operations to conform to the C standard and implicit assumptions of C programs on its implementation. Second, compilers make certain transformations that make pointer identification harder, in particular during optimizations. Third, defenses must deal with pointers that do not have metadata due to optimizations or uninstrumented libraries. Finally, the speed of decoding tagged pointers is influenced by micro-architectural properties.

5.1 C pointer operations

The presence of metadata tags in pointers means that, without additional measures, the integer comparisons and arithmetic operations may no longer accurately implement the semantics defined by the C standard [21]. Moreover, while the C standard generally attempts to leave data representation to be defined by the implementation, in practice many C programs make assumptions that go much further than the guarantees provided by the standard [7].

Additional instrumentation, in particular targeted removal of pointer tags, is required to achieve compatibility of arbitrary pointer tags with existing programs. Without this instrumentation, a program would make different computations or control flow decisions based on tagged pointers rather than regular ones. This can cause the program to crash or produce incorrect results. The problem only arises, however, when tags encode *per-pointer* metadata. This means that two pointers pointing to different locations in the same memory object may have different tags, causing operation semantics to change. Solutions using *per-object* metadata, such as SGXBounds [26], do not experience this problem, since they do not modify pointer tags after object allocation.

Pointer comparison. In the C standard, the outcome of comparison operators on pointers is only defined if the pointers either point to the same object or are part of the same aggregate object. Although other cases are explicitly left undefined, the standard does state in general that "the result depends on the relative locations in the address space of the objects pointed to". In practice, however, programs often assume that pointers are implemented as simple integers and expect a total order on them, for instance for sorting.

When tagging pointers, this assumption no longer typically holds, especially for dynamic tags. Tags should thus be masked away before comparing pointers to avoid an incorrect result. For Delta Pointers we mask away the tag including the overflow bit. This is safe, since the masked pointers are only used for comparison and never dereferenced, so no buffer overflow checks are needed.

Pointer subtraction. The C standard allows subtraction of pointers to the same array object. These are used to compute distances between objects in the address space. While normally implemented as an integer subtraction instruction, this may yield incorrect results in the presence of different pointer tags. The tags of subtracted pointers must therefore be masked away. Some programs even subtract pointers to different objects, thus violating the standard and making assumptions on the memory layout. However, these applications are still supported when tags are removed. Note that arithmetic optimizations by compilers may rewrite complex expressions involving pointers in such a way that these cases are introduced as well, as also discussed below. All these scenarios are supported by our Delta Pointers implementation.

Pointer alignment. Non-conforming programs often use bitwise operations on pointers to detect or enforce alignment properties. Such alignment works correctly for Delta Pointers without any additional instrumentation, since it can only round pointers down, increasing the distance to the end of the object. This will not lead to false positives since applications themselves must assume that the pointer remained unchanged and thus did not increase the distance to the end of the object (which is what Delta Pointers enforce).

5.2 Compiler support

Pointer tagging requires type information to be able to distinguish pointers, which, in turn, requires the defense to be implemented at the compiler level. Code instrumentation can be done at different compilation stages, in particular before and after optimizations. Instrumenting before optimizations is easier because then the code has not been transformed to patterns that hinder static analysis. Instrumenting all pointer values, however, severely handicaps optimizations: the instrumentation casts pointers to integers in order to add, modify, or remove the tag, breaking alias analysis. Hence, in order to attain high performance, pointer tagging must be applied after optimizations. This raises several issues that need to be addressed by pointer tagging implementations. When left unaddressed, these issues can either cause the instrumentation passes to fail due to unexpected inputs, or have them produce inaccurate instrumentation code that corrupts the program state at runtime. To the best of our knowledge, our Delta Pointers implementation is the first to provide wide compatibility with complex source code (e.g., SPEC CPU2006) due to the solutions described below.

Pointers as integers. Modern compilers such as LLVM implement optimizations that produce arbitrary typecasts from pointers to integer types. Hence, static analysis is needed to determine which values are pointers in optimized IR. For dereferencing operations, the problem does not occur because the pointer value must always be cast to a pointer type prior to dereference. Comparison, however, is defined on both integers and pointers, and subtraction only on integers. Therefore, a pointer value that has an integer type need not be typecast prior to these operations, making it unclear whether masking is needed. This problem can be solved using

use-def chains [30] to trace back the origin of the pointer to its definition (an allocation or function parameter) or a load from memory. Definitions contain the original variable type, which can be used directly to determine if the value is a pointer. Pointer loads from memory, however, can be transformed arbitrarily to integer loads by optimizations. If the loaded pointer is never dereferenced, the use-def chain does not provide information on its actual type, and the value is not masked, producing incorrect behaviour. The following example illustrates such a case:

```
int **a = ...;          // uint64_t *a;
int **b = ...;          // uint64_t *b;
diff = *b - *a;         // diff = *b - *a;
```

The pointers pointed to by `a` and `b` are never dereferenced, their values are only subtracted and the result is stored in memory. This allows the compiler to remove the (implicit) type casts of `*a` and `*b` to integers, effectively producing the code on the right, which alters the type information of the values in memory.

This problem can be partially solved in two ways. First, metadata describing the types can be added to values being loaded/stored before running optimizations. Certain optimizations, such as type-based alias analysis (TBAA) [10] already add such information, which can be reused. However, such metadata can potentially be removed by subsequent optimizations, and might thus not be complete. The second solution is to trace back the pointee type through the use-def chain of the address that is loaded, using nested type inspection to find a pointer type. This on itself works reasonably well since the folding of memory loads and type casts is often very localized, requiring only limited backtracing of the use-def chain to find the original pointer type. Our Delta Pointers implementation uses a combination of these solutions, which our experiments show is complete for all the tested programs.

Unions and pointer expressions. A union value of a pointer and an integer may produce a typecast from a pointer to integer. The operation can either be operating on the integer value or it can be operating on the pointer value but preceded by a typecast because the operation happens to only be defined on integers. For example, bitwise operations are only defined on integers but are also used for pointer alignment. Furthermore, compilers transform expressions involving pointers in ways that sometimes produce unexpected pointer operations. For example, consider $(b - a) \times 4$ where a and b are pointers. This should be caught as a regular pointer subtraction as per Section 5.1, but the compiler may transform this to $b \times 4 + a \times -4$, thus introducing pointer multiplication and then addition. We observed similar cases in SPEC CPU2006 involving division, remainder, bit shifts, bitwise OR, and bitwise XOR, all on pointer operands. The case above is easily solved by masking the pointer operands of multiplications, but the tag must be preserved when the result is a pointer that may later be dereferenced in order to do a bound check. In other words, the decision whether to mask the pointer operand of an arithmetic expression thus depends on whether the resulting expression is indeed used as a pointer (i.e., it is dereferenced). This is determined by traversing its def-use chain until such a use is encountered.

Although the combination of the above solutions theoretically does not cover all code patterns expressible in compiler IR, it works very well in practice. For instance, our Delta Pointers implementation correctly identifies all pointers in the C/C++ SPEC CPU2006 benchmark suite after full-fledged optimizations. Our design significantly improves compatibility with respect to state-of-the-art pointer tagging implementations such as SGXBounds [26], which forcibly omits 6 out of 19 SPEC benchmarks due to incorrect pointer identification.

5.3 Coverage considerations

Many pointer tagging applications assume that each dereferenced pointer is tagged with metadata. For example, SGXBounds stores a pointer in the tag that is dereferenced to retrieve further metadata. When a pointer misses metadata, a `NULL` pointer is dereferenced and the program crashes. However, reaching complete coverage is not always possible or even desired. For instance, optimizations by the defense may omit instrumentation on pointers that only have safe uses. When these pointers are used at the same site as unsafe pointers, the instrumentation at that site cannot assume the presence of metadata in the pointer. Furthermore, uninstrumented libraries may generate untagged pointers, and cannot handle tagged pointers passed in library call parameters.

A robust pointer tagging-based defense should therefore be designed to deal with missing metadata. Unfortunately, most existing defenses require extra branches to deal with missing metadata. Alternatively, and even less desirably, such defenses must instrument all libraries and disable aggressive optimizations. In contrast, Delta Pointers are robust by design against missing metadata because the zeroed metadata is not dereferenced as a pointer. It effectively treats missing metadata as a very large distance to the end of the object, larger than the object can possibly be, therefore simply disabling the bound check in a situation where metadata is missing (which is the expected behavior).

Uninstrumented libraries are, in fact, part of a general problem with pointer tagging: there is always a *protection boundary* at which marshalling must occur between tagged and untagged pointers. Even if all libraries are statically linked and instrumented, the boundary is only moved to system calls that cannot operate on tagged pointers (unless even the OS kernel is rewritten). Ideally, pointers handled by the protected application are always tagged and pointers outside the boundary are not. In other words, the storage of pointers can not be *shared* between protected and unprotected code. A complex example is `std::list::push_back` in an uninstrumented `libstdc++`. The implementation of this function is defined in a header file and therefore resides in the protected executable. It calls a library function that takes a generic node type for doubly linked lists as a parameter. This data structure is created in `push_back`, which is instrumented because it resides in the protected executable, and thus tags the node's `next` and `prev` pointers with metadata. Code inside `libstdc++` does not mask the pointers before dereferencing and crashes if these pointers are tagged.

Defining a protection boundary requires a set of rules that specify when to add or remove tags to pointers. Rules for removing tags are mostly universal, but rules for adding tags depend on the specific defense being implemented.

5.4 Performance considerations

The minimum instrumentation required to implement pointer tagging consists of tagging and masking. These are bitwise operations that are fast in modern, pipelined processors. However, they may still have an effect on the pipeline and increase register pressure when the masking constant does not fit in an immediate operand (as is the case for 64-bit masks on x86-64). For this reason, recent hardware designs are shipping with built-in support for *MMU-based masking*: AArch64 supports virtual address tagging [2] which introduces a top byte ignore option for the hardware to ignore the upper 16 pointer bits during dereference, specifically for the purpose of encoding metadata. Oracle's SPARC-M7 architecture [35] can even ignore up to 32 bits of metadata. This completely eliminates the need for software-based masking and paves the way for highly efficient encoding of per-pointer metadata. In other cases, efficient masking can be done with fewer restrictions on the bitmask width than on x86-64. For instance, ARM64's AND instructions support efficient variable-length encoding of certain 64-bit immediates, including the bitmask required by Delta Pointers.

Another performance consideration of pointer tagging solutions is protection against metadata corruption. Arithmetic on a pointer may overflow into the metadata bits, allowing an attacker to bypass the implemented defense. SGXBounds experiences this problem, and thus needs to move the metadata out of a pointer before every pointer arithmetic instruction and move it back in afterwards, incurring significant additional overhead. Delta Pointers do not need special treatment here: a pointer overflow will also overflow the delta tag into the overflow bit, correctly invalidating the pointer.

6 IMPLEMENTATION

We have implemented a prototype of Delta Pointers for Linux on the x86-64 architecture on top of the LLVM compiler infrastructure [27] (version 3.8). The code consists of 3,749 SLOC of LLVM C++ passes, which add the instrumentation described in Sections 4 and 5. An additional 846 SLOC make up runtime and helper libraries, including a static library that shrinks the address space of the process to make room for tags in pointers. The code is open source.[2]

In order to harden an existing program with Delta Pointers, the programmer adds compiler flags that invoke our passes during the compilation of source into the binary, and during linking to link in our static library. The resulting binary is then run through a post-processing script to shrink its address space. The resulting binary can then run as-is, raising an error upon detection of an out-of-bounds memory access. Dereferences of out-of-bounds pointers will cause the MMU to trigger a general protection or stack segment fault, which Linux will deliver to our process as a segmentation fault or bus error. We install a signal handler to distinguish such cases and report an appropriate error.

6.1 Address space reduction

User-space pointers in Linux are 47 bits. We limit this to 32 bits to support 32-bit tags. Only changing the allocator is not sufficient for this purpose, as the kernel maps the stack and loader at high memory addresses. The loader itself will also run code performing allocations before we get a chance to insert code. A kernel patch is the most straightforward way to limit the address space for mappings, but provides poor portability. Instead, we use the approach of Mid-Fat Pointers [25] to limit the address space in user mode: First, we prelink the binary, loader, and any shared libraries used by the program at locations that fit 32 bits. Then, during program startup, we move the stack and thread-local storage down in the address space. Finally, we reserve the memory area above 32-bit with an anonymous non-reserved mapping to avoid subsequent allocations in this address range.

[2] https://github.com/vusec/deltapointers

6.2 Instrumentation

Listing 1 in Section 4 describes the instrumentation added by our LLVM passes. We apply these changes after optimizations, including link-time optimization (LTO), so that these optimizations are not hindered by our inserted pointer-to-integer casts. A final optimization pass performs optimizations such as constant folding on the added instrumentation.

For dereferencing instructions, we consider LLVM's `load` and `store` instructions and memory intrinsics (which cover calls to the `memcpy` family). For memory intrinsics, we update the pointer tag with the size of the dereferenced memory range minus one so that the highest dereferenced pointer is checked. The size is truncated to the maximum object size of 31 bits to also check very large sizes caused by implicit casts from signed to unsigned integers.

6.3 Coverage

Finding all heap allocations. Detecting overflows for all buffers requires identification of all memory allocations so that the resulting pointers can be tagged. Stack allocations and globals are trivially identified by their unique representation in LLVM, but heap allocations are performed by calls to memory allocator functions. We currently support all allocation functions in the C and C++ standard libraries. Custom allocators that preallocate a pool of memory using `malloc` or `mmap`, however, must expose allocation function names and the position of their size arguments to the Delta Pointers implementation in order to support per-object buffer overflow detection (otherwise only per-pool overflows can be detected). This is the case for *nginx* which we support by adding its custom pool allocation functions to our predefined list.

Pointer marshalling at library calls. As explained in Section 5.3, making sure that all pointers in the protected executable have a tag requires a set of rules that define how pointers tags are added and removed at the protection boundary. For Delta Pointers, we have opted to place this boundary at the level of dynamic library calls in order to provide portability and maintainability, preserving the ability to update libraries without having to recompile all protected programs that use these libraries.

Pointers passed as parameters to library calls are masked in the same way as dereferencing operations, so the overflow bit is left intact. This design even prevents an attacker from using a vulnerable library function to dereference an already out-of-bounds pointer (however does not protect against out-of-bounds dereferences if the library code adds an offset to the pointer). Nested pointers in data structures, such as those used in `std::list::push_back`, are stripped of metadata when a pointer to the data structure is passed to a library function. The tags are not restored after the function

call, so these pointers remain unprotected inside the protected executable as well, thus assuming that the library implementations handling the data structures are safe. No tag is added back to the pointer, since it is not known how the function alters pointers in the data structure. Although this introduces untagged pointers in the protected executable, all functions that operate on the data structure are in fact implementations of standard library functions in header files. These functions are in fact outside the protection boundary, only included in the executable for optimization reasons (inlining). The only other case requiring similar instrumentation is `std::string::operator+=`.

Finally, data pointers returned by library functions are tagged to offer protection inside the executable. In particular, the rules specify how the distance from the returned pointer to the end of the referent object can be inferred from the call parameters. We have analyzed all functions in the C and C++ standard libraries and identified six categories:

- **copy**: Copy the tag of an argument. E.g., `strdup`.
- **diff**: $ret_{tag} = param_{tag} + (ret_{address} - param_{address})$. E.g., `strchr`.
- **static**: Tag is constant, size inferred from return type. E.g., `fopen`.
- **strlen**: $ret_{tag} = strlen(ret_{address})$. E.g., `getenv`.
- **strtok**: Special case for `strtok`: replace its implementation with a version that maintains the current end-of-object distance in a global variable.
- **noarith**: Disallow pointer arithmetic by assuming an object size of 1 byte. This is used for opaque return types such as that of `opendir`, whose returned pointer is not dereferenced inside the executable itself but only passed as a parameter to library calls.

Using these categories, we are able to instrument 99.7% of all dereferenced pointers in SPEC CPU2006. The remaining 0.3% are all related to shared state between protected code and unprotected library code. We verified through manual inspection that these cases can either be fixed with static analysis (e.g., `argv`) or can safely be ignored. An example of the latter case are C++ VTables which contain code pointers to virtual methods of objects. Each object stores an object to its VTable, which if instrumented cannot be dereferenced by libraries. The table structure is, however, an implementation detail of the compiler and not transparent to the programmer. All accesses to VTables are compiler-generated and can therefore be assumed to be in bounds (not accounting for compiler or type-confusion bugs). Our Delta Pointers implementation therefore omits tags on VTable pointers in favor of compatibility. Note that no checks need to be inserted on pointers without metadata, which Delta Pointers support by design.

6.4 Optimization

Some existing bounds checkers implement analyses that identify *safe* memory accesses, which are statically known to be in-bounds. Unfortunately, these optimizations are not directly applicable to Delta Pointers, because the bounds checks are implicitly performed at pointer arithmetic sites. Masking instrumentation can only be removed from a memory access if none of the possible pointer values can have a tag, meaning that any instrumentation that adds or modifies the tag must first be removed. This is not always safe to do, for example when the same allocation is also passed to a function that does a possibly out-of-bounds memory access.

We use static analysis to find allocations, propagation sites (pointer arithmetic) and masking sites (loads/stores) that do not need instrumentation. In particular, we use LLVM's scalar evolution (SCEV) analysis [13] to trace back pointer bounds from loads and stores and omit instrumentation where the pointer can be proven to be in bounds. The analysis records the distances to the object start and end $(0, size)$ at each allocation site. These bounds are then propagated over the def-use chain of the allocation. At a propagation site, the offset is added to the recorded start distance and subtracted from the end distance. At a load or store, the recorded distances are used in combination with the dereferenced number of bytes to check if the dereferenced pointer is in bounds. Instrumentation is omitted on pointers produced by allocations or propagation sites that are only dereferenced in bounds. Masks are also omitted from dereferenced pointers that do not contain a tag because of optimizations. Note that since we use SCEV analysis, which represents IR operations as expressions, both *size* and pointer offsets need not be constants, thus allowing for aggressive optimizations.

Another optimization we perform is the *hoisting* of bounds checks out of loops. Consider a simple loop that requires a bounds check in each iteration:

```
for (i = 0; i < 10; i++)
    buf[i] = `x';
```

It is easy to see that this check only needs to be performed once on `buf[9]`. We use the same SCEV analysis as described above to infer the maximum value at compile time and insert a dummy load before the loop to trigger a fault if the computed offset is out-of-bounds. Instrumentation on the operations inside the loop is removed. This optimization can only be performed if the pointer `&buf[i]` does not escape the loop body. Unfortunately, LLVM rewrites the exit condition in the above loop to `i == 10`, thus making `&buf[10]` the maximum value of the pointer inferred by SCEV. Although the pointer has this value after the loop, it is never actually dereferenced. We have implemented a simple pattern detection to support this case, but the problem still exists for complex loops found in real-world programs such as in the SPEC benchmarks. We consider this a limitation of the SCEV implementation and therefore out of scope for this work.

7 EVALUATION

To evaluate Delta Pointers we look at both their performance and security. To study the performance we benchmark SPEC CPU2006 and other real-world applications. We then evaluate the effectiveness of Delta Pointers by determining if they mitigate a number of recent CVEs reported by related work.

7.1 Runtime performance

We have evaluated Delta Pointers on the C and C++ programs of the SPEC CPU2006 benchmarking suite [19]. This suite contains a wide variety of complex real-world programs, including a Perl interpreter, XML parser and simulations. Additionally, we evaluated Delta Pointers on Nginx 1.10.2. We used Intel Xeon E5-2630v3 machines with 16 cores at 2.40 GHz and 64 GB of memory, running

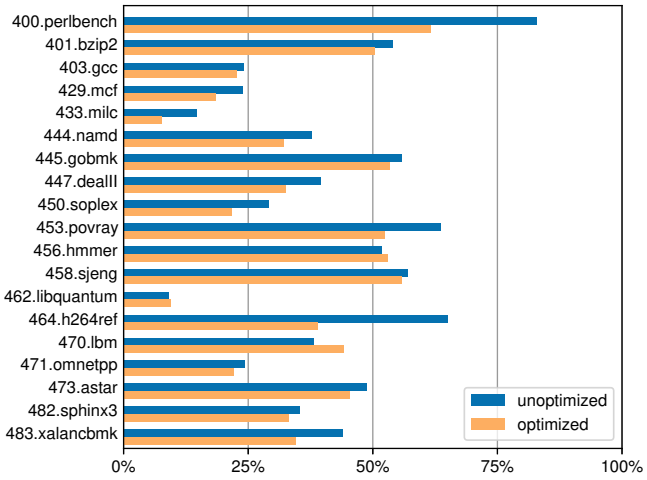

Figure 4: Runtime overhead of SPEC CPU2006 for our Delta Pointers prototype.

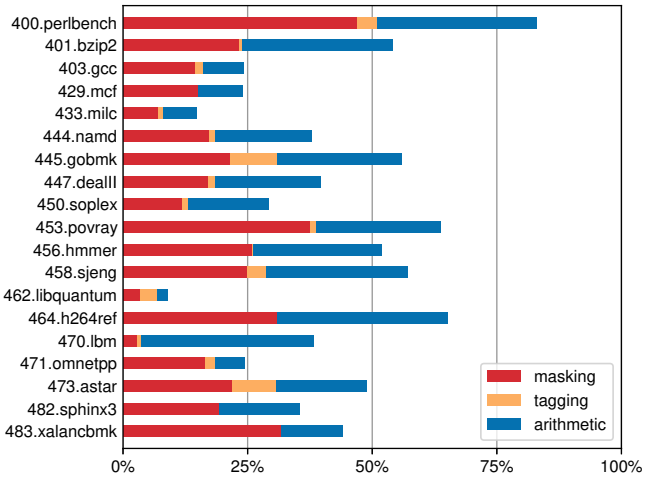

Figure 5: Runtime overhead of SPEC CPU2006 for different components of Delta Pointers instrumentation (with optimizations disabled).

64-bit CentOS 7.2.1511. Each overhead number is the median of 16 iterations of the same program (using the reference workset for SPEC), and we manually verified standard deviations to be negligible. For the baseline, we enabled link-time optimizations. Two of the 19 benchmarks are not compatible out-of-the-box with Delta Pointers: *403.gcc* uses upper pointer bits of pointers larger than 32 bits for page ordering, and corrupts a pointer tag by using NULL pointer subtraction and addition instead of a regular typecasts for type conversion between pointers and integers. *450.soplex* is incompatible with per-pointer bounds checkers: it patches pointers to point to out-of-bounds locations using pointer subtraction after reallocating a buffer containing pointers. This violation of the C standard is well-documented in related work [26, 34], and other researchers either patch or omit these programs as well. In order to avoid the impact of omitting benchmarks on the overall overhead, we wrote small source patches for *403.gcc* to configure pointer bitsize at 32 bits and to preserve a pointer tag at a single NULL pointer subtraction, and for *450.soplex* to fix the delta tag on patched pointers after `realloc`. The evaluation numbers thus include the entire C/C++ subset of SPEC CPU2006 (19 benchmarks in total).

We have benchmarked Delta Pointers both with and without the optimizations from Section 6.4. Figure 4 shows the measured runtime overhead for these configurations (raw runtimes are available at the end of the paper). In all cases, Delta Pointers show a negligible memory overhead. The instrumentation causes 41% geometric mean (geomean) runtime overhead, which is reduced to 35% by optimizations. In comparison, our implementation of using branches for upper-bound checks (which is equivalent to SGXBounds without underflow checks) achieves 48%, confirming that bitwise and arithmetic instrumentation is significantly faster than using branches. The instrumented binaries are on average 80% bigger, both in terms of instructions and file size.

To accurately compare these results with related work, we tried to reproduce competing results on the same setup in order to compare to the same efficient baseline on modern hardware. We evaluated SGXBounds on our setup and obtained 94% geomean overhead which is much higher than the 55% reported in the paper. After consistently seeing these results across machines, we contacted the authors but together we were not able to determine the root cause of this difference. The authors of Low-Fat Pointers were unable to share their prototype due to it being in an alpha state, and Baggy Bounds [1] is closed-source. We could easily evaluate AddressSanitizer because it is part of LLVM. The run times obtained on our experimental setup are available at the end of the paper.

Overall, we can see that our overflow detection design is far more efficient than ASan, which has 80% overhead. We only have slightly lower compatibility: we require small source patches for two SPEC 2006 benchmarks whereas ASan requires this for one benchmark. ASan also has a large memory overhead, whereas Delta Pointers have negligible memory overhead. Also note that ASan has weaker spatial detection guarantees, since it can only detect contiguous overflows. On the subset of SPEC 2006 benchmarks supported by SGXBounds, Delta Pointers have a 35% overall overhead, compared to the 94% of SGXBounds, while achieving higher compatibility (SGXBounds cannot run 6 benchmarks because it does not deal with some issues described in Section 5). Low-Fat Pointers does not support 4 out of the 19 SPEC benchmarks, but can be benchmarked using manual source fixes. The paper reports 52% overhead versus 35% for Delta Pointers. In addition, Low-Fat does not cover globals or the `NULL` pointer and provides lower compatibility overall.

To gain insight in the origin of the measured overhead, we have independently measured the overheads of different instrumentation components (with optimizations disabled). The results are in Figure 5. We observe that the overhead of masking at every pointer dereference is geomean 20%. This is perhaps higher than expected since the operations are bitwise ANDs which are expected to be fast. But the performance impact of register pinning and pipeline stalls, as described in Section 5.4, evidently proves non-trivial and likely requires hardware optimizations to further reduce the overhead. Tagging all allocations with delta tags is cheap, adding only

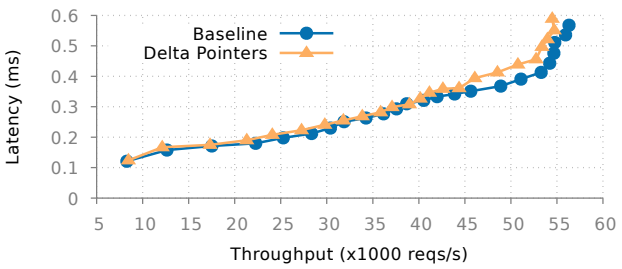

Figure 6: Overhead of Nginx web server for Delta Pointers.

2 percent point overhead. This is expected, since allocations typically happen outside of the main computation loops. Finally, like masking, instrumentation on pointer arithmetic is higher than expected at 19 percent point, making the overall overhead 41%. We have investigated the assembly generated for our instrumentation by the compiler and concluded that the instrumentation hinders optimizations made by the LLVM x86-64 backend. In particular, the instruction selection chooses to emit separate addition and multiplication instructions for pointer offset computations rather than using efficient scaling-based addressing mode as supported by `lea` and `mov`. This is especially the case for dynamic offsets that cannot be constant-folded. Unfortunately, dynamic offsets are prevalent in SPEC: we found that 72% of all pointer arithmetic instructions in the reported benchmarks use dynamic offsets. Thus, although our current overhead is already competitive, (admittedly non-trivial—due to the LLVM architecture) optimizations of the instrumentation of dynamic pointer arithmetic in the compiler backend may improve the results even more.

We have also tested Delta Pointers on the Nginx web server. The performance of Nginx primarily depends on I/O, and Nginx does relatively few pointer operations. Because of this we observed negligible effects on performance, as shown in Figure 6. Our benchmarks were performed requesting 64 byte pages with 8 workers over a 54 Gbit/s link. On average we observe a 4% increase in latency, with a maximum of 6% for the unsaturated case going down to 3% when saturation is reached. When the server is saturated we observe only a 2% drop in throughput for Delta Pointers.

7.2 Security

We have evaluated the effectiveness of Delta Pointers by examining a number of recent common vulnerabilities and exposures (CVEs) [8]. To be able to compare to previous work, we have examined all CVEs reported in recent work [12, 26]: 8 CVEs across 6 popular programs, including Heartbleed in OpenSSL and bugs in Nginx and PHP. Rather than only checking if existing exploits are circumvented, we used manual analysis to determine if the *exploitability* of an attack is affected.

As detailed below, Delta Pointers completely prevent 7 of the 8 analyzed attacks, with the uncaught bug only supported by a single existing bounds checker, demonstrating that Delta Pointers offer practical security guarantees. The undetected bug is not fundamental to the design of Delta Pointers, but is outside the protection boundary of our implementation. Adding support for the bug is trivial (a single line of code) by treating `recv` as a memory intrinsic. This could be done for all standard C library functions, as done by SGXBounds, to extend protection to outside the protection boundary.

CVE-2011-4971 in MemCached 1.4.15. A signed integer is set to a negative value by the attacker, and passed as a large unsigned size to `memcpy`. Our instrumentation on memory intrinsics covers this case, preventing out-of-bounds reads.

CVE-2013-2028 in Nginx 1.4.0. An attacker-controlled negative signed integer is passed as a large unsigned size to `recv` which copies data into a limited-size buffer. Delta Pointers cannot not detect this case since the write resides in an uninstrumented libc function. To cover this vulnerability, functions like `recv` could be encapsulated in wrappers that implement checks on the underlying memory accesses. Note that this requires manual analysis of the semantics of all library functions that access memory buffers based on parameter values which is only done by SGXBounds, so the bug is not covered automatically by any other existing defense.

CVE-2014-0160 in OpenSSL 1.0.1f (Heartbleed). A 2-byte attacker controlled response length is used to transmit back a buffer of an attacker-controlled size, allowing the attacker to leak up to 64KB of memory, including private keys. The buffer pointer is correctly tagged and the overread is detected successfully.

CVE-2016-1234 in glibc-2.19. While glibc is not compatible with LLVM out-of-the-box, we still analyze this CVE to compare effectiveness of our design to that of Low-Fat Pointers which reports it. The bug is a stack buffer overflow due to an access with the length of a directory name as offset, which can be up to 510 bytes larger than the buffer size on for instance the NTFS file system. Our Delta Pointers design would correctly tag the allocated buffer and prevent the attack.

CVE-2016-2554 in PHP-5.5.31. A string inside `struct _tar_header` is assumed to be null-terminated but can be attacker controller. By crafting a struct without any null-bytes this intra-struct overflow can be extended far beyond the size of the struct. This finally ends up in a `strncpy` with the overflowed size of the struct, which Delta Pointers detect because of our `strncpy` instrumentation.

CVE-2016-3191 in PCRE2-10.20. An attacker-controlled regex can cause a contiguous overflow on a stack buffer, as the `(*ACCEPT)` verb will write a closing parenthesis for any currently open parenthesis, without checking for the presence of such closing parenthesis nor whether there is space in the buffer. Our Delta Pointers design easily detects this case.

CVE-2016-6289 in PHP-7.0.3. Similar to CVE-2011-4971 a signed integer is passed to a `memcpy` call. Our Delta Pointers implementation instruments the intrinsic and detects any overflow.

CVE-2016-6297 in PHP-7.0.3. An attacker can supply a large string triggering an integer overflow in `strlen`. The resulting length is then passed to `memcpy` where it is cast to a `size_t` similar to CVE-2011-4971, which is detected.

SPEC CPU2006. We have also confirmed a number of benign buffer overflows in SPEC CPU2006 which are reported by related

work [26, 37]. `perlbench` contains a benign buffer overread in a `memcmp` call, which is caught by Delta Pointers because of our intrinsic handling. `h264ref` contains two bugs: one involving a global variable and one on the stack. The stack-based overflow is entirely optimized away by LLVM during vectorization (as LLVM can statically determine there is undefined behavior). If we disable these optimizations, Delta Pointers detect the bug. The bug where a global variable is overflown is also detected by Delta Pointers, in contrast to, for instance, Low-Fat Pointers. For the performance evaluation of Delta Pointers, we fixed these bugs using the source patch from AddressSanitizer.

8 DISCUSSION

Bounds narrowing. Our prototype does currently not support *bounds narrowing*, where bounds of sub-objects in composite types are enforced (e.g., an array in a struct). Since our Delta Pointers design records per-pointer metadata, such a feature could easily be added. Such strict enforcement of bounds is knows to cause compatibility problems [7, 34].

Integer overflow on pointers. Most processor architectures implement arithmetic overflow on regular integer additions: when all bits in an integer are set and 1 is added, the number wraps around to zero. In our pointer encoding scheme, an attacker may be able to clear the overflow bit by adding a very large offset that causes such an overflow. This can normally be mitigated by simply limiting offsets to 32 bits (which is already normally the case in real-world programs), but, in some cases, the attacker might be able to use a pointer addition inside a loop to iteratively overflow the pointer. Thus, in order to provide complete protection on the upper bound, Delta Pointers require an upper bound on the result of pointer additions. This is called *saturation arithmetic*. Saturating operations *clamp* their results to a given minimum and maximum, typically with all-zero and all-one bits respectively.

Some architectures have dedicated instructions for saturation arithmetic. For instance, ARM offers the `UQADD` instruction to perform saturating addition on 64-bit unsigned integers, which constitute our use case. Intel x86-64, however, only supports 8/16-bit saturation through `PADDUSB`/`PADDUSW`. 64-bit saturation on x86-64 is most efficiently performed by conditional (but non-branching) instructions (`CMOVcc`) which replace the result of an addition based on the value of the `FLAGS` register. To support saturation arithmetic, Delta Pointers could optionally use conditional instructions to replace the result of pointer arithmetic with the maximum integer value. Such instructions are reasonably fast since they operate only on registers, but they must be inserted on all pointer arithmetic, hence resulting in higher overhead. For our Delta Pointers prototype, we felt this was not a feature to prioritize given the additional performance cost and limited security improvement. Namely, the feature is only necessary if an attacker can add an offset of `(1 << 31) + object_size` bits to a pointer, either at once or iteratively, without dereferencing it in the meantime (since the intermediate pointers are detected as out-of-bounds). This is rarely an option in practice, since a pointer computed inside a loop is usually dereferenced inside the loop, and otherwise hoisted out of the loop by compiler optimizations. We have manually confirmed this for all the vulnerabilities analyzed in Section 7, for which saturation arithmetic would thus not provide any security improvement.

Unaligned access. When a pointer is cast to an arbitrary type and dereferenced, the number of dereferenced bytes may differ from the allocated pointer type. Delta Pointers do not support detection of such *unaligned* accesses, since the situation only arises in the context of a type confusion bug which is not in our threat model. For intellectual curiosity, we have, however, implemented optional support for unaligned access detection in the form of an additional pointer arithmetic that adds the dereferenced number of bytes minus one to the delta tag before each dereferencing instruction. This adds 3% overhead on SPEC 2006.

Delta tag compression. Delta tags take up half the pointer in the current design of Delta Pointers, severely limiting the address space (and thus ASLR entropy) in return for strong memory safety guarantees. Reducing the number of bits needed for the delta tag could alleviate this trade-off, as done in similar schemes. For example, Baggy Bounds [1] compresses object size tags in pointers by allocating power-of-two sized objects, storing only the exponent. Compression for Delta Pointers is not trivial since the delta tag stores the distance to the end of the object rather than the object size. At first sight it might seem possible to compress this by aligning all objects and their size to a number of *compression bits* (n). However, this naive scheme breaks when a pointer is unaligned with respect to its compression. For example, if we align all objects to 8 bytes ($n = 3$), we store 3 fewer bits in the tag. This makes it impossible to update the tag with offsets smaller than 8 bytes, e.g., `p=(char*)malloc(N)+1`. When done iteratively, 8 such pointer additions would overflow the object without being detected.

Arbitrary compression is possible by adding additional arithmetic operations to each pointer modification. The intuition is that, when aligning all objects to n bits, the lower n bits of the delta tag and the address contain the same information. Thus, we can store this information in only the address itself, enabling compression of the tag. Any addition smaller than 2^n only occurs on the address, but if it carries into the $(n+1)$th address bit we continue the addition on the delta tag. The remainder (the part of the offset that is a multiple of 2^n) is added directly to the tag. The following code shows how this is done using bitwise operators, when adding offset a to a pointer while using n compression bits:

```
carry   = ((ptr_old ^ ptr_new ^ a) >> n) & 1
tag_new = tag_old + (a >> n) + carry
```

The first line determines whether the lower n bits of the address carried into the next bit, and the second line replicates the carry on the tag. This scheme offers a larger address space, but incurs memory overhead due to alignment, and runtime overhead due to the additional instrumentation required.

Backend optimization. Section 7 details the performance hit of pointer arithmetic instrumentation on x86-64. Future work could feature an optimization of the LLVM backend that allows for efficient code generation of instrumented pointer arithmetic, better utilizing the scaling-based addressing mode of x86-64 instructions.

Table 1: Comparison of overflow checkers. Most evaluations use (nonoverlapping) sets of benchmarks, making the overhead numbers difficult to compare. The table is categorized by benchmarks used, using a random set of CPU '95, '00, '06 and Olden, CPU2000 and CPU2006 respectively. The reported overhead are geometric means of the respective benchmark suite.

System	C++	Metadata	Checks	Passing OoB pointers	Non-linear	Runtime	Memory
Softbound	✗	Table	Deref	✓	✓	67%	64%
Baggy Bounds	✗	Layout	Arith	✓[a]	✓	72%	11%
PAriCheck	✗	Shadow	Arith	✓	✓[b]	96%	18%
LBC	✗	Shadow	Deref	✓	✗	22%	7.7%
ASan	✓	Shadow	Deref	✓	✗	80%	237%
Intel MPX	✓	Table	Deref	✓	✓	139%	90%
LowFat	✓	Layout	Deref	✗	✓	54%	5.2%
SGXBounds	✓	Tag	Deref	✓	✓	89%	0.1%
Delta Pointers	✓	Tag	–	✓	✓	35%	0%

[a] Only up to $alloc\_size/2$ on 32-bit.
[b] Unless wrap-around on 16-bit labels occurs.

9 RELATED WORK

Overflow detection. There has been a plethora of research on buffer overflow detection over the past decades. Table 1 present a summary of the major systems. Early systems often relied on fat pointers which had a high overhead and introduced many compatibility issues, often requiring programmers to change existing code [4, 22, 32]. The system proposed by **Jones and Kelly** [23] instead relied on external metadata, associated *per-object*. Since such per-object designs retrieve pointer bounds by looking at the objects, they cannot support temporarily out-of-bound pointers, and perform their checks during arithmetic. Later designs attempted to fix these limitations [36] and performance issues [9].

The first practical design was that of **Baggy Bounds** [1], which encodes metadata in the memory layout. Similarly to Delta Pointers, Baggy Bounds also uses pointer tagging to store the size-class of an object, and encodes encodes out-of-bound pointers as invalid (non-canonical) to cause automatic hardware crashes. Instead of masking pointers before every memory access, Baggy Bounds instead places tags for valid pointers in the upper bits of the lower 48 bits, and creates aliased mappings for each tag. This way every tagged pointer is still valid in the address space, with the limitation that even fewer bits are available for both the tag and the pointer. **Low-Fat pointers** [11, 12] takes the baggy bounds design and optimizes it even more by grouping size-classes together in the address space. By sacrificing compatibility of not supporting out-of-bound pointers between contexts Low-Fat can achieve a higher performance. Like any other bounds checker, the Delta Pointers design supports out-of-bound pointers.

Instead of relying on per-object metadata, Delta Pointers rely on *per-pointer* metadata. **SoftBound** [31] transforms the source to keep track of the metadata for pointers, and stores them separately whenever pointers leak to memory. Because of its limited static analysis, SoftBound suffers from a low compatibility. **Intel MPX** [34] achieves a similar design with the support of hardware, introduced with the Skylake microarchitecture but incurs high overhead.

SGXBounds [26] presents a design well geared towards the current version of SGX, which has a limited address space. It encodes the upper bound inside the pointers, with a design based on **Boundless** [6]. Boundless stores distance information similar to Delta Pointers, but only to ultimately compute the upper bound as used by SGXBounds. These systems thus still require branches, and rely on memory accesses to record lower bounds. SGXBounds suffers from a low compatibility due to its limited static analysis, a problem which Delta Pointers address. Delta Pointers could also work well inside SGX enclaves. Outside of enclaves SGXBounds is shown to suffer from much higher overheads.

Some alternative designs do not record the bounds information itself. **PAriCheck** [38] instead tags every few bytes with a label, and during pointer arithmetic enforces that every pointer points to memory with the same tag. Sadly such a design suffers from impractical high overheads. **LBC** [18] and **AddressSanitizer** [37] instead place guard zones around every object, and verify every memory access is outside of a guard zone. Guard zones and their metadata incur a large memory overhead, and moreover, can only detect contiguous buffer overflows.

Pointer tagging. The concept of tagged pointers has been used for decades, but generally concerns the lower bit(s) of a pointer [3, 14]. Baggy Bounds [1], Boundless [6], SGXBounds [26], and Mid-Fat [25] all store tags in the upper bits of pointers. Our design provides a more comprehensive analysis increasing compatibility of pointer tagging over these approaches. Hardware support for pointer tagging can be found in recent architectures with ARMv8's virtual address tagging [2] and Oracle's SPARC-M7 SSM/VA masking [35].

10 CONCLUSION

In this paper we presented Delta Pointers, a design for a fast and compatible buffer overflow detector. In contrast to existing solutions, Delta Pointers do not rely on memory lookups nor branches, yielding a competitive, low-overhead design with 35% performance overhead and negligible memory overhead. Our design relies on pointer tagging to maintain the distance from the current pointer to the end of the object to implicitly invalidate overflowed pointers. We hope our findings on pointer tagging will encourage future research, and as such, our framework and Delta Pointers prototype are available open source.

ACKNOWLEDGMENTS

We thank our shepherd, Cristian Cadar, and the anonymous reviewers for their valuable feedback. This work was supported in part by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI "Dowsing" and NWO 639.021.753 VENI "PantaRhei", in part by the United States Office of Naval Research (ONR) under contract N00014-17-1-2782, and in part by Cisco Systems, Inc. through grant #1138109. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of any of the sponsors or any of their affiliates.

OVERHEADS AND RAW RUNTIMES

Table 2: Normalized runtime overheads of SPEC CPU2006 for Delta Pointers, an implemention using the pointer tagging framework on Delta Pointers but using branches for upper bound checks ("Branches ubound"), and related work.

		Components		Delta Pointers		Branches ubound		Related work	
Benchmark	Lang.	Mask	Mask + tag	No opts	Opts	No opts	Opts	ASan	SGXBounds
400.perlbench	C	1.47	1.51	1.83	1.62	3.34	1.97	4.01	-
401.bzip2	C	1.23	1.24	1.54	1.50	1.72	1.63	1.70	2.22
403.gcc	C	1.14	1.16	1.24	1.23	1.47	1.34	2.23	-
429.mcf	C	1.15	1.15	1.24	1.18	1.36	1.29	1.70	1.58
433.milc	C	1.07	1.08	1.15	1.08	1.27	1.09	1.35	1.59
444.namd	C++	1.17	1.18	1.38	1.32	1.42	1.36	1.52	1.84
445.gobmk	C	1.21	1.31	1.56	1.53	2.17	1.91	1.66	2.18
447.dealII	C++	1.17	1.18	1.40	1.32	1.56	1.32	2.26	-
450.soplex	C++	1.12	1.13	1.29	1.22	1.37	1.22	1.57	-
453.povray	C++	1.37	1.39	1.64	1.52	2.47	2.00	2.72	-
456.hmmer	C	1.26	1.26	1.52	1.53	2.00	1.60	1.91	2.51
458.sjeng	C	1.25	1.29	1.57	1.56	2.47	1.98	1.73	2.32
462.libquantum	C	1.03	1.07	1.09	1.09	1.12	1.11	1.08	1.19
464.h264ref	C	1.31	1.30	1.65	1.39	2.32	1.44	2.15	-
470.lbm	C	1.03	1.03	1.38	1.44	1.21	1.19	1.01	1.63
471.omnetpp	C++	1.16	1.18	1.24	1.22	1.43	1.30	2.09	-
473.astar	C++	1.22	1.30	1.49	1.45	1.76	1.57	1.54	1.84
482.sphinx3	C	1.19	1.17	1.35	1.33	1.74	1.57	1.69	2.24
483.xalancbmk	C++	1.32	1.30	1.44	1.35	1.84	1.64	2.08	2.68
Geomean		1.20	1.22	1.41	1.35	1.72	1.48	1.80	-
Subset sgxbounds		1.17	1.19	1.38	1.35	1.63	1.47	1.55	1.94

Table 3: Raw runtimes of SPEC in seconds, corresponding to the overheads in Table 2. All numbers were gathered on the DAS-5 cluster (https://www.cs.vu.nl/das5/), each number is the median of 16 runs. Note that Delta Pointers and other overflow checkers each require a different build configuration and baseline.

	Baselines			Components		Delta Pointers		Branches ubound		Related work	
Benchmark	LTO	ASan[a]	SGXBounds[b]	Mask	Mask + tag	No opts	Opts	No opts	Opts	ASan	SGXBounds
400.perlbench	262	282	-	386	396	480	424	878	518	1133	-
401.bzip2	440	432	438	542	545	678	662	758	719	736	974
403.gcc	255	259	-	291	296	317	313	374	342	577	-
429.mcf	228	227	227	262	262	283	270	310	295	387	359
433.milc	458	482	491	490	494	525	493	581	501	650	780
444.namd	331	328	328	388	392	457	438	471	451	498	604
445.gobmk	380	404	389	462	498	593	584	827	727	671	850
447.dealII	225	253	-	263	266	315	298	350	297	572	-
450.soplex	197	195	-	220	222	254	239	270	240	306	-
453.povray	118	130	-	163	164	194	180	293	236	356	-
456.hmmer	380	382	385	478	479	577	582	762	610	731	966
458.sjeng	417	427	413	520	536	655	650	1032	828	737	957
462.libquantum	336	339	343	347	358	366	368	375	374	367	409
464.h264ref	468	481	-	612	610	774	651	1085	674	1034	-
470.lbm	353	354	339	362	365	487	509	427	421	357	551
471.omnetpp	280	291	-	326	331	348	342	400	365	607	-
473.astar	319	351	343	388	416	474	463	561	500	542	630
482.sphinx3	453	428	446	540	529	614	604	787	712	724	1001
483.xalancbmk	171	184	179	225	223	246	230	314	280	383	480

[a] No LTO, since ASan does not support LTO on all benchmarks.
[b] Linked against musl, no LTO but all bitcode is combined in a single module with 03, libc++ inlined as bitcode.

REFERENCES

[1] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-compatible Defense Against Out-of-bounds Errors. In *USENIX Security*.

[2] ARM. 2015. *ARM Cortex-A Series – Programmer's Guide for ARMv8-A*.

[3] Mike Ash. 2013. Friday Q&A 2013-09-27: ARM64 and You. (2013). https://www.mikeash.com/pyblog/friday-qa-2013-09-27-arm64-and-you.html

[4] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. 1994. Efficient Detection of All Pointer and Array Access Errors. In *PLDI*.

[5] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *S&P*.

[6] Marc Brunink, Martin Susskraut, and Christof Fetzer. 2011. Boundless Memory Allocations for Memory Safety and High Availability. In *DSN*.

[7] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. 2015. Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine. In *ASPLOS*.

[8] The MITRE Corporation. 2018. Common Vulnerabilities and Exposures. http://cve.mitre.org/. (2018).

[9] Dinakar Dhurjati and Vikram Adve. 2006. Backwards-compatible Array Bounds Checking for C with Very Low Overhead. In *ICSE*.

[10] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. 1998. Type-based Alias Analysis. In *PLDI*.

[11] Gregory J. Duck and Roland H. C. Yap. 2016. Heap Bounds Protection with Low Fat Pointers. In *CC*.

[12] Gregory J. Duck, Roland H. C. Yap, and Lorenzo Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers. In *NDSS*.

[13] Robert van Engelen. 2001. Efficient Symbolic Analysis for Optimizing Compilers. In *CC*.

[14] Edward A. Feustel. 1973. On The Advantages of Tagged Architecture. *IEEE Trans. Comput.* 22, 7 (July 1973).

[15] Robert Gawlik, Benjamin Kollenda, Philipp Koppe, Behrad Garmany, and Thorsten Holz. 2016. Enabling Client-Side Crash-Resistance to Overcome Diversification and Information Hiding. In *NDSS*.

[16] Enes Göktaş, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, , Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. 2016. Undermining Entropy-based Information Hiding (And What to do About it). In *USENIX Security*.

[17] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*.

[18] Niranjan Hasabnis, Ashish Misra, and R. Sekar. 2012. Light-weight Bounds Checking. In *CGO*.

[19] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* (2006).

[20] Intel Corporation. 2017. *Intel 64 and IA-32 Architectures Software Developer's Manual*.

[21] ISO/IEC. 1999. International Standard ISO/IEC 9899:1999 (E) - Programming Language C. (1999).

[22] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *USENIX ATC*.

[23] Richard WM Jones and Paul HJ Kelly. 1997. Backwards-compatible bounds checking for arrays and pointers in C programs. In *AADEBUG*.

[24] Tobias Klein. 2009. CVE-2009-0385. (February 2009).

[25] Taddeus Kroes, Koen Koning, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. 2017. Fast and Generic Metadata Management with Mid-Fat Pointers. In *EuroSec*.

[26] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. SGXBounds: Memory Safety for Shielded Execution. In *EuroSys*.

[27] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*.

[28] David Litchfield. 2005. *Buffer Underruns, DEP, ASLR and improving the Exploitation Prevention Mechanisms (XPMs) on the Windows platform*. Technical Report. NGSSoftware Insight Security Research (NISR).

[29] Matt Miller. 2017. Heap corruption issues reported to Microsoft. (2017). https://twitter.com/epakskape/status/851479629873332224

[30] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation*.

[31] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *PLDI*.

[32] George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-safe Retrofitting of Legacy Code. In *POPL*.

[33] Angelos Oikonomopoulos, Cristiano Giuffrida, Elias Athanasopoulos, and Herbert Bos. 2016. Poking Holes into Information Hiding. In *USENIX Security*.

[34] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches. *arXiv* (2017).

[35] Oracle. 2014. M7: Next Generation SPARC. (2014).

[36] Olatunji Ruwase and Monica S. Lam. 2004. A Practical Dynamic Buffer Overflow Detector. In *NDSS*.

[37] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC*.

[38] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. 2010. PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs. In *ASIACCS*.