Mitigating Information Leakage Vulnerabilities with Type-based Data Isolation

Alyssa Milburn[†*], Erik van der Kouwe[*] and Cristiano Giuffrida[*]
[†]Intel [*]Vrije Universiteit Amsterdam, The Netherlands
amilburn@zall.org, {vdkouwe,giuffrida}@cs.vu.nl

***Abstract*—Information leakage vulnerabilities (or simply *info leaks*) such as out-of-bounds/uninitialized reads in the architectural or speculative domain pose a significant security threat, allowing attackers to leak sensitive data such as crypto keys. At the same time, such vulnerabilities are hard to efficiently mitigate, as every (even speculative) memory load operation needs to be potentially instrumented against unauthorized reads. Existing confidentiality-preserving solutions based on data isolation label memory objects with different (e.g., sensitive vs. nonsensitive) colors, color load operations accordingly using static points-to analysis, and instrument them to enforce color-matching invariants at run time. Unfortunately, the reliance on conservative points-to analysis introduces overapproximations that are detrimental to security (or further degrade performance).**

In this paper, we propose Type-based Data Isolation (TDI), a new practical design point in the data isolation space to mitigate info leaks. TDI isolates memory objects of different colors in separate memory *arenas* and uses efficient compiler instrumentation to constrain loads to the arena of the intended color by construction. TDI's arena-based design moves the instrumentation from loads to pointer arithmetic operations, enabling new aggressive speculation-aware performance optimizations and eliminating the need for points-to analysis. Moreover, TDI's color management is flexible. TDI can support a few-color scheme with sensitive data annotations similar to prior work (e.g., 2 colors) or a many-color scheme based on basic type analysis (i.e., one color per object type). The latter approach provides fine-grained data isolation, eliminates the need for annotations, and enforces strong color-matching invariants equivalent to ideal (context-sensitive) type-based points-to analysis. Our results show that TDI can efficiently support such strong security invariants, at average performance overheads of <10% on SPEC CPU2006 and nginx.

I. Introduction

Despite advances in security engineering, information leakage vulnerabilities (or *info leaks*) remain a major security threat. Modern systems software is riddled with info leak bugs [59] and Spectre-based variations [32] have expanded the already large attack surface. Unfortunately, existing mitigations that aim to significantly reduce such attack surface incur nontrivial performance costs. In this paper, we show such costs are not fundamental and an efficient, fine-grained data isolation strategy based on secure allocation and lightweight compiler instrumentation can mitigate info leaks, with single-digit performance overhead for practical cases of interest.

a) The info leak era: Info leaks based on spatial (out-of-bounds, type confused reads) or temporal (uninitialized, use-after-free reads) memory errors are crucial in modern software exploitation [59]. Such vulnerabilities enable attackers to leak private data such as crypto keys (e.g., Heartbleed [50]). Moreover, they enable reliable ROP [60] by allowing attackers to bypass mitigations such as ASLR and stack cookies, or by leaking a massaged memory object location [59]. While info leaks in the architectural domain have dominated software exploitation in the last decade, the attack surface has recently expanded to the speculative domain with Spectre [32]. Spectre-BCB (Bounds Check Bypass) is a widespread example of an out-of-bounds read vulnerability using speculative execution.

b) Mitigating info leaks: Mitigating info leaks in a practical way is notoriously difficult. Mitigations that entirely eliminate the attack surface in the architectural (e.g., memory safety [74]) and speculative (e.g., load fencing [49]) domain are expensive and normally out of reach of the performance budget available in production settings. More practical confidentiality-preserving solutions described in literature are based on data isolation: isolating memory objects in the address space to make them inaccessible from other objects vulnerable to info leaks [12], [15], [38], [51], [65]. Such solutions generally color memory objects and load operations based on the color of the objects they are allowed to access, as dictated by static points-to analysis. Loads are then instrumented to enforce such color-matching invariants at run time by means of pointer masking [15], [33], domain switching [12], [33], [38], [51], [53], [62], [65] or run-time checks [12], [15], [33]. Some (not all) of these techniques (e.g., pointer masking) are also Spectre-safe. Some coarse-grained solutions use a few, often two (sensitive vs. nonsensitive) colors set by user annotations (e.g., labeling an allocation site for crypto keys as sensitive) [12], [15], [51], while other fine-grained solutions use many colors, based on the clusters automatically determined by points-to analysis [7], [20], [62].

Regardless of the particular scheme, existing data isolation techniques—barring those targeting very specific code patterns [33]—rely on static points-to analysis to determine the set of possible targets of load operations. Since such analysis is conservative and context-insensitive (other than having trouble scaling to large programs), the set of possible targets is often largely overapproximated even in state-of-the-art implementations such as SVF [63] or data isolation-tailored ones such as DataShield's [15]. Such overapproximations are problematic for either security, as they lead to much weaker color-matching invariants, or for performance, when additional metadata-based run-time checks are used to compensate for

[†]Alyssa currently works at Intel; this work was done at the VU.

this weakness [7], [15]. Even without expensive run-time checks, the cost of instrumenting pervasive load operations for data isolation is nontrivial. For example, generic load pointer masking-based solutions with only two colors incur over 17% average overhead on SPEC CPU2006 [33].

c) TDI: In this paper, we present Type-based Data Isolation (TDI), a new design point in the data isolation space with strong performance and security guarantees against both architectural and speculative info leak vulnerabilities. TDI's key insight is that we can eliminate expensive load-based instrumentation and imprecise points-to analysis if we rearrange the address space layout and constrain pointers within specific address ranges. In particular, TDI allocates independent memory regions (*arenas*) for each memory object color, both on the heap and stack, and then uses lightweight compiler instrumentation to ensure each pointer of any given color stays within its arena (i.e., object color) by construction. TDI's design provides several benefits compared to prior work.

First, our arena-based strategy moves the compiler instrumentation from loads to pointer arithmetic operations. Not only does this eliminate any dependency on context-insensitive points-to analysis (which would degrade precision and security), it also provides much better performance. Intuitively, since many load operations depend on the same (or similar) computed pointers we can reason about, this significantly reduces the number of instrumentation points. Moreover, as we will show, such strategy is particularly amenable to other optimizations, such as efficient masking and the use of inter-arena *guard zones*. While similar optimization techniques have been explored by traditional SFI solutions [24], [35], [41], [57], [73], we show that TDI's unique pointer arithmetic design enables much more aggressive optimizations, significantly outperforming prior work. We also detail and address the challenges of making our optimizations speculation-aware.

Second, our design (and instrumentation) is entirely agnostic to the object coloring scheme. Specifically, TDI supports arbitrary coarse- or fine-grained (i.e., many-color) isolation schemes—with colors determined by explicit user annotations or static analysis—despite significantly outperforming prior (annotation-based) data isolation solutions limited to coarse-grained (e.g., 2-color) schemes. By default, TDI uses simple static type analysis [66] to isolate each individual object type in its own arena. This scheme supports annotation-free protection and provides very fine-grained isolation—well beyond object coloring based on the clusters determined by points-to analysis used by WIT [7] and others [20], [62].

With such a scheme, architectural or speculative info leak vulnerabilities cannot be exploited to leak data across any two given object types. For example, a crypto key can never be leaked by means of a vulnerable string or buffer of any other type. We show that TDI can flexibly protect such situations in OpenSSL with both coarse- and fine-grained coloring.

Third, our design eliminates the need for imprecise and hard-to-scale points-to analysis altogether. Our masking instrumentation does not rely on any particular object coloring scheme, as we simply constrain pointers within their predetermined arena rather than attempting to enforce color-matching invariants by reasoning about the targets of load operations. This strategy matches the precision of the underlying object coloring scheme, with no overapproximations. As a result, our standard configuration using per-type arenas can enforce color-matching invariants equivalent to load-side counterparts using ideal (context-sensitive) type-based points-to analysis.

To summarize, our contributions are:

- We design and implement a prototype[1] of TDI, a low-overhead Type-based Data Isolation system based on lightweight compiler instrumentation.
- We explore the challenges of efficiently implementing such instrumentation, presenting aggressive but speculation-aware optimizations allowing TDI to be applied to real-world code with low performance overhead.
- We automate TDI's object coloring using state-of-the-art type analysis, resulting in a fine-grained isolation system that aggressively contains info leak vulnerabilities in both the architectural and speculative domain.
- We evaluate our TDI prototype using standard benchmarks and the modern nginx web server. Our results show TDI incurs only single-digit average performance overhead on SPEC CPU2006 and nginx.

II. Threat model

We assume a typical modern software exploitation scenario with an attacker exploiting either spatial (out-of-bounds reads, type confused reads) or temporal (uninitialized reads, use-after-free reads) info leak vulnerabilities while all the standard modern mitigations, such as ASLR, DEP, stack cookies, etc., are in place. The attacker has not (yet) achieved control-flow hijacking, and aims to leak private data (e.g., crypto keys) or information needed to hijack control (e.g., pointers or stack cookies). The attacker can exploit both classical and speculative info leak vulnerabilities. A typical example in the former category would be a classical out-of-bounds read with an attacker-controlled value which is not bounds-checked before being used to index an array. A typical example in the latter category would be a speculative out-of-bounds read with an attacker-controlled value which is only architecturally bounds-checked before being used as an array index. An attacker may speculatively bypass the bounds check and leak data using a (e.g., cache) covert channel a la Spectre-BCB [32]. While we mostly focus on speculative out-of-bounds reads used by the widespread Spectre-BCB variant, all the other classical info leak vulnerabilities exploited in the speculative domain (e.g., speculative type confusion) are in scope. Other unrelated Spectre (e.g., Spectre-BTB [32]) or transient execution variants (e.g., MDS [67]) are out of scope and addressed by complementary (e.g., hardware) mitigations. We also consider other vulnerabilities (e.g., memory corruption) out of scope.

III. Overview

TDI hardens C/C++ programs by preventing pointers from escaping the memory area—'arena'—in which they were

[1] Our current code can be found at https://github.com/vusec/typeisolation.

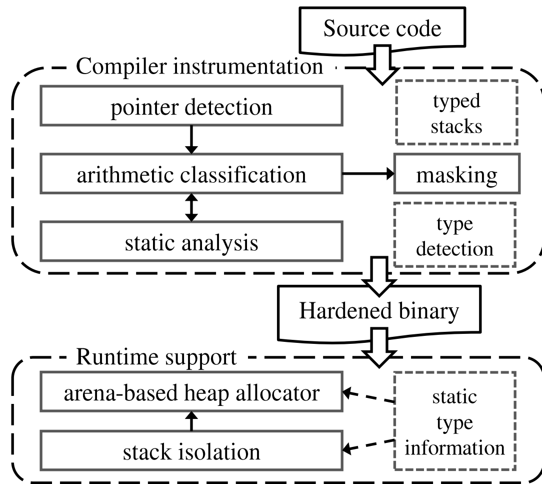

Fig. 1. High-level overview of TDI.

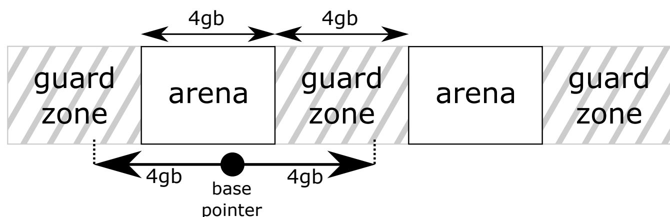

Fig. 2. Overview of TDI's arena layout, with guard zones of ≥4GB.

allocated. Our design, as shown in Figure 1, relies on both compiler instrumentation and runtime code. At compile time, we instrument all pointer arithmetic to ensure that all pointers stay in their original arena. At run time, our arena-based allocator allows programs to allocate memory in appropriate isolated heap/stack regions. Besides explicit program annotations, TDI also offers support to automatically allocate both heap and stack objects in arenas based on their type.

We constrain pointers to their original arena by *masking* pointers to preserve the upper 32 bits during pointer arithmetic (i.e., we only allow the lower 32 bits to change), as shown in Listing 1. As long as all arenas are at least 4GB in size and appropriately aligned, this ensures that pointers always point to the same arena both before and after pointer arithmetic.

Masking pointers after every individual instance of pointer arithmetic would be very inefficient; for example, a pointer used to access successive elements of a struct or array would need to be repeatedly re-masked. As shown in Figure 2, we relax the need for masking by adding guard zones around each arena. If a base pointer is known to be inside an arena, then we know that any pointer within 4GB is either in the same arena, or in a guard zone. This insight allows us to optimize TDI by identifying and removing unnecessary masking.

IV. Instrumentation

In this section, we discuss the design of TDI's compiler instrumentation, which enforces our security guarantees by

```
char myFunction(char *validPtr, size_t idx)
  char *newPtr = validPtr + idx;
  upperBits = (validPtr & 0xffffffff00000000);     ①
  lowerBits = (newPtr & 0xffffffff);
  newPtr = upperBits | lowerBits;
  char ret = *newPtr;                               ②
  return ret;
```

Listing 1: This pseudocode shows a potentially-unbounded memory access ②. TDI's instrumentation (marked with a dark background) preserves the upper bits of newPtr ①. This ensures that the access at ② cannot be further than 4GB from validPtr, and so cannot escape validPtr's arena.

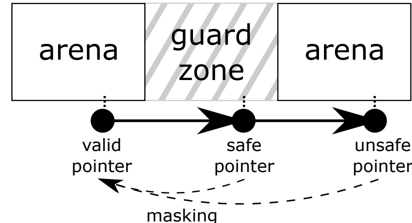

Fig. 3. Valid, safe and unsafe pointers; safe pointers are always within 4GB of a valid pointer into the same arena, which means that while they *may* point into a guard zone (as illustrated), they will never point to a different arena. If the result of pointer arithmetic may lie in a different arena, we call the result an *unsafe* pointer and mask it before use (as shown in Listing 1).

preventing pointers from crossing arena boundaries. Although our design is not compiler-specific, we discuss some of the details and challenges in the concrete context of LLVM.

TDI allocates each memory object (whether stack or heap) in an arena based on its type; the 32 most significant bits of each pointer identify its arena. The result of pointer arithmetic may end up in a different arena, which would allow an attacker to break isolation. We prevent this using pointer masking; however, masking after every instance of pointer arithmetic results in unacceptable overhead. To determine which pointers we must mask, we divide pointers in three classes:

- A *valid pointer* can be proven to be in the same arena in which the pointer from which it derives was allocated;
- A *safe pointer* either points to the same arena or to a guard zone, in which case a dereference will fault;
- An *unsafe pointer* may point to a different arena.

These three classes are illustrated in Figure 3.

We can identify valid pointers based on their sources; for example, allocation functions always return valid pointers. If we can prove that the result of pointer arithmetic is within 4GB of the original pointer, we can classify that result as safe; otherwise, such a result is unsafe.

Dereferencing a valid or safe pointer does not threaten security, but we must ensure that unsafe pointers are not dereferenced by loads or stores.

The *base pointer* of a pointer is the most recent known valid pointer that a pointer derives from. We can ensure that a pointer is valid by overwriting the high-order bits (which represent the arena) with those of its valid base pointer,

i.e. *masking* the pointer, as described above. Such a masked pointer is always valid, since it points inside the same arena as the valid base pointer, and can then be safely dereferenced.

To ensure a valid base pointer is always available, we only allow safe pointers to be used locally within a function; any pointer which escapes a function's scope (e.g., by being stored to memory, returned from a function, or passed as a parameter to another function) must be valid (i.e., must be masked if necessary). This means we can determine pointer categories with only intraprocedural analysis, since any pointers from outside the scope of the function must be valid.

To summarize, we mask pointers in two situations:

- When an unsafe pointer may be used by a load (or optionally, as the address used by a store). This means we cannot prove that it is safe (<4GB away from a valid pointer), and may now be pointing to a different arena.
- When a pointer value escapes the local analysis. For example, when a pointer is stored in memory, passed as a function argument or used as a return value. This ensures that all pointers entering a function are themselves valid.

A. Challenges

Our instrumentation identifies pointer arithmetic and classifies the results as valid, safe, or unsafe. Doing this analysis on real-world code must overcome the challenges below.

1) Arithmetic on non-pointer types: Since pointers can be cast to/from other types, such as integers, we need to distinguish pointer arithmetic from non-pointer arithmetic; we want to mask all resulting pointers, but pointer arithmetic is not always intended to result in a valid pointer. For example, pointers may be subtracted to obtain a delta. Even though the result of such arithmetic may escape local scope, the result must not be masked by our instrumentation, since that would produce incorrect code. Compiler passes also manipulate pointers, including converting them to untyped values.

We resolve this by doing *pointer detection*, allowing us to find which variables/intermediates are truly (non-)pointers.

2) Non-constant offsets: Array indexes or other pointer offsets are often non-constant. Our efficiency relies on being able to prove that such offsets are within 4GB of a known-valid pointer, allowing us to mark the result as safe; however, offsets are often stored in 64-bit variables.

Pointer arithmetic is often performed on the result of previous pointer arithmetic; since safe pointers are *not* necessarily valid, efficient instrumentation also requires that we handle such 'chained' arithmetic. Pointers may only be modified and/or used in some program paths, making it more difficult to reason about their behavior. A common example is a pointer which is incremented inside a loop, as we see below.

We resolve this using the static analysis we describe below, *range analysis* and *dominating pointer access analysis*.

3) Speculative execution: Since we include speculative execution (Spectre) in our threat model, bounds provided by existing static analysis can be unsafe. For example, LLVM's ScalarEvolution (SCEV) will use array bounds checks to prove that an array access is in-bounds, but that array may still be accessed on a transient path. We resolve this by ensuring our custom analysis is valid in the context of speculative execution.

B. Pointer detection

We found that, in many cases, variable types marked in the original source code and in the compiler IR do not accurately reflect whether a variable is used as a pointer or not. To ensure masking wherever needed while retaining compatibility with existing code, we designed static analysis to detect pointer/non-pointer status. Our main insight is that variables should be classified (where possible) based on how they are used rather than how they are declared.

Our approach marks variables as pointers, non-pointers, or negated pointers in several steps. In each step, we mark variables based on their usage (if their type is still unknown), and then we use forward and backward propagation to infer the type of other variables. For example, our first step considers all pointer dereference operations and marks their operands as pointers. The main idea of our propagation is that a pointer plus an offset is still a pointer, a pointer minus another pointer (or plus a negated pointer) is an offset, and other operations typically yield non-pointers. Details are in Appendix C.

C. Categorization

To be able to apply efficient masking, we categorize all pointer arithmetic results we detect as valid, safe, or unsafe. We start with the first such instructions in a given function and proceed top-down (based on domination). Since the classification of arithmetic can depend on other arithmetic, we can also temporarily place them in a group of *unknown* arithmetic. We repeat this process until all arithmetic has been classified; if there are only circular dependencies left, we mark the first remaining arithmetic as unsafe and continue.

We only need to mask pointers which are dereferenced by a load (or store) instruction, are used in pointer arithmetic, or which escape the local function. This includes both direct and indirect uses (including integer casts which are later used by arithmetic). We ignore instructions which are not used in such ways, which removes almost all ambiguous cases of pointer arithmetic. We discuss some remaining cases in our evaluation.

At this stage, we apply *dominating pointer access* analysis, which we describe in Section IV-F, which can prove that some pointers are valid or safe. Otherwise, we trace the instruction's base pointer. If the source is outside the local scope (a function argument, returned from another function, or loaded from memory), we consider the source to be valid. If the source is a pointer arithmetic instruction (or a merge of several such instructions), then we use the classification of that instruction (and defer processing if that instruction has not yet been classified). Otherwise, we consider the source to be unsafe.

Our classification for the result of pointer arithmetic then depends on the classification of the base pointer: (1) a valid base pointer means that the result is safe; (2) a safe base pointer means that the result is unsafe (and we will mask the pointer before using it); (3) an *unsafe* base pointer must itself be masked. Since a masked pointer is valid, the result of

```
void func(char *validPtr, size_t idx) {
   char *ptr = validPtr;
   if (...)
①    ptr = ptr + 2;
   else
②    ptr = ptr + 4;
③ char val = *(ptr + 1);
}
```

Listing 2: The access at ③ is safe, because the maximum offset to validPtr is less than 4GB.

the arithmetic is then safe. If we cannot prove that an offset is bounded to 4GB (nor, as described below, truncate the offset to enforce this), we mark the current arithmetic result as unsafe.

D. Range analysis

One important supporting component of our analysis involves determining the maximum range of offsets used in pointer arithmetic. We require a Spectre-BCB-aware analysis which calculates the worst-case (largest) distance to a known-valid pointer on all paths within a function, which prevents using standard compiler analysis (such as LLVM's SCEV). Instead, we designed an alternative analysis which performs a recursive check of all conditional control flows. We calculate the bounds by considering instructions such as truncations, arithmetic (e.g., AND operations), the bitwidth of loads/variables, and sources such as constant values. When merging several possible bounds (e.g., phi nodes), we use the worst-case bound among all incoming values.

One special case is where we can prove a 32-bit bound, i.e. a maximum offset of 4GB, which is multiplied by the object size, such as during an array lookup. We can ensure that such an index is always safe by ensuring that the guard zone is at least 2^{32}*sizeof(type) bytes (see Section VI). The majority of such calculations are performed for types ≤ 8 bytes (64-bit pointers or doubles); if our arenas use 32GB guard zones, then we can classify all scaled 32-bit offsets for such types as safe, assuming the base pointer is known to be valid.

When we are using this range analysis to determine whether a pointer is safe, and the distance to a valid pointer cannot be proven to be less than 4GB, simply truncating the offset in bytes to 4GB (32 bits) is sufficient to ensure that the resulting pointer is safe. Note that masking is still required to make it valid, and that truncation is unnecessary if the analysis later decides to mask this pointer.

E. Chaining distances

A pointer is safe when the distance to a valid pointer is known to be less than 4GB. Even when pointer arithmetic is based on a safe pointer, we can compute bounds using the offsets of previous arithmetic, and use that information to prove that the result is still within 4GB of a known-valid pointer. A simple example is shown in Listing 2. Here, our analysis checks the phi node for `ptr` at ③, which has incoming values from ① and ②, and concludes that the maximum offset to a known-valid pointer is less than 4GB even though the intermediate pointers may not be valid.

```
void func(char *validPtr, int idx) {
   char *A = validPtr + idx;
   char *B = A + 1024;
① char valA = *A;
② char valB = *B;
}
```

Listing 3: After ①, pointer A is known to be valid, so when execution reaches ②, we know that Pointer B is safe.

```
void func(char *validPtr, size_t size) {
① char *ptr = validPtr;
   for (size_t n = 0; n < size; n++) { ②
③   ptr = ptr + 1;
④   char val = *ptr;
   }
}
```

Listing 4: The pointer arithmetic at ③ is dominated by the loop header at ②. Since all candidates for ptr are valid in the context of ②, we know that ptr is safe at ④.

Again, standard compiler analysis could provide this information (by calculating the distance between pointers), but it does not consider speculative flows. Instead, we use our own (simple) control-flow-insensitive distance analysis, which is also needed to support several the other analysis stages.

F. Dominating pointer accesses

When a safe pointer is dereferenced to access (load or store) memory, any code after that memory access can assume that the used pointer was valid. If a safe pointer is *not* valid, then it must point into a guard zone; after such a pointer is accessed, a fault will occur, and execution will not continue.

This allows us to improve our categorization of pointers for all code dominated by (guaranteed to run after) a memory access. Existing compiler static analysis does not (easily) provide the information we need. Alias analysis focuses on proving that pointers are *never* pointing to the same *object*, while our analysis must prove that pointers are *always* pointing to the same *arena* (i.e. within 4GB of a valid object).

Listing 3 provides a simple example. The access at ② is dominated by the access at ①. Since the pointer B at ② is less than 4GB away from the pointer A used at ①, then B could be categorized as safe in the context of ②, and architecturally it could be accessed without masking. However, note that in this particular case, in the face of a Spectre attack, B might still be dereferenced speculatively even if A points to a guard zone. This limits the applicability of this type of optimization in a threat model where we must also consider transient attacks.

However, only a small number of loads/stores will be speculatively queued [40]. As long as we limit the distance between successive accesses (we limit it to 64kB), and ensure that base pointers are *always* valid (masking them where needed), we can make such optimizations speculation-aware.

For example, consider Listing 4, where `ptr` is incremented inside the loop. On the first iteration of the loop, the new `ptr` at ③ is based on the valid pointer from ①. On later iterations, the new `ptr` at ③ is dominated by the previous access at ④.

```
void func(char *validPtr, size_t size) {
  for (size_t n = 0; n < size; n++) {
      char val = *(validPtr + n);
  }
}
```

Listing 5: Each access in the loop is safe, due to being at most 1 byte away from a known-valid pointer.

```
; RAX: valid pointer, RCX: pointer to be masked
xor %eax, %ecx ; xor low 32 bits (clears upper bits)
xor %rax, %rcx ; xor all bits
```

Listing 6: Example x86-64 code for pointer masking.

We can check whether all incoming values at ② (the loop header) are valid in the context of ④, i.e. whether *all* the potential values of `ptr` are valid at this point. This allows us to determine that the access at ④ is safe.

Finally, we consider Listing 5, which uses a loop induction variable rather than modifying the pointer in the loop. Even though we avoid use of non-speculative-safe analysis, we can analyze simple cases where pointers are being offset by a fixed stride based on a loop induction variable. In this example, we detect that the offset `n` is a loop induction variable which increments (or decrements) at each iteration, and that the distance to the known-valid pointer from the previous loop iteration is <4GB (or, again, a fraction of this due to speculative safety). This allows our analysis to confirm that the access is valid in the context of the loop. Although in practice the involved strides are often much larger (e.g., stepping over arrays of large structs), the same reasoning applies.

TDI provides static analysis for each of the three situations described above, and uses the information obtained to mark pointers as valid or safe. Importantly, we chain the analysis described above; for example, we can detect accesses which are close to a known-valid access, which in turn was based on loop induction variable analysis. Combined, this allows us to significantly improve the performance of many inner loops and other performance-sensitive code, by removing unnecessary masking where we can prove pointers to be safe.

G. Masking

Finally, although our design relies on the static analysis described above to avoid the need for masking where possible, our instrumentation also needs to efficiently emit code for applying masks where it cannot be avoided.

We can efficiently mask pointers on both x86-64 and AArch64 by making use of implicit clearing of the upper 32 bits of 64-bit registers, when they are used as 32-bit registers. A 32-bit XOR of a valid pointer with a (potentially) unsafe pointer, followed by a 64-bit XOR, will preserve the lower bits of the unsafe pointer, but overwrite the upper 32 bits.

We found the code emitted by compilers for this sequence to be very efficient. For example, the code in Listing 6 is a typical x86-64 code sequence emitted by LLVM for our pointer masking; bitmask-based arithmetic is used in cases where such a sequence would be inefficient.

V. Arena-based allocation

The only requirement that TDI imposes on allocated objects is that preserving the upper bits of a pointer must not result in corruption of valid pointers; that means that the lower N bits of any pointer must remain constant for the entire object, and thus that allocated objects must not exceed the arena size. If necessary, larger objects can be supported by using a larger arena size, or relaxing checks in some circumstances; we discuss some real-world examples later.

TDI relies on objects being allocated within appropriate arenas. Source code annotations and runtime calls could be manually added to source code to allocate objects in appropriate arenas. However, this requires a substantial investment in time, and only protects a subset of code/data. Instead, TDI provides automatic type-based isolation based on type information from previous work [66].

For heap allocations, our arena-based allocator allows objects to be allocated in a specific arena. Where type (or callsite) information is available, TDI allocates each type of object in a different arena. Otherwise (e.g., allocations by uninstrumented libraries) we allocate in a generic untyped arena, isolated from other arenas. All arenas are allocated dynamically, allowing isolation decisions to be made at runtime.

We allocate the stack in an independent arena, isolating it from other arenas. We can also enforce type-based allocation on the stack, by isolating stack objects in typed arenas.

To support programs which use out-of-bounds pointers which are just after or before the valid memory range of an object, we also reserve some space at the start and end of every arena. This is required because our instrumentation can mask such pointers to ensure they are valid; a subsequent unmasked use (as a safe pointer within 4GB) may end up accessing the guard zone rather than the intended object.

VI. Implementation

A. Compiler instrumentation

We built our prototype implementation based on LLVM 9.0 (together with clang), using a pass (~ 2500 SLOC) that implements the static analysis and categorization described in Section IV-C, along with instrumentation of unsafe arithmetic and/or offset truncation. Our distance and loop analysis is currently only supported on GEPs. We perform our transformation at LLVM IR level, running our pass (and some support passes) after other LLVM IR transformation passes are complete.

Our pointer detection uses type-based alias analysis (TBAA) metadata (added by clang) to help find loads of pointers (even when typed as integers). We treat vectors of pointers as pointers, instrumenting arithmetic on vectors where needed; our masking is often overly conservative since some of our analysis does not support vectors, and we mask all pointers inserted into aggregates (structs and LLVM-level arrays).

B. Arena allocation

We implemented an arena-based allocator on top of tcmalloc [25]. We chose to build our allocator on tcmalloc to allow fair performance comparisons with a non-isolated baseline, but

we could also modify an existing arena-based allocator such as PartitionAlloc to provide the needed behavior. We partially based our work on the patches from Type-after-Type [66].

We do not use the first and last tcmalloc page (8K) of each arena, so that pointers which are just before or after an allocated object are left unmasked; this also keeps stack pointers inside arenas. If an arena runs out of space, we allocate additional arenas with new guard zones.

As discussed in Section IV-D, we optimize typical index scaling cases by mandating 32GB guard zones; this allows $\sim$ 3600 4GB arenas in the typical 47-bit userspace address space of current x86-64 processors (see Section IX-D).

The addresses $\leq$32GB should be left unmapped, to prevent a (valid) NULL pointer from being used to access an arena. This requires building binaries as position-independent executables (PIE) – the standard for most recent Linux distributions. A similar relocation is required for LLVM's SLH mitigation.

C. Type-based isolation

We use Type-after-Type [66] (TAT) to obtain type information. TAT detects types at allocation sites based on C-level data type information including type casts and `sizeof` operators. When such data type analysis fails (i.e., for untyped `char` allocations), TAT resorts to using the callsite ID as the type.

To deal with custom allocator wrappers, TAT conservatively detects and aggressively inlines them *before* the type analysis. This strategy adds context sensitivity to the analysis, boosting precision and resulting in fine-grained type identification without run-time tracking [6]. In particular, this reduces the number of untyped allocations and also ensures the residual untyped allocation callsite IDs yield one type per allocation context (including custom wrappers) rather than one per allocator call.

We ported TAT to LLVM 9.0, fixed various bugs, expanded its allocation function support, and modified its runtime library to allocate safe stacks in typed arenas. Stack pointers are stored in per-thread arrays; static indices for each type are assigned during LTO (link-time optimization).

SafeStack (which TAT builds upon) uses SCEV to statically determine whether accesses are in-bounds (and thus safe). Due to speculative flows, we only allow this for constant offsets; this results in more objects being placed on typed stacks.

TDI reduces the number of such moved objects using a custom interprocedural analysis pass which marks function pointer arguments which are only accessed in-bounds. This allows some objects – in particular, pointers to variables used to store lengths/sizes – to remain on the safe stack.

VII. Evaluation

We first consider some examples of TDI's mitigation of classical and Spectre-BCB vulnerabilities, and then provide an evaluation of benchmarks (SPEC CPU2006/2017) and a web server (nginx with OpenSSL). We consider three basic configurations in all our evaluations:

- Typed allocation: We apply type analysis and allocate all heap and stack objects in typed arenas. This could be seen as an spatial extension of Type-after-Type.
- Masking: We instrument pointer arithmetic; since this requires arena alignment, we run with our arena-based allocator placing everything in a single heap arena; similarly, all stack contents are a single arena.
- Full protection: We apply our complete defense, including typed allocation and instrumenting pointer arithmetic.

A. Vulnerabilities

To confirm TDI's protection against real-world vulnerabilities, we checked relevant issues in the CVE database; here, we discuss a small selection to illustrate the different ways in which TDI can mitigate issues:

CVE-2016-1234 (glibc): Linear stack buffer overflow in glob. Although we cannot build all of glibc, we built glibc's glob.c using TDI after removing some clang-incompatible lines from headers. The stack buffer is identified as having a unique type and is allocated in an isolated arena, which would prevent the overflow from being exploitable.

CVE-2018-16845 (nginx): *ngx_http_mp4_read_atom* subtracts a header size from an (unchecked) value read from an mp4 file, resulting in an integer overflow. One potential exploitation path uses the 'trak' parser to recurse into the vulnerable function, allowing control of *end* and potentially adding a 64-bit offset to *buffer_pos* (a pointer to the input buffer). We confirmed that, as expected, TDI masks this (and other) pointer arithmetic. Since the buffer is allocated from a unique callsite and thus is placed in a separate arena, this appears to fully prevent exploitation of the bug.

CVE-2018-16890 (curl): Integer overflow allows an attacker to disclose data (via a NTLMv2 response) via an attacker-controlled 32-bit array index. The array is allocated by a malloc call inside `Curl_base64_decode`; again, TDI does not need to apply any masking, since heap arena isolation prevents an attacker from disclosing any data except other allocations from that callsite.

CVE-2019-3859 (libssh2): This CVE covers several different issues; we consider one in `kex.c`. Attacker-controlled 32-bit lengths provided during SHA1 key exchange are not checked, potentially leading to reads beyond the end of a buffer; the sum of the offsets can be >4GB, but TDI masks the pointer arithmetic and mitigates the vulnerability.

B. Spectre-BCB

To ensure that our instrumentation is applied to potential Spectre-BCB gadgets, we applied TDI to the corpus of 27 Spectre v1 variants provided by the authors of Spectector [27] (including the 15 examples from Kocher [31]). TDI correctly masks the potentially out-of-bounds loads for all 27 examples.

We also examined four Spectre demos from Google's Safeside [1] suite, which we modified to allocate the private (secret) string using malloc. Originally, the public and private strings were both static global strings, stored in the same arena. We also made changes to prevent truncation or masking when calculating cross-arena offsets, which would typically be

attacker-supplied rather than calculated by the code itself, and confirmed that the examples work when TDI is not applied[2].

We mitigate three of these four examples:

1) spectre_v1_pht_sa: This is a Spectre-BCB example, which is covered by our threat model; as expected, the private string no longer leaks when TDI is applied, since the array access is correctly masked.

2) spectre_v1_btb_sa: This example uses a mispredicted indirect branch to cause type confusion. Even though this is not covered by our threat model, the private string no longer leaks when TDI is applied. The transient (mispredicted) branch target uses an out-of-bounds read and TDI prevents it from accessing the private string. (If we modify the code to remove the out-of-bounds read, the example leaks the private string after TDI is applied, as expected.)

3) spectre_v1_btb_ca: This uses a mispredicted indirect branch to transiently execute code to read the private string. Since the transiently executed code is intended to be able to read the private string, this is outside our threat model, and the code leaks the private string even after TDI is applied.

4) spectre_v4: This example is intended to demonstrate Spectre-SSB. It causes an out-of-bounds array index to be transiently used while waiting for a store to complete. Since the array index is out-of-bounds, TDI masks the array access and the private string no longer leaks.

C. SPEC CPU2006 and CPU2017

We evaluated the performance of TDI using SPEC CPU2006, to aid comparison with previous work. We also present results from SPEC CPU2017 (without OpenMP). We ran these evaluations on Xeon E5-2630 v3 CPUs with 64GB of RAM. Transparent Huge Pages were disabled and the benchmarks were pinned to a single core. In both cases, we include all C/C++ benchmarks and use the reference SPECspeed data. We run each benchmark/configuration at least 5 times; the reported numbers are the median value from these runs.

We modified some of the benchmarks to make them build and run successfully with TDI. We also applied these changes to the baseline where relevant. These changes can be grouped into three categories (details are in Appendix A):

- build problems: we added an #include to dealII's code.
- undefined pointer arithmetic: we apply gcc patches and exclude one perlbench function.
- large allocations in CPU2017: we disable LTO (and thus instrumentation) for the SPEC I/O wrapper for xz, which allocates a >4GB array for one test. We also annotate one struct type in deepsjeng (via flags), which is used only for a >4GB hash table (accessed via a masked index).

For our baseline, we compile the benchmarks using an unmodified LLVM, and link against an unmodified version of tcmalloc. We compiled all benchmarks with -O2, and PIE flags. All our benchmarks were compiled using LTO (via the gold linker), and the same flags were passed to the linker.

[2]We excluded the out-of-scope ret2spec demos since they do not leak any data on our Cascade Lake machine, presumably due to hardware mitigations.

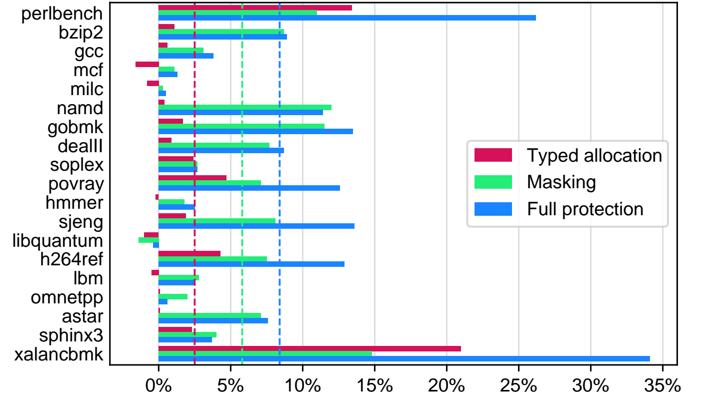

Fig. 4. CPU2006 runtime overhead

We did *not* use -fno-strict-aliasing; TBAA information improves our pointer analysis, and we did not have miscompilation issues in this version of LLVM. Otherwise, clang could be modified to output TBAA metadata despite this flag.

Figure 4 shows runtime overhead for our three basic configurations. Geometric means are 2.5% for typed allocation, 5.8% for masking, and 8.4% for the combined full TDI protection. This is significantly more efficient than prior load pointer masking-based solutions with only two colors (e.g., over 17% on CPU2006 for [33]). We can see that overhead is high (>15%) for two benchmarks, perlbench and xalancbmk, due to the cost of typed allocation; perlbench's overhead is due to the type-safe stack (2% heap, 13.4% heap+stack), while xalancbmk's overhead is due to both (11.5% heap, 21% heap+stack). Much of the perlbench overhead appears to be due to LLVM register allocator issues [66], and could be mitigated by limiting inlining.

We also evaluated runtime overhead for two alternatives (full results can be found in Figure 11 in Appendix D):

(1) TDI without stores; here, we do not instrument pointers used only by stores. Although the benefit is significant for sjeng, hmmer suffers due to different base pointers being masked on the hot path (which could be resolved with runtime profiling). The geomean on CPU2006 is 8.1%, compared to 8.4% for full protection; the cost of reduced protection would seem to outweigh this minor performance gain.

(2) TDI without dominator pointer access analysis. This analysis significantly benefits some benchmarks (e.g., namd, hmmer, sjeng and xalancbmk), and reduces the geomean overhead from 10.4% to 8.4%. We believe that improving our analysis could probably improve this overhead further; in any case, the benefit seems worth the implementation effort.

The runtime overhead of TDI on SPEC CPU2017 is shown in Figure 5; the geomean (12.5%) is higher than that of CPU2006. Overall, masking is the source of the majority of the overhead (geomean 8.0%), although omnetpp suffers from inefficiencies in our heap allocation. This is partially due to shortcomings in the analysis of our prototype; the CPU2017 versions of x264 and imagick contain significant numbers

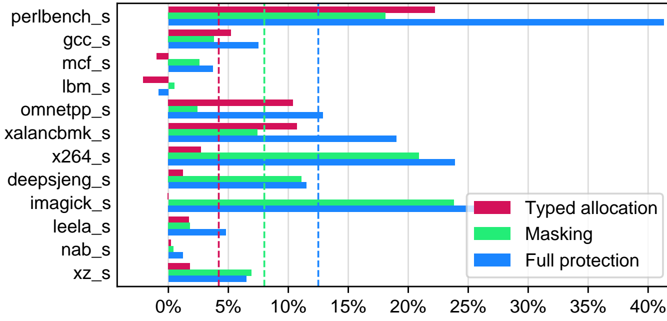

Fig. 5. CPU2017 runtime overhead

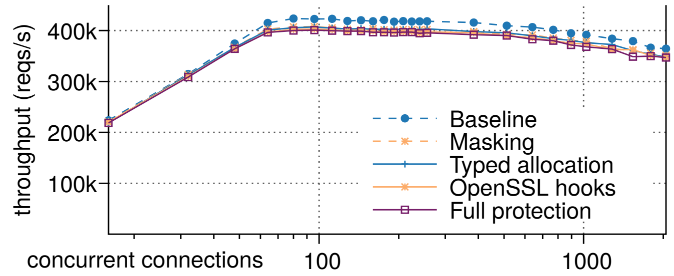

Fig. 6. nginx throughput

of (non-GEP) pointer arithmetic instructions in situations unsupported by our analysis, and are conservatively masked.

Our benchmarking of xz shows high variance, with a standard deviation of ∼5% (including the baseline). Other benchmarks (e.g., mcf and lbm) have stddev <1%; the speedups shown when using typed allocation are consistent. As discussed by Mytkowicz et al. [45], measurement bias is difficult to avoid in this form of evaluation. Our instrumentation and runtime inevitably have side-effects which will influence performance. For example, arena allocations may cause more cache conflicts; allocations at the start of arenas will share lower bits, and many arenas are only used for small allocations. This could be mitigated by adding small offsets to the arena base, e.g., based on internal type IDs or allocation order. However, we did not observe any significant performance change when subtracting small (cache-line-sized) offsets from the base pointers of the typed stacks.

Full TDI's memory overhead (peak RSS) on CPU2006 has a geomean of 15.5% (vs unmodified tcmalloc); this is due to increased memory fragmentation caused by our allocation strategy, amplified by tcmalloc configuration (e.g., minimum page cache sizes) which are inappropriate for arenas. To ensure fairness of our baseline comparison, we left these values unmodified. Details can be found in Figure 9 in the appendix.

We also compared the runtime overhead of TDI to LLVM's Speculative Load Hardening (SLH) mitigation. SLH has a significantly stronger speculative threat model which aims to prevent loads from executing by mixing predicate state (from branches) into the pointers being loaded, providing a mitigation against the majority of Spectre v1 attacks. However, overhead when applying (x86) SLH to CPU2006 is prohibitively high (geomean 75.6%), and it provides only speculative safety. (Overhead should be slightly lower without indirect call/jump hardening, but we encountered code generation errors when disabling it.) Again, detailed results are in the appendix.

D. nginx

We tested TDI using the nginx 1.18.0 web server. We used default options and enabled SSL, but disabled the 'geo' module (due to undefined behavior, see Appendix A). We linked against OpenSSL 1.1.1h[3] using LTO (and -O2), hardening both nginx and OpenSSL with TDI. We confirmed that the OpenSSL tests pass after full hardening, and used a hardened `openssl` binary to generate 2048-bit RSA keys for SSL.

[3]configured with `no-shared`, `no-asm` and `no-zlib`.

Note that nginx does not fully benefit from our automated type-based protection, since allocations in nginx's pools (including shared memory slab pools) lose the benefit of intra-pool type isolation. However, since different types of pools are identified based on callsites, pools containing disjoint types remain isolated from each other, as well as from the many other arenas identified by the type analysis (Section VII-F). One improvement could be to allocate each pool instance in a separate arena, providing finer-grained isolation.

The 'OpenSSL hooks' configuration uses TDI's instrumentation but assigns arenas using OpenSSL's allocator hooks; as we discuss later, such arenas are surprisingly coarse-grained.

We evaluated nginx by serving a small file (64 bytes) via SSL (with default settings), using two Xeon Silver 4110 machines with 100Gb/s Ethernet (plain HTTP is largely I/O bound). We configure nginx to use 16 workers, and use 16 threads of wrk2 [3] to make the requests.

Throughput results are shown in Figure 6 (median of 3 runs of 30s each); all cores are saturated for ≥96 connections. Saturated throughput at that point is 5.4% lower than the baseline for full TDI, 3.6% for masking, 3.8% for the typed allocator and 4.8% for the hook-based allocation. 90th percentile latency is 4.7% higher for full TDI, and 2.9%, 3.6% and 3.9% for masking, typed allocator and the hooks respectively.

E. Instrumenting system libraries

TDI's protection does not rely on complete instrumentation of system libraries, since pointers passed to external functions or stored in memory are always masked. For example, a call to `memcpy` will always be provided with valid pointers to the expected arenas, and any pointers copied *by* `memcpy` will already have escaped analysis, and so also have been masked.

Since glibc does not support clang, alternative C libraries have compatibility issues, and TDI's stack instrumentation currently requires LTO, we expect TDI to be used in practice with an uninstrumented system libc. Despite this, we also evaluated the overhead of applying TDI's full stack/heap protection to libc, by using musl (and libc++) rather than glibc.

Throughput overhead for our nginx+OpenSSL benchmark is 8.4% at the point of saturation (vs 5.4% without libc

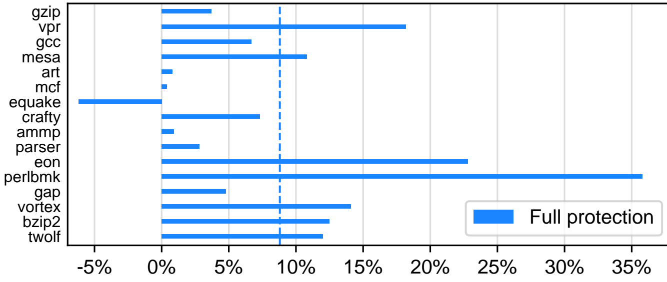

Fig. 7. CPU2000 runtime overhead

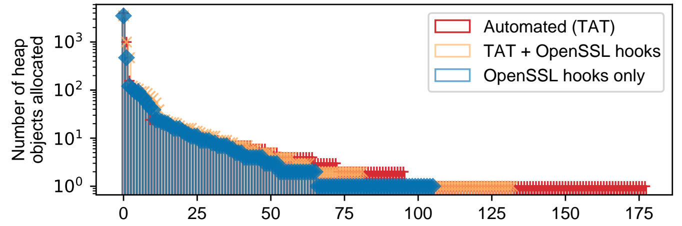

Fig. 8. Number of objects allocated in each nginx+OpenSSL heap arena.

instrumentation), and lower using alternative configurations such as using 64kB files (4%) or (multi)thread pools (6.8%). Geomean runtime overhead is 10.3% for SPEC CPU2006 (vs 8.4%); as before, xalancbmk and perlbench are largely responsible. Similarly, geomean overhead is 13.9% for CPU2017 (vs 12.5%). Details can be found in Appendix D.

We also evaluated TDI on SPEC CPU2000, to aid comparisons with prior work. Again, details of the (mostly minor) changes are in Appendix A. Figure 7 presents our performance results for full protection with complete instrumentation (including musl/libc++). As shown in the figure, the geomean runtime overhead is 8.8%—with the highest overhead (35.8%) for perlbmk, similar to previous results.

Our CPU2000 overhead is comparable to domain-based sandboxing solutions such as NaCl [72] ($\sim$ 7%)—despite our support for arbitrary (rather than NaCl-only) programs and intra-domain isolation. Moreover, our overhead is much lower than state-of-the-art software fault isolation techniques that rely on highly optimized address masking instrumentation on loads/stores [73] (rather than pointer arithmetic like TDI). Specifically, Zeng et al.'s solution [73], which can only support the limited number of colors allowed by load/store masking, yields 19% overhead on top of a CFI baseline and on a CPU2000 subset excluding costly benchmarks like perlbmk. More fine-grained solutions like WIT [7] can support more colors (limited by the imprecision of context-insensitive points-to analysis), but load instrumentation can increase overhead (10% on a CPU2000 subset excluding costly benchmarks like perlbmk) "by more than a factor of three" [7]. Note that these numbers (from [7] and [73]) are not directly comparable due to the different evaluation platforms.

F. Isolation granularity

Although TDI can be used as a traditional coarse-grained (e.g., 2-color) isolation scheme even in the absence of any automated color analysis (significantly outperforming prior load/store address masking solutions, as noted), we briefly evaluated how arenas are assigned in practice by the automated type analysis in a fine-grained, many-color configuration.

For our nginx(+OpenSSL) benchmark, the automated type analysis (TAT) statically identifies 197 colors (and arenas) on the stack and 649 colors (and arenas) on the heap. On the heap, the data type analysis assigns a total of 96 types to allocations at 583 call sites, while the remaining 553 types are identified by the context-sensitive callsite ID analysis based on wrapper detection and inlining (Section VI-C).

We also looked at the per-arena object distribution during the execution of the benchmark. Figure 8 shows the number of objects allocated in each heap arena during startup and the first client request. The 'OpenSSL hooks' results manually assign arenas by using OpenSSL's built-in support for hooking allocator functions (`CRYPTO_set_mem_functions`); we used one-line wrappers which assign an arena ID based on the callsite information provided by OpenSSL. There were 178, 133, and 106 heap arenas for the automated (TAT), TAT+hooks, and (manual) hooks configurations respectively. The two arenas with the highest number of objects are used to store OpenSSL object names and their related hashes, and all configurations have a relatively large 'long tail' of arenas used only for a single object.

Notably, this shows that attempting to manually assign arenas by hooking OpenSSL's allocator functions leads to *coarser* arenas than a fully-automated approach, even when TAT is also used to assign arenas and can merge allocations of the same type. The fully-automated approach can produce finer-grained arenas because OpenSSL's allocator hooks use indirect calls and only provide direct callsite information (filename/line numbers). For example, OpenSSL provides wrapper functions for allocating and resizing 'buffers'; OpenSSL's allocation functions are called from these buffer wrapper functions, resulting in a large number of allocations from a small number of callsites. TAT instead detects the buffer code as allocator wrappers, and instead allocates arenas based on the parent callsite since the buffer data is untyped (char *).

We also inspected arena usage for some of the SPEC benchmarks. On the CPU2000 subset evaluated by WIT [7], TDI's type analysis yields a number of colors comparable to WIT's points-to analysis (which fares well on such simple benchmarks with many stack allocations). However, unlike WIT, TDI can easily handle the entirety of CPU2000 and even much more complex programs. Moreover, while WIT is limited to 256 colors, TDI uses a larger number of colors even on the slightly more complex CPU2006 benchmarks. For example, xalancbmk allocates 186 stack and 200 heap arenas, and gcc allocates 110 stack and 192–198 heap arenas (depending on the benchmark). Appendix E contains arena statistics for the other benchmarks.

VIII. Residual attack surface

A. Spatial safety

TDI's arena allocation could be applied without masking (with much lower overhead). However, non-linear memory vulnerabilities are becoming the primary form of spatial safety vulnerability in the architectural [43] and speculative [32] domain, which may allow attackers to bypass guard zones. These are exactly the situations for which we apply masking.

As for the residual attack surface with full protection, TDI cannot prevent overflows within/across objects of the same color (i.e., type). This provides strong isolation for info leaks, although in some cases there is a remaining attack surface for intra-pool leaks. For example, OpenSSL stores data involving highly confidential data (private/session keys) in the same bignum types as data related to public keys; an info leak bug specifically revealing bignum data for a public key may also allow an attacker to obtain confidential bignum data. If desired, TDI supports annotations to further improve isolation of critical objects, much like existing data isolation solutions.

TDI also offers limited protection against memory corruption exploits which are outside our threat model. For instance, if an attacker can overwrite a pointer (e.g., in a struct), they can potentially bypass our mitigation. We make such attacks more difficult by limiting the set of pointers at reach (pointers within the same object type) and their ability to leak pointers.

B. Spectre

TDI provides the same protection against Spectre-BCB attacks as it does against non-speculative information leaks—preventing cross-arena leakage. Most other Spectre variants are clearly out-of-scope and best mitigated by techniques such as retpoline or hardware-based mitigations. However, a theoretical attack surface remains in Spectre V1 gadgets that exploit speculative issues beyond memory safety (e.g., logic bugs). We also do not prevent attacks exposing potential code/stack addresses nor secrets which are already present in registers. SLH also does not mitigate many such cases. If more comprehensive protection is required, it may be possible to use our arena-based approach to reduce the overall performance impact of a more conservative SLH-style mitigation.

C. ASLR

Our prototype allocator allots pages linearly from the base of each arena, but this is not required by our design; arenas can be placed at any 4GB aligned address and pages can be assigned non-linearly within arenas with no impact on ASLR entropy. However, our design does reduce the entropy available for *fine-grained* ASLR, since the available virtual address space is reduced (by $\sim$3 bits in our prototype), as well as the entropy for large allocations which cannot cross a 4GB boundary. If an attacker leaks a pointer of a given type, they obtain the high bits for the arena; other pointers of the same type are likely to be in the same arena. However, they obtain no information about pointers of *other* types, which are more likely to be of interest to attackers.

D. Pointer arithmetic

TDI relies on instrumenting pointer arithmetic; specifically, the *security* guarantees require that all pointer arithmetic is instrumented, while the *correctness* guarantees require that non-pointer arithmetic is not instrumented. Our prototype implementation demonstrates that balancing these needs is possible for real-world C/C++ code.

However, there are some cases where this is not possible. For example, when a union contains both a pointer and an integer value, there may not be a correct approach, if arithmetic may be relevant for both values. Similarly, code may store pointers as integers. Although we make use of sources such as TBAA, sometimes arithmetic on such values cannot be statically detected. Such code is simply incompatible with static instrumentation, but broader analysis or approaches like tagged unions [54] may help in some cases.

Other memory safety work solves these difficulties in different ways. For example, Low-fat Pointers [23] ignores 'uglygeps', excludes 23 CPU2006 functions (including gcc and perlbench) and does not instrument integer arithmetic. Although our analysis is more complete, we still had to apply some patches; we expect similar results in other software.

IX. Prototype limitations

A. Completeness

We instrument code at the LLVM IR level. Instructions could be reordered or modified during code generation in a way that compromises our mitigation. There are also inevitably unknown bugs in our prototype passes; however, we did not find any missing instrumentation when manually inspecting the output assembly code from TDI.

B. Type-based isolation

We rely on the type analysis of Type-after-Type [66] and the limitations mentioned in their paper may result in multiple types being placed in the same arena. Complementary approaches such as TypeClone [9] are an option for improving security guarantees or reducing the number of types.

Our prototype of TDI does not place global variables in type-based arenas; they are placed in data/BSS sections, in a shared arena. Address-taken global variables could be isolated by converting them to heap allocations. Custom memory allocators may also need changes to ensure TDI's type-based isolation guarantees are as fine-grained as possible.

C. Temporal safety

tcmalloc's design does not isolate size classes once memory is returned to central pools, so new allocations of types with a non-power-of-two size can overlap with previous allocations of such types. Our allocator allows memory to be returned to (typed) central pools, reducing temporal safety (but not isolation) when misalignment may occur. This could be solved by rearchitecting tcmalloc, or using a different baseline allocator.

D. Compatibility

Custom memory allocators which directly call `mmap` or `brk` must ensure that allocations do not span arena boundaries if they allocate memory regions >4GB. Custom memory allocation code can also reduce security. For example, OpenSSL's 'secure' allocation functions resize buffers by allocating new memory and `memcpy`ing the old contents, rather than using realloc; such functions *reduce* security if used with TDI.

TDI limits maximum object size due to pointer masking. our prototype limits objects to 4GB. If larger objects are required, manual annotations can be used, or masking can be disabled entirely for some functions/types. We demonstrated this for two CPU2017 benchmarks. Where a huge number of types are used in a program, we can run out of virtual address space, which can be solved by increasing the coarseness of the type classification or reducing the guard zone size. In any case, x86-64's 47-bit space already allows more than 16,000 arenas, and 5-level paging (or ARMv8.2-LVA on ARM) adds support for 56-bit userspace addresses. Note that TDI imposes no limitations on how many objects can be allocated.

X. Related work

A. Secure allocators

Similar to TDI, secure allocators change the stack/heap allocation strategy to improve security. A common approach is to provide probablistic security by randomizing the positions at and/or order in which allocations are made, as done by StackArmor [18] (stack), DieHard [10] (heap), and OpenBSD's allocator [44]. DieHarder [47] adds guard pages to DieHard (like e.g., Electric Fence); however, allocations remain distinguished by sizes, not types. Archipelego [39] allocates one object per page, allowing guard zones between objects. FreeGuard [61] provides probabilistic security using a combination of randomization, delayed reuse, and guard zones.

Other efforts focus on temporal memory errors such as use-after-free. MarkUs [5] delays freeing memory until pointers no longer appear in memory, while FFmalloc [71] uses one-time allocations (with no memory reuse at all).

Cling [6] mitigates heap memory reuse exploits using independent allocator regions for each allocation site. Type-after-Type [66] extends Cling's design with compile-time type detection, improved wrapper detection, and stack support. Automatic Pool Allocation [37] relies on points-to compiler analysis to split allocations into separate typed pools, which allows for temporal protection of a subset of C [21]. All these defenses use some kind of "typed" pools to enable type-safe memory reuse, but do not provide spatial data isolation.

Modern web browsers also use manual allocation-level isolation to improve security, such as IE's Isolated Heap. In particular, PartitionAlloc [2] is Chrome's default allocator (as of March 2021). It allows (manual) allocation in isolated arenas ('partitions'), mitigating some temporal and linear overflow vulnerabilities, and could be used as an alternative allocator for TDI. V8 also sandboxes WebAssembly by limiting memory offsets (to the sum of two 32-bit offsets) and allocating guard zones for ±8GB around the heap [4], with accesses always using a valid base address.

B. Data isolation

Address-based defenses [33] typically use annotations of sensitive types or data to isolate one or more specific regions of memory. *Domain-based* defenses [33] instead protect sensitive *code*, protecting the data used by that code, and only allowing access when execution has switched to the relevant domain. Existing solutions fall in either one or both classes of defenses and implement different isolation mechanisms.

DataShield [15] uses annotations and masking via instrumentation. Data-flow analysis identifies potential sensitive data accesses, needing slower metadata checks, and protects non-memory flows. Non-sensitive data is placed in memory <4GB and pointers are truncated to 32 bits. Overhead in artifical case studies, annotating a single type as sensitive, is 9.12% and 27.21% for two CPU2006 benchmarks, and lower ($\sim 0\%$ using x86 prefixes) when code provably cannot access sensitive data.

ConfLLVM [12] also uses annotations along with segmentation or Intel MPX. CPU2006 overhead is 24.5% without any private data, although this includes CFI, and excludes perlbench and xalancbmk (highest overhead in our evaluation).

MemSentry [33] evaluates a variety of these solutions, implementing domain-based (virtualization and MPK) and address-based (encryption, masking, and MPX) defenses. The authors report 17.1% overhead for load masking on CPU2006.

Palit et al. [51] encrypt sensitive data using annotations and points-to analysis. The overhead is 4-33% when protecting only keys. MemCat [46] attempts to distinguish *attacker-controlled* data using compile-time policy and allocates those objects on a separate heap/stack. CPU2006 overhead is 21%.

ERIM [65] uses MPK (also Spectre-BCB-safe) to isolate memory used by a trusted domain; they demonstrate low-overhead protection of CPI's [36] safe region. SeCage [38] uses EPT (page table switching), automatically splitting off code to protect annotated secrets, and xMP [53] uses a similar approach to manually protect kernel data structures or cryptographic data in user-space code.

Data Flow Integrity [17] (DFI) uses points-to analysis to determine which stores should be accessible to each load, enforcing fine-grained isolation at runtime with costly instrumentation that checks/updates a metadata table on loads/stores. Write Integrity Testing [7] (WIT) reduces DFI's overhead by only protecting stores and limiting object colors to at most 256, adding guards between objects to compensate for imprecise points-to analysis; overhead is 10% on a CPU2000 subset (without eon or perlbmk, TDI's worst cases). Other variants of schemes relying on points-to analysis also exist [62].

Hardware memory tagging (e.g., MTE [26]) provides an alternative isolation primitive; it could be used as an alternative to arenas, with TDI used to protect tags from info leaks.

Finally, TDI draws inspiration from optimization strategies used by prior SFI [69] and other solutions. For instance, Zeng et al. [73] use simpler forms of range and dominating pointer analysis on x86 assembly to eliminate SFI instrumentation

on loads/stores. In contrast, TDI reasons over pointer *arithmetic* at the compiler IR level, allowing simpler but more effective static analysis to aggressively remove instrumentation. Previous work also used guard areas to eliminate SFI instrumentation; in particular, on loads/stores with a fixed base pointer [57], a fixed pointer offset [41], or to only detect linear buffer overflows [11], [61]. In contrast, TDI uses guard pages to reason about whether computed pointers are *safe* with respect to arbitrary "valid" base pointers, rather than directly reasoning about load/store accesses. TDI also considers speculative flows, which limit (or prohibit) the applicability of much of this previous optimization work. Finally, TDI's instrumentation relies on efficient address masking similarly to some SFI solutions [35] (others resort to bounds checking [24]), but uses masking to *preserve* bits after pointer arithmetic, rather than using a bitmask to *remove* bits at loads/stores. This allows TDI to support fine-grained isolation, rather than only the coarse-grained (e.g., 2-color) isolation of traditional SFI solutions.

C. Bounds-checking defenses

Some bounds checking defenses have similarities to our work. Baggy Bounds Checking [8] instruments arithmetic using tagged pointers (on 64-bit), with ∼60-70% overhead. Low-Fat Pointers [23] simplifies this by encoding bounds into *valid* pointers, instrumenting arithmetic and accesses, with 113% overhead. Delta Pointers [34] also instruments accesses and arithmetic (documenting challenges similar to TDI's); by encoding the *delta* to object ends in pointers, the authors limit detection to overflows and total memory space to 4GB, with 35% overhead. Similar in spirit to TDI's guard zones, Delta Pointers offloads checks to the MMU to improve performance.

Dhurjati et al. [19] use points-to analysis (via [37]) to optimize bounds checks. By omitting checks when points-to analysis fails, this avoids compatibility problems (unlike similar work such as [20]) at the cost of security, and achieves average overhead of ∼12% on the (simple) Olden benchmarks.

AddressSanitizer [58] is a compiler-based debugging tool, using instrumentation, shadow memory and delayed reuse, but recent overhead is still ∼80% on CPU2006. Newer sanitizers such as CUP [13] and EffectiveSan [22] detect broader ranges of threats, with significantly higher overhead.

D. Spectre mitigations

Canella et al. [14] describe three categories of Spectre mitigations: mitigating covert channels (e.g., reducing timer accuracy or hardware changes), aborting speculation (e.g., fences or retpoline), and making secret data unreachable.

Compilers can automatically insert fences after vulnerable branches to stop speculation [29], but attempts to implement this efficiently for Spectre-BCB (e.g., in MSVC) have been shown to be error-prone [31]. Blade [68] proposes fencing/masking only paths where data may speculatively leak, which the authors apply to WebAssembly. Operating systems such as Linux and other solutions [49], [70] use similar selective (and thus noncomprehensive) fencing policies based on manual annotations or results of program (i.e., gadget) analysis. State-of-the-art comprehensive solutions such as LLVM's Speculative Load Hardening [16] (SLH) mitigation offer a complete but costly alternative. SLH forces a data dependency on the control flow leading to potentially-vulnerable loads, by mixing bits of the predicates used by the control flow into the pointers used by such loads. In contrast to such mitigations, TDI provides a gadget-agnostic defense with strong and fine-grained data isolation guarantees at low overheads.

Web browsers apply similar mitigations such as masking array indexes [42], [52] and applying SLH-type poisoning [42], [64]. Such mitigations are typically easier to comprehensively deploy within JIT environments, but coarser-scale solutions such as Site Isolation are still considered more cost effective [55]. Moreover, some of the efficient masking solutions used by modern browsers such as Firefox use coarse-grained masks which still allow (limited) out-of-bounds accesses to objects of a different type [28], in contrast to TDI.

Ghostbusting [30] proposes mitigating Spectre-BCB vulnerabilities with data isolation via domain switching, and ConTExT [56] protects annotated sensitive data using hardware extensions or uncacheable ('non-transient') memory mappings. The former has been only evaluated with synthetic programs, the latter reports 71% for OpenSSL RSA vs our ∼16%, although our threat models differ significantly.

Other efforts focus on detecting Spectre-BCB and similar vulnerabilities. oo7 [70] finds potential Spectre vulnerabilities using BAP to propagate taint from untrusted sources. Spectector [27] instead applies symbolic execution to source code. SpecFuzz [49] focuses on fuzzing software to find Spectre-BCB gadgets and seeks to reduce the overhead of SLH (but also its security guarantees) by excluding branches that do not appear vulnerable. TDI's overhead for OpenSSL's ECDSA benchmark is ∼7%, vs SLH's ∼70%; SpecFuzz improves the latter by only 5%, although the performance difference is less extreme for other cases. We could attempt to use SpecFuzz to reduce our masking, but this would remove non-speculative protection and potentially increase our speculative attack surface due to false negatives.

XI. Conclusion

We have shown that we can efficiently harden programs against temporal and spatial (even speculative, a la Spectre-BCB) info leak vulnerabilities, by using arenas to provide N-color isolation. We have also demonstrated that our protection can applied automatically by exploiting fine-grained type information for object coloring.

Our type-based arena allocation on the heap and stack has typical run-time overhead far below 5% and already provides a strong mitigation against classical temporal and linear (adjacent) spatial attacks. We significantly broaden this protection by masking pointers to keep them in their intended arena, mitigating non-adjacent and speculative vulnerabilities. TDI still achieves acceptable run-time overhead by minimizing the need to mask pointers. We believe this overhead could be further improved with assistance from compiler frameworks.

Acknowledgements

We thank the anonymous reviewers, Koen Koning, and Taddeus Kroes for their valuable feedback. This work was supported by Intel Corporation through the Side Channel Vulnerability ISRA, by the Netherlands Organisation for Scientific Research through projects "TROPICS" and "Theseus", by EKZ through project "VeriPatch", by Cisco Systems, Inc. through grant #1138109, and by the Office of Naval Research (ONR) under awards N00014-16-1-2261 and N00014-17-1-2788. This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

References

[1] "Google SafeSide," https://github.com/google/safeside, September 2020.
[2] "Partitionalloc," https://chromium.googlesource.com/chromium/src/+/master/base/allocator/partition_allocator/PartitionAlloc.md.
[3] "wrk2," https://github.com/giltene/wrk2, September 2019.
[4] "WebAssembly Out of Bounds Trap Handling," 2016.
[5] S. Ainsworth and T. M. Jones, "MarkUs: Drop-in use-after-free prevention for low-level languages," in *S&P '20*.
[6] P. Akritidis, "Cling: A memory allocator to mitigate dangling pointers," in *USENIX Security '10*.
[7] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with wit," in *S&P '08*.
[8] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors." in *USENIX Security '09*.
[9] M. Barbar, Y. Sui, and S. Chen, "Flow-sensitive type-based heap cloning," in *ECOOP '20*.
[10] E. Berger and B. Zorn, "DieHard: probabilistic memory safety for unsafe languages," in *PLDI '06*.
[11] S. Bhatkar and R. Sekar, "Data space randomization," in *DIMVA '08*.
[12] A. Brahmakshatriya, P. Kedia, D. P. McKee, D. Garg, A. Lal, A. Rastogi, H. Nemati, A. Panda, and P. Bhatu, "ConfLLVM: A compiler for enforcing data confidentiality in low-level code," in *EuroSys '19*.
[13] N. Burow, D. McKee, S. A. Carr, and M. Payer, "CUP: Comprehensive user-space protection for C/C++," in *AsiaCCS '18*.
[14] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *USENIX Security '19*.
[15] S. A. Carr and M. Payer, "Datashield: Configurable data confidentiality and integrity," in *AsiaCCS '17*.
[16] C. Carruth, "Speculative load hardening," July 2018.
[17] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *OSDI '06*.
[18] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, "Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries." in *NDSS '15*.
[19] D. Dhurjati and V. Adve, "Backwards-compatible array bounds checking for c with very low overhead," in *Proceedings of the 28th international conference on Software engineering*.
[20] D. Dhurjati, S. Kowshik, and V. Adve, "SAFECode: enforcing alias analysis for weakly typed languages," in *PLDI '06*.
[21] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner, "Memory safety without garbage collection for embedded applications," in *TECS '05*.
[22] G. J. Duck and R. H. Yap, "EffectiveSan: type and memory error detection using dynamically typed C/C++," in *PLDI '18*.
[23] ——, "Heap bounds protection with low fat pointers," in *CC '16*.
[24] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "Xfi: Software guards for system address spaces," in *OSDI '06*.
[25] S. Ghemawat and P. Menage, "TCMalloc: Thread-caching malloc," 2009.
[26] M. Gretton-Dann, "Arm a-profile architecture developments 2018: Armv8.5-a," 2018.
[27] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows," in *S&P '20*.
[28] N. Hadad and J. Afek, "Overcoming (some) spectre browser mitigations," https://alephsecurity.com/2018/06/26/spectre-browser-query-cache/, 2018.
[29] Intel, "Speculative execution side channel mitigations," July 2018, revision 3.0.
[30] I. R. Jenkins, P. Anantharaman, R. Shapiro, J. P. Brady, S. Bratus, and S. W. Smith, "Ghostbusting: Mitigating Spectre with intraprocess memory isolation," in *HoTSoS '20*.
[31] P. Kocher, "Spectre Mitigations in Microsoft's C/C++ Compiler," 2018.
[32] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *S&P '19*.
[33] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No need to hide: Protecting safe regions on commodity hardware," in *EuroSys '17*.
[34] T. Kroes, K. Koning, E. van der Kouwe, H. Bos, and C. Giuffrida, "Delta pointers: Buffer overflow checks without the checks," in *EuroSys '18*.
[35] J. A. Kroll, G. Stewart, and A. W. Appel, "Portable software fault isolation," in *2014 IEEE 27th Computer Security Foundations Symposium*.
[36] V. Kuznetzov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *OSDI '14*.
[37] C. Lattner and V. Adve, "Automatic pool allocation: Improving performance by controlling data structure layout in the heap," in *PLDI '05*.
[38] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation," in *CCS '15*.
[39] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn, "Archipelago: trading address space for reliability and security," in *ASPLOS '08*.
[40] G. Maisuradze and C. Rossow, "Speculose: Analyzing the security implications of speculative execution in cpus," *arXiv preprint arXiv:1801.04084*, 2018.
[41] S. McCamant and G. Morrisett, "Evaluating sfi for a cisc architecture," in *USENIX Security '06*.
[42] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *arXiv preprint arXiv:1902.05178*, 2019.
[43] M. Miller, "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape," in *BlueHat IL '19*.
[44] O. Moerbeek, "A new malloc (3) for openbsd," in *EuroBSDCon 2009*.
[45] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *ASPLOS '09*.
[46] M. Neugschwandtner, A. Sorniotti, and A. Kurmus, "Memory categorization: Separating attacker-controlled data," in *DIMVA '19*.
[47] G. Novark and E. D. Berger, "DieHarder: securing the heap," in *CCS '10*.
[48] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2018.
[49] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer, "Specfuzz: Bringing spectre-type vulnerabilities to the surface," in *USENIX Security '20*.
[50] OpenSSL, "TLS heartbeat read overrun (CVE-2014-0160)."
[51] T. Palit, F. Monrose, and M. Polychronakis, "Mitigating data leakage by protecting memory-resident sensitive data," in *ACSAC '19*.
[52] F. Pizlo, "What Spectre and Meltdown Mean For WebKit," Jan 2018.
[53] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, "xmp: Selective memory protection for kernel and user space," in *S&P '20*.
[54] J. Rafkind, A. Wick, J. Regehr, and M. Flatt, "Precise garbage collection for C," in *ISMM '09*.
[55] C. Reis, A. Moshchuk, and N. Oskov, "Site isolation: Process separation for web sites within the browser," in *USENIX '19*.
[56] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss, "ConTExT: A generic approach for mitigating Spectre," in *NDSS '20*.
[57] D. Sehr, R. Muth, C. L. Biffle, V. Khimenko, E. Pasko, B. Yee, K. Schimpf, and B. Chen, "Adapting software fault isolation to contemporary cpu architectures," 2010.
[58] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, 2012, pp. 309–318.
[59] F. J. Serna, "The info leak era on software exploitation," *Black Hat USA*, 2012.

[60] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *CCS '07*.
[61] S. Silvestro, H. Liu, C. Crosser, Z. Lin, and T. Liu, "Freeguard: A faster secure heap allocator," in *CCS '17*.
[62] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity." in *NDSS '16*.
[63] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *CC '16*.
[64] B. L. Titzer and J. Sevcik, "A year with Spectre: a V8 perspective," Apr 2019.
[65] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, efficient in-process isolation with protection keys (MPK)," in *USENIX Security '19*.
[66] E. Van Der Kouwe, T. Kroes, C. Ouwehand, H. Bos, and C. Giuffrida, "Type-after-type: Practical and complete type-safe memory reuse," in *ACSAC '18*.
[67] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *S&P '19*.
[68] M. Vassena, K. V. Gleissenthall, R. G. Kici, D. Stefan, and R. Jhala, "Automatically eliminating speculative leaks from cryptographic code with blade," in *POPL '21*.
[69] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *SOSP*, 1993.
[70] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, "oo7: Low-overhead defense against spectre attacks via program analysis," *IEEE Transactions on Software Engineering*, 2019.
[71] B. Wickman, H. Hu, I. Y. D. Jang, J. L. S. Kashyap, and T. Kim, "Preventing use-after-free attacks with fast forward allocation," in *USENIX Security '21*.
[72] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A sandbox for portable, untrusted x86 native code," in *S&P '09*.
[73] B. Zeng, G. Tan, and G. Morrisett, "Combining control-flow integrity and static analysis for efficient and validated data sandboxing," in *CCS '11*.
[74] T. Zhang, D. Lee, and C. Jung, "Bogo: buy spatial memory safety, get temporal memory safety (almost) free," in *ASPLOS '19*.

APPENDIX A
UNDEFINED POINTER ARITHMETIC IN SOFTWARE

Although TDI is compatible with a range of software, as discussed in Section IX, it is still incompatible with software which performs undefined pointer arithmetic and uses the results across function boundaries. During testing, we found such pointer arithmetic issues in several pieces of software. We document these issues here to assist future researchers.

We discovered that CPU2006's version of gcc stores out-of-range pointers (pointing to *before* the start of the allocated object) in global variables, which are later dereferenced by other functions. Although TDI can handle pointers being slightly out-of-bounds (see Section VI), the negative delta in these cases is non-constant and can be quite large, resulting in pointers being wrapped. We investigated and found two pre-2006 gcc patches which remove these cases, in both cases since they are undefined behavior; they are r62672 from 2003-02-11, "Don't use offset pointers.", and r89543 from 2004-10-25, "avoid undefined pointer arithmetic on qty_table".

The obstack code in CPU2017's version of gcc stores cross-object pointer deltas in a temporary variable inside a struct allocated on the heap, which is undefined behavior. In fact, the default behavior of the obstack code – including the version in CPU2006's gcc – is to avoid this by using a non-standard C extension, where supported by the compiler. However, the code in CPU2017 was modified by SPEC to disable the use of this code path. We re-enable it by removing the newly-added `!defined(SPEC)` from obstack.h. Note that there is other undefined behavior present in CPU2017 (such as arithmetic using NULL pointers) which we resolve in our canonicalization pass but present issues for upstream LLVM[4].

```
u_char                *p;
ngx_http_geo_range_t **ranges;

ranges[i] = (ngx_http_geo_range_t *)
            (p - (u_char *) fm.addr);
```

Listing 7: Simplified code example from nginx 1.18.0's `ngx_http_geo_create_binary_base` function.

CPU2016's soplex uses a pointer delta to adjust pointers after a call to realloc. We did not encounter this in our tested configuration (since these objects are allocated in the same arena), and patching it appears to be non-trivial and invasive. This issue has also been observed by other researchers [48].

Both the CPU2006 and CPU2017 versions of perlbench make use of cross-object delta calculation in the mergesort code. Our pointer analysis correctly handles this for CPU2006, but in CPU2017 we were forced to exclude the mergesort function. TDI warns about these issues at compile time, along with various other unexpected patterns in the code, such as casting a 0x55555555 constant to a pointer.

We only encountered issues with a single CPU2000 benchmark, 254.gap, due to a custom allocator/garbage collector ('Gasman') which would need to be modified to support arena-based allocation. We excluded (when instrumenting) the `TypHandle` struct (used by GAP's 'Gasman' allocator/-garbage collector,) as well as several functions.

nginx provides an illustrative example of arithmetic which must be excluded, annotated or modified to be successfully compiled with TDI. Our pass prints an error when trying to instrument the http_geo module of nginx, due to the code in Listing 7, which subtracts a pointer from a pointer and stores the result in an array of pointers (*fm* is a file mapping object; we believe this code is trying to update the base address of an array of pointers, and ranges should be a *ptrdiff_t**). Since the base pointer cannot be determined, masking cannot be applied, and compilation fails.

APPENDIX B
EVALUATION BUILD DETAILS

We applied patches to musl and libc++ to fix LTO issues (such as removing weak symbols) and patched several benchmarks to fix build issues (such as missing includes). We do not instrument code which performs cross-arena arithmetic (e.g., ELF header parsing, vDSO support and stack unwinding); for similar reasons, we do not instrument our allocator itself.

We needed to pass various compatibility flags and make some minor source changes (e.g., including header files) to make the SPEC benchmarks build in our environment. A

[4]https://lists.llvm.org/pipermail/llvm-dev/2017-July/115064.html

TABLE I
MODIFICATIONS APPLIED TO SPEC BENCHMARK CODE.

Software	Reason	Solution
CPU2000 gap	Use of legacy termio	Replaced with termios
CPU2000 gap	Custom garbage collector	Excluded type
CPU2006 gcc	Undefined behavior (negative offsets)	Applied (pre-2005) upstream patches
CPU2017 gcc	Undefined behavior (cross-arena deltas)	`!defined(SPEC)` removed from obstack.h
CPU2017 perlbench	Undefined behavior (cross-arena deltas)	Excluded `S_mergesortsv`
CPU2006 dealII	Missing #include	Added #include
CPU2017 xz	>4GB allocation in SPEC wrapper	Disable LTO for `spec_mem_io.c`

summary of the (non-trivial) source changes can be seen in Table I (together with the fixes for the issues described above).

APPENDIX C
POINTER DETECTION

While implementing TDI, we found that in real-world programs, neither the types in the LLVM IR nor even the types in the source code accurately reflect whether a value is used as a pointer or not. As such, TDI requires pointer detection to ensure all pointer dereferences are properly masked, and no integers are corrupted by pointer masking. We describe our design at a high level in Section IV-B, and include the details here for transparency and reproducibility.

Our pointer detection classifies each value in the LLVM IR as one of four groups: pointers, offsets, negated pointers, and non-pointers. Additionally, during the analysis a value can be classified as unknown or invalid. Initially, we consider every value to be in the unknown class. Our analysis proceeds in four steps that mark unknown-class values based on their usage, starting with the usages that provide most confidence about (non)pointer status. Each step is followed by forward and backward propagation, marking those values that are used to compute the newly marked values and those that are computed from those values.

1) Marking: We perform marking in four steps: (1) We first mark variables dereferenced in loads and stores (which must therefore be pointers). (2) We mark function arguments or return values based on types from the relevant function prototypes. (3) We then mark values which are loaded/stored based on the type of the pointer used. (4) Finally, we mark any remaining unknown values based on their (LLVM IR) type.

After *each* of these marking steps, we propagate pointer types both backwards and forwards. If we find that a value is used as both a pointer and a non-pointer type, we mark it as having a pointer type.

2) Propagation: We propagate pointer types through arithmetic. We consider pointers which are used in shifts, divisions and multiplications to be transformed beyond use, as with AND operations discarding the high bits of a pointer, and mark them as non-pointers. We perform some further analysis on some specific arithmetic operations (and GEP instructions, in LLVM IR). Pointers which are added (or otherwise combined, e.g., ORed) to a non-pointer remain pointers.

If a pointer is subtracted from another pointer, it becomes a non-pointer; if a pointer is subtracted from a non-pointer, it becomes a negative pointer. If a negative pointer is added to a pointer, then we can mark the result as a non-pointer. We treat similar patterns in the same way; for example, we mark a pointer XORed with a negative constant as a negative pointer. We found this to be essential for analyzing both real-world C code and the output of some LLVM transformations.

3) Base pointers: If an arithmetic operation is determined to result in a pointer type, then we add it to a list of potential pointer arithmetic operations, to be considered by the remaining stages of our analysis. We then attempt to determine the base pointer for each instance of such arithmetic. If only one of the operands for an arithmetic operation is a pointer, then we take that operand as the base pointer. If neither or both operands are pointers, then we mark the base pointer as *invalid*. We found this last case to occur in code calculating pointer hashes, or in control flow paths with unused results.

However, we do not remove these invalid instances of arithmetic from the list of potential pointer arithmetic operations; we later output a warning if we conclude that such arithmetic should be masked. Although such situations are rare, they do occur; an example can be found in Appendix A. During evaluation, we erred on the side of caution, producing an error rather than a warning, and manually annotated 4 cases which could encounter this error in some build configurations.

APPENDIX D
ADDITIONAL RESULTS

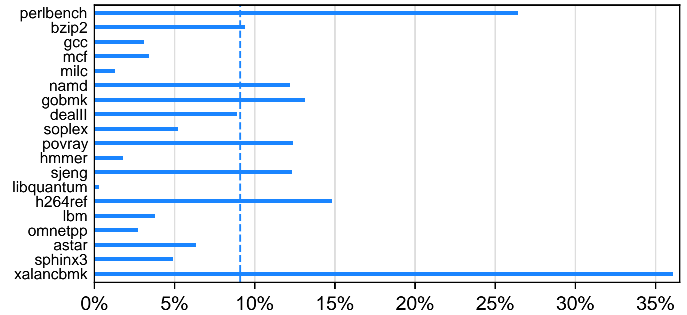

Fig. 9. CPU2006 (peak) memory overhead

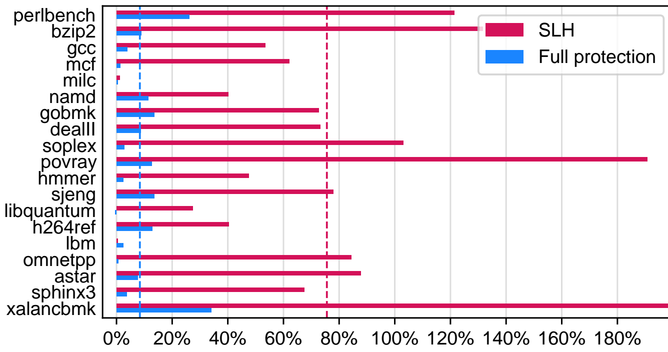

Fig. 10. CPU2006 runtime overhead vs SLH (xalancbmk is 405%)

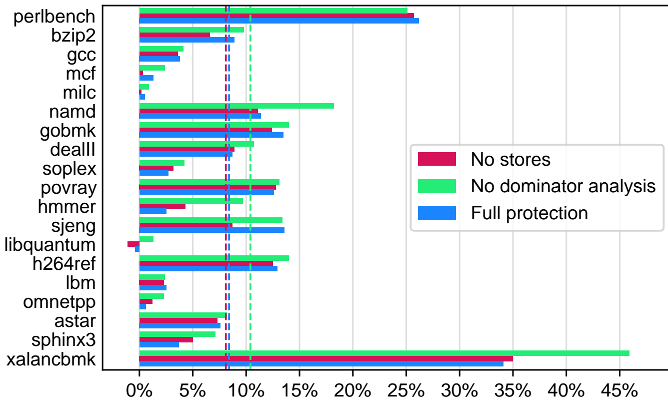

Fig. 11. CPU2006 runtime overhead with alternative configurations

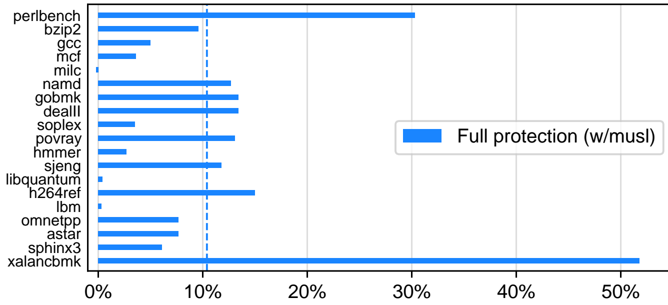

Fig. 12. CPU2006 runtime overhead with instrumented musl/libc++

Appendix E
Arena Statistics

The number of arenas actually allocated at runtime is the sum of the stack and heap arenas column in Table II.

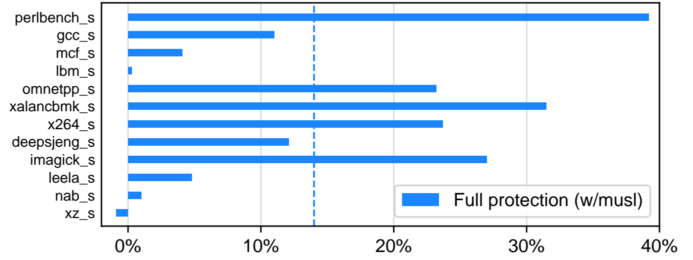

Fig. 13. CPU2017 runtime overhead with instrumented musl/libc++

TABLE II
Arena allocation statistics for SPEC CPU2000 and CPU2006 benchmarks (incl. musl/libc++).

Benchmark	Stack arenas	Heap arenas[1]	Heap type IDs[2]	Heap call-site IDs[2]
164.gzip	36	9	9	22
175.vpr	45	39–70	33	66
176.gcc	66	21	21	251
177.mesa	38	23	47	31
179.art	33	10	13	19
181.mcf	34	7	10	18
183.equake	34	9	12	18
186.crafty	35	8	10	20
188.ammp	37	15	23	18
197.parser	36	6	8	19
252.eon	68	42	30	81
253.perlbmk	52	60	22	148
254.gap	38	6	8	21
255.vortex	53	9	8	24
256.bzip2	36	11	10	22
300.twolf	34	87	37	94
400.perlbench	59	69–80	28	222
401.bzip2	35	10	9	23
403.gcc	110	192–198	98	414
429.mcf	34	7	10	18
433.milc	39	15	17	29
444.namd	40	17	12	35
445.gobmk	47	15	17	20
447.dealII	116	135	66	252
450.soplex	66	90–95	25	202
453.povray	66	98	79	147
456.hmmer	40	34–50	30	139
458.sjeng	36	8	12	18
462.libquantum	36	8	12	19
464.h264ref	37	50–53	43	37
470.lbm	33	7	8	20
471.omnetpp	55	88	61	1200
473.astar	37	33	15	40
482.sphinx3	40	98	47	95
483.xalancbmk	186	200	212	1764

[1] The number of heap arena IDs used (at least one object allocated) at runtime; this is specified as a range where some subbenchmarks allocate objects in fewer arenas.

[2] The number of heap types and/or heap callsites (i.e., untyped) assigned unique arena IDs at compile time. The total number of potential heap arena IDs is the sum of these two columns.