BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free

Tong Zhang
ztong@vt.edu
Virginia Tech

Dongyoon Lee
dongyoon@cs.vt.edu
Virginia Tech

Changhee Jung
chjung@cs.vt.edu
Virginia Tech

Abstract

A memory safety violation occurs when a program has an out-of-bound (spatial safety) or use-after-free (temporal safety) memory access. Given its importance as a security vulnerability, recent Intel processors support hardware-accelerated bound checks, called Memory Protection Extensions (MPX). Unfortunately, MPX provides no temporal safety.

This paper presents *BOGO*, a lightweight full memory safety enforcement scheme that transparently guarantees temporal safety on top of MPX's spatial safety. Instead of tracking separate metadata for temporal safety, *BOGO* reuses the bounds metadata maintained by MPX for both spatial and temporal safety. On `free`, *BOGO* scans the MPX bound tables to invalidate the bound of dangling pointers; any following use-after-free error can be detected by MPX as an out-of-bound error. Since scanning the entire MPX bound tables could be expensive, *BOGO* tracks a small set of hot MPX bound table pages to check on `free`, and relies on the page fault mechanism to detect any potentially missing dangling pointer, ensuring sound temporal safety protection.

Our evaluation shows that *BOGO* provides full memory safety at 60% runtime overhead and 36% memory overhead for SPEC CPU 2006 benchmarks. We also show that *BOGO* incurs a reasonable 2.7x slowdown for the worst-case malloc-free intensive benchmarks; and moderate 1.34x overhead for real-world applications.

CCS Concepts • **Security and privacy → Software security engineering**; • **Software and its engineering → Compilers**; • **Hardware →** *Emerging architectures*; *Emerging languages and compilers*; *Emerging tools and methodologies.*

Keywords Memory Safety; MPX

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304017

ACM Reference Format:
Tong Zhang, Dongyoon Lee, and Changhee Jung. 2019. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3297858.3304017

1 Introduction

Memory unsafe languages such as C/C++ are prone to bugs leading to memory safety violations [57]. Spatial safety violation occurs when memory access is not within the object's bound (e.g., buffer overflow), while temporal safety violation[1] happens when accessing a deallocated object (e.g., use-after-free).

Memory safety violations are a common source of real-world security breaches [1]. While buffer overflow vulnerabilities have been exploited for return oriented programming [48] and other code reuse attacks [11, 14, 58], use-after-free vulnerabilities have also been exploited to corrupt control flow: e.g., virtual function table hijacking [51].

Keeping pace with a broad range of research (§2), hardware support for security has been adopted in mainstream commodity processors – notably, Intel's Memory Protection Extensions (MPX) for hardware-accelerated bounds checking. MPX keeps track of per-pointer bound metadata (the base and bound of heap/stack objects) in bound tables. On a pointer dereference, MPX checks if the pointer value remains within the bounds, ensuring spatial memory safety. Unfortunately, MPX does not support temporal memory safety. Thus, full memory safety can only be achieved by augmenting MPX with a separate temporal safety solution. However, existing solutions for temporal memory safety (§2.3) require their metadata tracking and checking, doubling the time and space overheads when combined with MPX.

This paper presents *BOGO*[2], a lightweight full memory safety enforcement scheme that works with MPX (Intel Skylake onwards). This work demonstrates a novel software solution that transparently extends MPX to support both spatial and temporal memory safety, without additional hardware support or significant performance degradation: Buy spatial

[1] We use "temporal memory safety" and "no use-after-free vulnerabilities" interchangeably, though the former subsumes the latter. The same is true for "spatial memory safety" and "no out-of-bound vulnerabilities".
[2] The name is inspired from an initialism for Buy One, Get One free.

memory safety, and get temporal memory safety (almost) free!

The key insight to realize the "promotion" is that the MPX bound table can be searched for dangling pointers to an object when it is freed. Since the bound table entry already maintains the bound information for each pointer, dangling pointers can be identified by checking for bounds enclosing the address being freed. For each dangling pointer `p` found, *BOGO* invalidates the bound information of the bound table entry, indexed by that pointer `p`. On the later dereference of `p` (use-after-free), MPX instrumentation will find an invalid bound (as a part of its bound checking), and raise an exception. In effect, *BOGO* achieves temporal safety by transforming it into spatial safety.

This approach relieves *BOGO* of the burden to maintain and check a separate temporal memory safety metadata, reducing time overhead and more drastically space overhead. *BOGO* introduces a new synergistic way to enforce spatial and temporal memory safety by repurposing one for another. However, scanning the entire MPX bound tables on each `free` could lead to significant performance overhead. *BOGO* leverages a novel page-protection-based technique to address this performance challenge. *BOGO* tracks the working set of MPX bound table pages and only searches those *hot* pages on `free` for performance. To track a dangling pointer potentially in the rest *cold* pages, *BOGO* makes the cold MPX bound table pages non-accessible. Any following access to a cold page is always preceded by a page fault. *BOGO*'s page fault handler scans the faulted MPX page and invalidates any dangling pointers therein, guaranteeing soundness.

This paper makes the following contributions:

- To the best of our knowledge, *BOGO* is the first temporal memory safety protection solution that does not maintain its own metadata, but seamlessly reuses bound metadata tracked for spatial memory safety.
- *BOGO* transparently provides an MPX-enabled binary with full memory safety without application change or other hardware support.
- We implement `llvm-mpx`, an LLVM-based MPX pass, with sound bound checking optimizations, outperforming existing MPX compilers.
- The experimental results show that *BOGO* can support full memory safety at comparable (in many cases, better) runtime overhead and much less memory overhead, compared to the state-of-the-art solutions.
- We stress-test *BOGO* with the worst-case malloc-free intensive benchmarks, and also evaluate *BOGO*'s interoperability and scalability for real-world multithreaded applications.

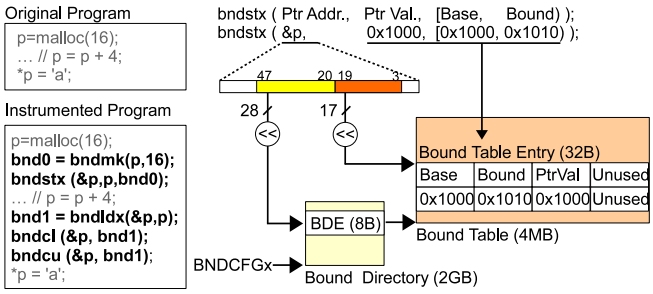

Figure 1. MPX stores the base and bound metadata (`0x1000-0x1010`) of the pointer `p` (with an initial value of `0x1000`) in the two-level Bound Tables, and checks if the later use of pointer `p` (with a new value, say `0x1004`, after some updates) is within the base and the bound.

2 Background and Related Work

2.1 Spatial Memory Safety

Spatial memory safety violations (*bounds errors*) arise when memory access is not within the object's bound. Some [17, 19, 45, 53, 63, 70] use guard blocks or canaries at the beginning and end of memory objects to detect out of bound memory accesses. Others [9, 20, 22, 35, 50] allocate metadata for each memory object and perform explicit checking on pointer manipulation. Still, other schemes [10, 29, 44, 66] maintain metadata for each pointer and check at the time of dereferencing. Metadata may be stored in unused most-significant bits of 64-bit pointer [32]. Custom hardware support [21, 39, 49, 55, 61, 65] has also been proposed. SoftBound [41] is of particular interest, as its per-pointer *base* and *bound* metadata tracking significantly influences the design of Intel MPX.

2.2 Memory Protection Extension.

MPX provides ISA support for accelerating pointer-based bounds checking. It gives four 128-bit bound registers (`bnd0-3`) and instructions: `bndmk` to create base and bound metadata; `bndldx` and `bndstx` to load/store metadata from/to the disjoint metadata, called bound tables; `bndcl` and `bndcu` to check pointer with lower and upper bounds; and more.

Figure 1(left) shows how a program is instrumented for bounds checking. The base and bound metadata is initialized on pointer creation, and propagated on pointer manipulation. When a pointer is dereferenced, MPX checks if the pointer is within the corresponding range, and raises an exception upon an out-of-bound access. Figure 1(right) illustrates how MPX maintains the base and bound metadata for each pointer. In short, an MPX bound table entry is indexed by the virtual address of the pointer (not the object). Similar to page tables, MPX uses two-level (pointer) address translation to store per-pointer metadata in memory, namely through the first-level *Bound Directory* (BD, 2GB on x86_64) and the second-level *Bound Table* (BT, 4MB). *Bound Directory Entry* (BDE, 8B)

Acronym	Full Phrase
BT	Bound Table
BTE	Bound Table Entry
BTP	Bound Table Page
BD	Bound Directory
BDE	Bound Directory Entry
HPQ	Hot BTP Queue
FAQ	Freed Address Queue

Table 1. Acronym table.

Types	Time Overhead				Space Overhead
	`malloc`	Ptr mani./arith.	`free`	Ptr deref.	
Identifier-based	low	high	-	high	high
Pointer graph-based	low	high	low	-	high
BOGO (this work)	low	-	medium	low	-

Table 2. Comparison of temporal memory safety solutions. Memory (de)allocations are typically less frequent than pointer manipulation, arithmetic, or dereference. Thus, metadata lookup/update on the latter are expensive. '-' means no overhead.

stores the starting address of a BT, and *Bound Table Entry* (BTE, 32B) holds the four types of metadata: the base, bound, pointer value, and unused space (8B each). A single BT (4MB) consists of 1024 *Bound Table Page* (BTP, 4KB), each of which embraces 128 BTEs. Table 1 lists the acronyms used in this paper. We refer to `BTE[p]` as the BTE of the pointer `p`; `BTP[p]` as the BTP that hosts the BTE of the pointer `p`; and other notations are defined similarly.

Notably, MPX does not provide temporal memory safety protection, requiring adoption of one of the following solutions to ensure full memory safety.

2.3 Temporal Memory Safety

Temporal memory safety violations occur when a program accesses a deallocated object. A multitude of solutions for temporal memory safety has been proposed. SAFECode [24, 25] and SVA [20] use pool allocation for objects with same. It guarantees dangling pointer dereference is within correct type pool thus mitigate the damage. Pool allocation has been applied to a page-protection based scheme [23] as well. Recently, EffectiveSan [26] proposes a lightweight type-based temporal memory safety checking, but it would miss use-after-free of the object with the same type. Others [12, 30, 45, 53] track per-object metadata: e.g., AddressSanitizer [53] poisons deallocated objects, and report access to the poisoned area.

Another line of temporal memory safety work also maintains per-pointer metadata like MPX, which can be broadly grouped into two categories.

Identifier-based scheme. As MPX keeps track of per-pointer bounds metadata for spatial safety checks, identifier-based solutions [10, 42, 66] maintain separate per-pointer metadata for temporal safety checks. For example, CETS [38, 42] associates each pointer with two identifiers named *key* and *lock address*, and propagates them along pointer manipulation such that the two values match if and only if a pointer is valid.

Pointer graph-based scheme. Pointer graph-based solutions such as DangNull [34] and DangSan [59] keep track of all the pointers to each memory object, building the pointer graph where nodes and edges are the objects and their connections, respectively. On `free`, they consult the pointer graph to *nullify* all the pointers to the object being deallocated, guaranteeing the absence of dangling pointers. A use-after-free error is detected in the form of a segmentation fault due to a null pointer dereference. A similar approach can be found in [13, 71].

Table 2 summarizes the sources of overhead for those two schemes. As a common downside, both maintain their own metadata for temporal memory safety. As a result, the overall overhead would be simply added up when they are combined with (per-pointer based) MPX for full memory safety.

In the next section, we introduce *BOGO*, a new temporal memory safety solution that reuses the bounds metadata tracked for spatial memory safety, and thus avoids additional cost for temporal memory safety metadata.

3 Overview of *BOGO*

The goal of *BOGO* is to provide full memory safety on top of MPX-enabled processors without significant overheads. With *BOGO*, users can buy such processors for spatial memory safety, and get temporal memory safety (almost) free; hence relieving the burden of the compiler and architectural support for temporal memory safety guarantee. More precisely, this paper focuses on temporal memory safety for heap objects (use-after-free), and *BOGO* in the current form does not provide temporal safety for stack objects (use-after-return). Additional support required for stack objects is discussed in §7.

Threat Model and Assumptions. *BOGO* relies on MPX for spatial memory safety, and adds temporal memory safety upon it. Therefore, *BOGO* assumes that underlying MPX-enabled processors can be trusted, and there are no hardware security bugs in the processor circuit fabrication [62, 67]. This work also assumes that adversaries cannot corrupt the MPX metadata by using non-memory-safety related attacks such as row hammer attacks [52, 60] or illegitimately having higher (root) privilege. Any attempts to corrupt the MPX metadata by exploiting memory safety vulnerabilities will be detected by *BOGO* itself. As *BOGO* relies on the soundness of MPX metadata, we further assume that MPX instrumentation is applied to all the source codes when soundness is required.

Overview. *BOGO* takes binary compiled with MPX instrumentation and transparently achieves temporal memory safety by reusing MPX metadata. At a high level: On `free` of a pointer `p`, it searches BTs for the entry whose bound overlaps with the object being deallocated. The existence of

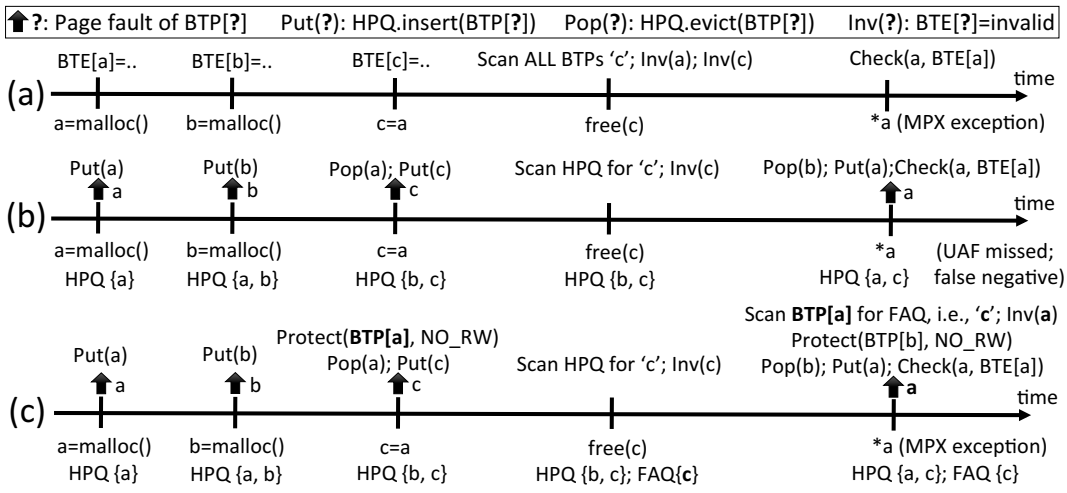

Figure 2. (a) FullScan; (b) PartialScan only; and (c) PartialScan and PageFaultScan. (b) misses the use-after-free error, and (c) solves the problem with PageFaultScan. For brevity, (b) and (c) omit the bound creation and propagation. A bold up-arrow represents a BTP fault. `Put` and `Pop` inserts/evicts a BTP to/from HPQ. `Inv` invalidates a BTE.

such a BTE implies that another pointer, say `q`, also pointing to the same object, becomes a dangling pointer. When found, it invalidates the metadata of dangling pointers. Later dereference of pointer `q` will be checked by MPX (for the default spatial memory safety), leading to an `OutOfBound` exception because of the invalidated bound. The beauty of *BOGO* is that it enforces temporal memory safety by triggering the violation of spatial memory safety. Users can differentiate temporal from spatial violations by checking a special value in the bound register.

BOGO attempts to eliminate dangling pointers on `free` like the aforementioned pointer graph-based temporal memory safety solutions. However, there is one big difference. *BOGO* does not maintain additional metadata (e.g., pointer graph) for temporal memory safety. Instead, it reuses MPX metadata as is, and scans the BTs to invalidate dangling pointers.

FullScan. Consider an example in Figure 2(a). On `free(c)`, a naive FullScan checks all BTs, finds aliased pointer `a`, and invalidates `BTE[a]` so that later use of dangling pointer `a` would result in an MPX exception. However, searching the whole MPX BTs can lead to unacceptable performance degradation due to the large search space, thus care must be taken to minimize the cost.

PartialScan. To bound the scan cost on `free`, *BOGO* tracks a small set of hot, recently used BTPs (Bound Table Pages) using a page protection mechanism, keeps them in the *Hot BTP Queue* (HPQ), and performs PartialScan that looks for dangling pointers only on the hot BTPs in the current HPQ.

Figure 2(b) shows an example. Suppose the bounds of pointers `a`, `b`, and `c` are stored in different BTPs: `BTP[a]`, `BTP[b]`, and `BTP[c]`. Further assume that the size of HPQ is 2. Each page fault on the first pointer access causes the corresponding BTP to be inserted into HPQ. When HPQ becomes full on the `BTP[c]` page fault (at the statement `c=a`), *BOGO* evicts the cold (least recently added) `BTP[a]` from the HPQ and inserts the hot `BTP[c]` into the HPQ. Then, `free(c)` checks dangling pointers only against `BTP[b]` and `BTP[c]` in the current HPQ, a small subset of all BTPs.

However, this PartialScan may lead to a false negative (i.e., missing a use-after-free error) when a dangling pointer happens to be in a cold BTP. In Figure 2(b), as `BTP[a]` is not in HPQ, `BTE[a]` was not scanned/invalidated by `free(c)`, even though `a` and `c` are aliased. Thus, the following dereference of the dangling pointer, `*a`, remains undetected.

PageFaultScan. *BOGO* introduces PageFaultScan to guarantee sound temporal memory safety. When a BTP is evicted from HPQ, *BOGO* marks the cold BTP not-readable and not-writable so that the later access to the cold BTP can be trapped by a page fault. Note that when a pointer is dereferenced, MPX always accesses its BTE for the spatial memory safety check, resulting in a page fault. Meanwhile, on `free`, *BOGO* also tracks the freed addresses in the Freed Address Queue (FAQ) if they are partially checked only over hot BTPs: i.e., by PartialScan, not by FullScan. Upon a page fault of a cold BTP, *BOGO* checks against freed addresses in FAQ to see whether there exists any dangling pointer to the addresses in the cold BTP. Then, *BOGO* recovers page access permissions of the BTP and puts it into the HPQ.

Using the same example, Figure 2(c) illustrates that *BOGO* guarantees the detection of all use-after-free errors using both PartialScan and PageFaultScan. Though PartialScan on `free(c)` did not invalidate the bound of `BTE[a]`, the dereference of pointer `a` would result in a page fault on which PageFaultScan can detect the use-after-free error by scanning the `BTP[a]` for the freed address `c` stored in FAQ.

4 *BOGO* Approach Details

This section presents how *BOGO* tracks hot BTPs to bound the scan cost on `free` (§4.1); how PartialScan and PageFaultScan can ensure no false negative (§4.2); and how to achieve redundancy-free and false-positive-free PageFaultScan (§4.3).

4.1 Hot Bound Table Page Tracking

Figure 3 (Lines 2-12) shows how *BOGO* makes use of a page protection mechanism to track hot BTPs at a low cost. Upon a BTP fault, *BOGO* restores the read/write permissions (Line 3) and puts this "hot" (most recently accessed) BTP into the bounded Hot BTP Queue (HPQ) (Line 9). When the HPQ is full, *BOGO* evicts the "coldest" (least recently added) BTP in a FIFO manner, and makes it not-readable and not-writable (Lines 9-12). The latter part of this section discusses the rest of the BTP fault handler.

4.2 PartialScan +PageFaultScan =Low Overhead+No False Negative

On `free`, *BOGO* scans BTs and invalidates the BTE of dangling pointers. Figure 3 (Lines 14-21) presents how *BOGO* instruments `free`. *BOGO* can safely rely on PartialScan and PageFaultScan as long as it can hold free addresses in the FAQ. In some cases, the FAQ can be configured to be large enough to avoid FullScan. In other cases, if the FAQ becomes full, *BOGO* falls back to FullScan that checks all the free addresses in FAQ over all the BTs (Line 20). We discuss FullScan optimization in §5.2. After FullScan, *BOGO* may reset FAQ (Line 21) as there are no longer pending temporal memory safety checks to perform. Any unused BTs can be safely reclaimed at this point. This approach trades performance for soundness.

For performance, *BOGO* favors PartialScan (Line 18) that looks up dangling pointers only over hot BTPs in the HPQ. To ensure soundness (see the difference between Figure 2(b) and Figure 2(c)), the free addresses that are checked via PartialScan are collected in the Free Address Queue (FAQ) along with the free time[3] (Line 16). These free addresses remain in the FAQ until the next FullScan (Line 20-21), and meanwhile they are checked over the (cold) BTPs resulting in page faults via PageFaultScan (Line 4-8). The `evict_time` and `free_time` will be discussed in the next section.

4.3 PageFaultScan+RedundancyPredication = Low Overhead + No False Positive

Though PageFaultScan ensures no false negatives, it may lead to a false positive. Consider the following code:

```
a=malloc(8); b=malloc(64); c=a; free(c);
a=malloc(8); *b; c=b; free(b); *a;
```

The pointer `a` was once a dangling pointer after `free(c)`, but is reassigned by the second `a=malloc(8)` of the same size, rendering `*a` legal. However, if this `malloc` reuses a freed object for locality, which is the case for modern allocators, PageFaultScan may raise a false alarm.

Figure 4(a) illustrates the case with the heap snapshot change over time. At time `t1`, `a=malloc(8)` returns `0x10` and sets `BTE[a]`. At `t3`, `BTP[a]` is evicted from HPQ. At `t4` on `free(c)`, PartialScan does not check `BTP[a]`, and puts the freed address (`0x10`) in FAQ. At `t5`, the second `a=malloc(8)` happens to return the same location `0x10`, as shown in the third heap snapshot. At `t7`, `BTP[a]` becomes cold again. As a result, the last `*a` at `t9` results in a page fault on `BTP[a]`. PageFaultScan finds an overlap between `BTE[a]` and `0x10` in FAQ. However, this is a false alarm.

One naive workaround would be not to release the memory to the system on `free` so that the freed object cannot be reused for later allocation, until *BOGO* performs FullScan. However, obviously, it will increase the memory footprint significantly.

Redundancy Predication. We present a novel *redundancy prediction* technique. The crux of the problem is due to redundant checks. Focus on the three cases: `t4` when PartialScan adds the freed address `0x10` into the FAQ; `t7` when `BTP[a]` becomes cold; and `t9` when PageFaultScan performs a check on the freed address `0x10`. Recall that we originally introduced PageFaultScan because PartialScan did not perform a check on cold BTPs at that time. However, in this scenario, `BTP[a]` becomes cold after a PartialScan, implying that PageFaultScan does not need to check the freed address `0x10` on `BTP[a]`. More formally speaking, it is safe for PageFaultScan to skip the scanning of BTP for a given freed address `faddr`

```
1 /* [btp] and [faddr] form a single element list
       with the parameter */
2 OnBoundTablePageFault(btp)
3   mprotect(btp,RW)
4   evict_time = get_evict_time(btp)
5   for each faddr in FAQ
6     free_time = get_free_time(faddr)
7     if (evict_time < free_time)
8       scan([btp],[faddr]) // PageFaultScan
9   evicted_btp = insert(HPQ,btp)
10  if (evicted_btp != NULL)
11    set_evict_time(evicted_btp)
12    mprotect(evicted_btp,NONE)
13
14 OnFree(faddr)
15  free(faddr)          // actual free
16  insert(FAQ,faddr)    // index as free time
17  if (FAQ.length != MAX)
18    scan(HPQ,[faddr]) // PartialScan
19  else
20    scan(ALL,FAQ)      // FullScan
21    reset(FAQ)
22
23 scan(btp_list,faddr_list)
24  for each btp in btp_list
25    for each faddr in faddr_list
26      // scan and invalidate if overlaps
27      for each bte in btp
28        if(bte.base<=faddr && faddr<=bte.bound)
29           bte.base = INVALID
30           bte.bound = INVALID
```

Figure 3. *BOGO* handler algorithms.

[3]This is an abstract time that we implement as the FAQ index, to avoid the cost of using real timestamps.

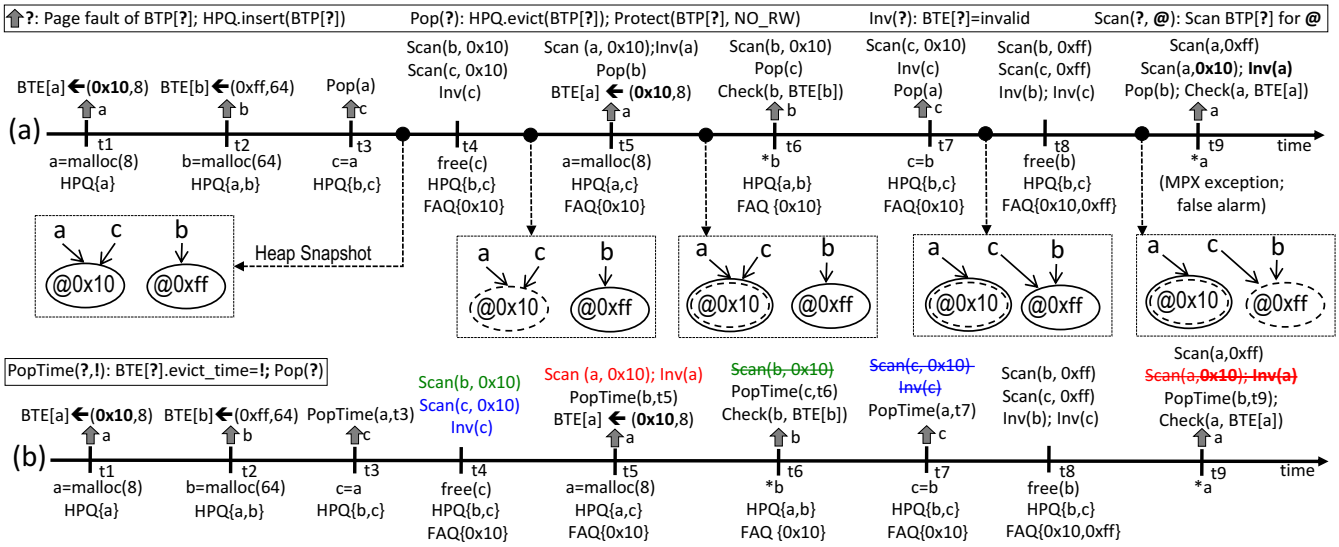

Figure 4. Redundancy prediction: (a) shows a false positive case, and (b) shows how *BOGO* removes the redundant scans and eliminates the false positive. Actions of *BOGO* appears above the bar while the status of HPQ/FAQ does underneath the code. Each color represents a different redundant scan.

in FAQ if $Time^{BTP}_{evict}$ is greater (i.e., later) than $Time^{faddr}_{freed}$. We provide the proof at the end of this section.

Based on this observation, *BOGO* keeps track of $Time^{faddr}_{freed}$ when `faddr` is added in FAQ (Line 16), and $Time^{BTP}_{evict}$ when a `BTP` is evicted from HPQ (Line 11). On PageFaultScan, *BOGO* compares their times (Lines 4-7), and avoids redundant checks, leading to better performance and no false positives.

To illustrate, Figure 4(b) shows how *BOGO* deals with the false positive with redundancy prediction. Since the eviction time of the `BTP[a]` (`t7`) is greater than the freed time of `0x10` (`t4`), PageFaultScan (`t9`) safely skips `Scan(a,0x10)` and avoids the false alarm. For the same reason, *BOGO* skips PageFaultScan at times `t6, t7` for `BTP[b]` and `BTP[c]`, respectively. However, PageFaultScan at `t5` cannot be skipped, and it needs to check the faulted page `BTP[a]` with the freed address `0x10`. Note, for all the scans that can be safely skipped, they are paired with the corresponding previous scan that performs the same bound check. Such a pair is shown using the same color in Figure 4(b), where three pairs exist (green, blue, and red).

Now we prove that this redundancy elimination is safe.

Theorem 4.1. *On a BTP fault, it is safe (no missing detection) for PageFaultScan to skip the scanning of the BTP for a given freed address faddr in FAQ if* $Time^{BTP}_{evict}$ *is larger than* $Time^{faddr}_{freed}$.

Proof. We provide a direct proof sketch. Recall *BOGO* basically has two scan methods, PageFaultScan and the free time PartialScan which checks HPQ. Thus, we need to prove that either method has already scanned the BTP that satisfies the time condition, i.e., $Time^{faddr}_{freed} < Time^{BTP}_{evict}$. First, if the BTP was a part of HPQ at $Time^{faddr}_{freed}$, the `free` must have scanned the BTP obviously. Thus, this case makes it redundant to scan the BTP in the current PageFaultScan (referred to as *currPFS*).

Second, if PartialScan on `free` did not scan the BTP (i.e., it was not in the HPQ at $Time^{faddr}_{freed}$), it must have been evicted before; let's refer to the time as $Time^{BTP}_{pastEvict}$. The implication is there must be a PageFaultScan (referred to as *pastPFS*) between two evictions, and it must have happened after the `free` which otherwise would have scanned the BTP, and we get the following:

$$Time^{BTP}_{pastEvict} < Time^{faddr}_{freed} < Time^{BTP}_{pastPFS} < Time^{BTP}_{evict} < Time^{BTP}_{currPFS} \quad (1)$$

Then, we investigate if the *pastPFS* scanned the BTP. As shown above inequality, for the *pastPFS*, the eviction time of the BTP ($Time^{BTP}_{pastEvict}$) is not larger than the free time ($Time^{faddr}_{freed}$). Thus, *pastPFS* must have scanned the BTP, making it redundant to scan the BTP in *currPFS*. Consequently, Theorem 4.1 must be true. □

In sum, with PageFaultScan and redundancy prediction, *BOGO* can eliminate unnecessary scans, achieving better performance and no false positives.

5 Optimization

5.1 No PageFaultScan Optimization

On `free`, when FAQ is not full, *BOGO* performs PartialScan that checks hot BTPs and stores the freed address into FAQ so

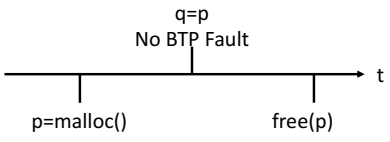

Figure 5. Example of sound PartialScan. No further PageFaultScan is required.

	Partial Scan		Page Fault Scan		Full Scan	
	number of scans	per-scan cost $O(\|HPQ\|)$	number of scans	per-scan cost $O(\|FAQ\|)$	number of scans	per-scan cost $O(\|ALL\| * \|FAQ\|)$
HPQ ↑	-	↑	↓	-	-	-
FAQ ↑	↑	-	-	↑	↓	↑

Table 3. Impacts of increasing HPQ and FAQ on the number and the cost of Partial, Page Fault, and Full Scans.

that later PageFaultScans can detect dangling pointers that PartialScan might miss (§4.2). This PageFaultScan backup mechanism is necessary because dangling pointers may have resided in cold BTPs. If *BOGO* can prove the absence of dangling points in the cold BTPs, then it does not need to add the freed address into the FAQ, bringing two benefits: 1) to save the FAQ space (triggering FullScan slowly) and 2) to avoid succeeding PageFaultScans against the freed address.

Consider an example in Figure 5 where there were no BTP faults between the memory allocation and deallocation. The absence of BTP faults implies that any potential copy of pointer `p`, the necessary condition of dangling pointers, must have happened only with those pointers whose BTEs lie in the hot BTPs. Otherwise, a BTP fault would have been triggered and HPQ has been altered. In this case, on `free`, *BOGO* only needs to check the freed address with the hot BTPs in the current HPQ, and there is no need to add it to the FAQ for later PageFaultScans. In practice, applications often have short-living heap objects where `malloc` and `free` are adjacent to each other in time.

To support this optimization, *BOGO* maintains a small hash table which tracks the addresses of objects that have been allocated since the last BTP fault. *BOGO* stores the address being allocated on `malloc`, and checks if the hash table holds the address being deallocated on `free`. When there is a match, PartialScan applies the proposed optimization by not putting the freed address into the FAQ. *BOGO* resets the hash table on a BTP fault (as a part of HPQ maintenance).

5.2 FullScan Optimization

When FAQ is full, *BOGO* performs FullScan over the entire BTs (Figure 3 Line 20). A naive implementation would iterate over the (1^{st}-level) BD to find all the valid (2^{nd}-level) BTs. Scanning the huge BD leads to severe performance degradation. Even for a valid BT, many of its BTPs may have not been accessed, and thus scanning them would cause unnecessary page faults. To avoid scanning the BD and all BTs, which would be an order of magnitude slower, *BOGO* uses a custom syscall to get only the accessed BTPs, and scans them directly. The syscall looks up per-process memory descriptor for virtual memory area(VMA) reserved for MPX and returns those pages whose *accessed bit* is set in the page table.

6 Dynamic Adaptation of Queue Size

BOGO maintains two queues: HPQ and FAQ. Their sizes have a significant impact on the number and cost of Partial, Page Fault, and Full Scans that determine the overall performance. This section first analyzes the cost of each scan (§6.1) and the impacts of HPQ and FAQ sizes (§6.2). Then, we introduce *scan cost*-based dynamic adaptive scheme that adjusts the size of HPQ at runtime for optimal performance (§6.3).

6.1 Scan Cost Analysis

In general, the cost of each scan is the product of the number of BTPs to scan and the number of free addresses to scan (Figure 3 Lines 24-30). Note that the innermost loop in Line 27 iterates over (constant number of) 128 BTEs (of size 32B each) in a BTP (of size 4KB). Therefore, the cost of PartialScan is $O(|HPQ|)$ as it scans the Hot BTPs in the HPQ against a single pointer address being freed. The cost of PageFaultScan is $O(|FAQ|)$ as it checks the freed addresses in the FAQ against a single (once cold now hot) BTP being page faulted. Lastly, the cost of FullScan is $O(|ALL| * |FAQ|)$ as it checks the freed addresses in the FAQ against all BTPs.

6.2 Impact of HPQ and FAQ Sizes

Table 3 summarizes the impacts of increasing the size of HPQ and FAQ on the number and the cost of each scan, while decreasing its sizes has an opposite effect.

Increasing HPQ. HPQ keeps track of Hot BTPs to perform PartialScan, and thus increasing the size of HPQ would increase the cost of PartialScan. The number of PartialScans is irrelevant to the size of HPQ. To be precise, the number depends on the `free` frequency and the size of FAQ (as it determines which Partial or Full Scan to take on `free`). On the other hand, increasing HPQ would decrease the number of PageFaultScans as HPQ can hold more hot BTPs. Note that the total cost of each scan would be proportional to the number of scans and the per-scan cost. The size of HPQ has opposite impacts on these two scans. Therefore, for the common cases where PartialScan and PageFaultScan are used (without FullScan), the size of HPQ should be tuned to make a good balance on both scans. Our sensitivity study on the HPQ size in §9.3.4 shows that each application has a different optimal HPQ size, motivating our adaptive scheme in §6.3.

Increasing FAQ. Unlike HPQ, the bigger FAQ, in general, leads to better performance. First, its impacts on PartialScan is small because the number of PartialScans varies slightly.

Second, increasing FAQ at a glance may look like harming PageFaultScan as it iterates over the free addresses in the FAQ. However, in reality, this is not true because PageFaultScan stops scanning when it finds a freed address whose free time is earlier than the eviction time of the BTP being page faulted as discussed in §4.3 and Figure 4. In other words, the cost of PageFaultScan even with the very large FAQ is in effect bounded. Lastly, as the cost of FullScan is way more expensive than the other two scans, it is better to keep its number low by making the FAQ big enough. Therefore, in the next section, we focus on tuning the HPQ size at runtime.

6.3 Scan Cost-based HPQ Adaptive Scheme

Based on the above observation, *BOGO* dynamically adjusts the size of HPQ to balance PartialScans and PageFaultScans. To this end, *BOGO* divides a program execution into regular intervals, called *quanta*. Then, at runtime *BOGO* measures and compares the cost (execution time) of PartialScans and PageFaultScans in a quantum, and then decides whether to reduce or enlarge the size of HPQ for the next quantum. If the cost of PageFaultScan is higher than that of PartialScan, *BOGO* increases the HPQ, and vice versa. This scan cost based adaptive scheme allows *BOGO* to adapt application specific characteristics (e.g., `free` frequencies) and the program phase changes even within the same application. Our experimental results in §9.3 show that *BOGO* with adaptive HPQ outperforms the one with profiled-based manual configuration. By default, *BOGO* uses the quantum of 100 ms, sets the initial size of HPQ to be 16, and changes the HPQ size exponentially.

7 Discussion

Free-after-free. Consider the following code:

```
a=malloc(8); b=a; free(a); a=malloc(8); free(b);
```

Suppose two mallocs are allocated to the same region. Then, `b` would free `a`'s buffer. If one sees `free(b)` as the use of `b`, *BOGO* can perform a bound check on `free(b)`. As *BOGO* invalidates the `BTE[b]` on `free(a)`, it can detect such a case.

Use-after-return. This refers to dereference of deallocated stack object [27]. *BOGO* can be extended to detect it by invalidating the bounds belonging to current stackframe upon return. Static analysis can help to avoid such a check in many cases, thus supporting the detection at a low cost. For instance, static analysis can tell whether there is a pointer that points to the stackframe and escapes the function. Such cases are expected to be rare: (1) a pointer to the current stackframe is returned; (2) a pointer to the stackframe is propagated across the function. It implies that for most of returns, the scan could be avoided.

Multithreading. Metadata-based memory error detectors may lead to false positives or false negatives when a pointer operation and metadata updates/checks do not happen in an atomic manner. For solutions like CETS [42] that only lookup, update, and check per-pointer metadata at a time, if a program is data-race-free, atomicity could be achieved by placing instrumentation codes into the same critical section as the original pointer operation. For a program with data races, this remains a challenge. On the other hand, for solutions like DangSan [59] and *BOGO*, that access other pointer's metadata (for invalidation), the problem gets worse because concurrent metadata updates from independent pointers may form a race condition: e.g., while one scans on `free`, another may update the metadata. DangSan chooses to favor performance over soundness without additional support. The current *BOGO* prototype shares the same limitation. However, it is possible to mark the BTPs to be scanned as non-accessible during scanning and make a concurrent thread wait at the page granularity (instead of stopping the world). This design remains future work. During our experiments with multithreaded applications (§9.5), we did not observe false warnings (false negatives are unknown).

Custom Memory Allocator. They need to be patched to invoke *BOGO*'s scan, which otherwise may lead to false negatives. They can be identified by using techniques like [15].

8 Implementation

The llvm-mpx pass consists of 9240 LoC, along with 1605 LoC in LLVM framework diff. The custom syscall consists of 419 LoC in the kernel diff.

8.1 Spatial Memory Safety

We implemented `llvm-mpx` pass using LLVM [33] to support spatial memory safety on MPX. It instruments at the IR level, protecting heap, address-taken stack and global objects. It follows the same per-pointer bound checking convention used in SoftBound [41] and `gcc-mpx`. Yet, it instruments more instructions than SoftBound: e.g., atomic, vectorization, and invoke instructions. Moreover, `llvm-mpx` models the same set of `libc` functions as `gcc-mpx`: e.g., `malloc`, `memcpy`, and `strcpy`. As the address of the pointers being checked is always taken (e.g., `bndldx(&p)`) in the IR, pointer variables are not promoted to registers, and `llvm-mpx` checks them all. `llvm-mpx` keeps the BT up-to-date, which *BOGO* relies on, while `gcc-mpx` and `icc-mpx` often store bound information in the stack instead of BT.

Optimizations. `llvm-mpx` performs three optimizations during the instrumentation: **(1)** Bound check elimination: if memory access can be statically verified, it elides MPX checks. This is analogous to bound check optimization in the pioneering work of Gupta [28]. `gcc-mpx` also has the same form of optimization. **(2)** Dead bound elimination: The lack of bound checks can make the corresponding bound and the related instructions (e.g., `bndmk/bndldx/bndstx`) dead if they are not "used" by others. it identifies such dead codes by following the use-def chain [37] and eliminates them. **(3)** Bound check consolidation: if it can statically calculate the

range of the access in a loop or a vectorized code, it consolidates the checks into one check and pays the overhead only once. This is a very simple form of optimization proposed in Gupta's work [28] and WPBound [68].

Since the above optimizations are safe for spatial memory safety [28, 68], they do not compromise *BOGO*'s temporal memory safety guarantee. For example, optimization (1) only deals with local arrays or globals, not the heap objects that are the target of *BOGO*. Optimization (2) won't trigger if pointers are copied, returned, etc. (i.e., bound is "used").

Bound narrowing. The current prototype does not implement bound narrowing [8]. When a program accesses a specific field of a struct object, the compiler can shrink the bound to that field, rather than the full object, for the fine-grained bounds checking. However, this causes a compatibility issue breaking some SPEC 2006 applications with C idioms due to the resulting false positives [16, 47]. Bound narrowing is optional in `gcc-mpx`, and not supported in many other tools [24, 39, 40, 43, 54] including SoftBound [38, 41].

8.2 Temporal Memory Safety

BOGO is built upon LLVM-4.0, glibc-2.23 and linux-4.10. The kernel is modified to support the followings: (1) BTP permission initialization; (2) FullScan optimization (§5.2); (3) custom `mprotect` avoiding touch unrelated kernel data structures; and (4) signal delivery when a BTP is reclaimed (becomes unavailable) so that it can be removed from HPQ, avoiding a potential segmentation fault during scanning.

9 Evaluation

9.1 Methodology

We used three sets of benchmarks for evaluation. SPEC CPU 2006 is used in §9.3 for detailed performance evaluation. The malloc-free benchmark [31, 46] is used in §9.4 for stress-test. Finally, 9 real-world (multithreaded) applications are tested in §9.5. The setup is a 4GHz quad-core Intel i7-6700K CPU with 16GB RAM. The performance numbers are the average of 5 runs. Except where otherwise mentioned, all experiments are done with the following configurations: (1) the size of FAQ is 65535 and FullScan is used when it becomes full; (2) the initial size of HPQ is 16 with dynamic adaptive scheme (§6.3) enabled; (3) `reference` input is used for SPEC.

Patching Spatial Safety Errors. With `llvm-mpx`, we found the same set of bounds errors in original SPEC applications as reported in Oleksenko et al.'s work [47] (see their Section 4.4). Thus, we patched [7] them to perform bound-error-free performance evaluation[4].

Instrumentation before or after Optimizations. It is worth noting that `llvm-mpx` pass is applied after standard optimizations including LTO (link-time optimization). That is, apply -O2 for each bitcode file, then perform -O2 LTO to create a single file. `llvm-mpx` is applied at last. The same convention of using all possible optimizations before the instrumentation was adopted in SoftBound [41] and others [38, 42, 56, 69].

We investigated the high runtime overhead of `icc-mpx` and `gcc-mpx`. By scrutinizing the order of applied compiler passes in `gcc-mpx` and `icc-mpx`, we noticed that they first performed MPX instrumentation thus preventing other optimizations. For example, `gcc-mpx`'s instrumentation happens very early in the compiler pass order, i.e., the 12th among 174 passes. And those before the instrumentation are not actually optimization passes. Therefore, all the optimizations can be significantly restricted, e.g., dead code elimination can be suppressed due to the inserted MPX bounds checking code.

Figure 6 highlights the impact of the optimize-before-instrument convention on the performance overhead of `llvm-mpx`; each bar represents the average overhead of all SPEC 2006 applications which is normalized to that of baseline with no spatial safety support. By following the convention, our `llvm-mpx` incurs 1.26x slowdown in the 6th bar: O2→LTO→Inst. When we instrument before optimizations like `gcc-mpx` and `icc-mpx`, the overhead is significantly increased, i.e., 3.35x slowdown in the 4th bar: Inst→O2→LTO.

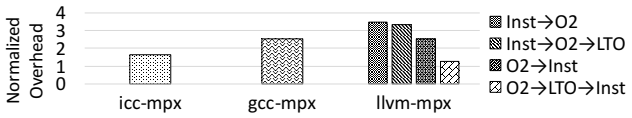

Figure 6. MPX compilers overheads: geomean of SPEC 2006.

	Source	Application	Bug manifest
Spatial	BugBench [36]	bc-1.06	storage.c:177 util.c:577
		gzip-1.2.4	gzip.c:828
		man-1.5h1	main.c:977
		ncompress	compress42.c:892
		polymorph-0.4.0	polymorph.c:120,195,198,200,202
	CVE-2004-2167	latex2rtf-1.9.15	definitions.c:155
	CVE-2007-4060	corehttp-0.5.3alpha	http.c:32
	CVE-2011-4971	memcached-1.4.5	memmove()
	CVE-2016-6289	php-7.1Git-2016-06-29	zend_virtual_cwd.c:1243
	CVE-2016-6297	php-7.1Git-2016-06-30	zip_stream.c:289
	CVE-2017-9928	lrzip-0.631	lrzip.c:979
	CVE-2017-9929	lrzip-0.631	lrzip.c:1074
	CVE-2018-5268	OpenCV-3.3.1	grfmt_jpeg2000.cpp:343
	CVE-2018-6187	MuPDF-1.12.0	pdf-write.c:2901
	SPEC 2006	400.perlbench	perlio.c:748, sv.c:4124
		450.soplex	islist.h:287,357 svector.h:351
		464.h264ref	mv-search.c:1016
Temporal	NIST/Juliet [4]	102205 102226 102248 102287 102307 102311 102367 102444 102528 102609 102611 102613 102615 102617 102619 152889 102225 102247 102267 102289 102308 102321 102411 102468 102577 102610 102612 102614 102616 102618 102663 2151	
	CVE-2014-9661	FreeType 2.5.3	ftstream.c:182
	CVE-2015-7801	optipng-0.6.4	opngoptim.c:977
	CVE-2017-10686	nasm-2.14rc0	dereferences of free'd Token obj
	CVE-2017-15642	sox-v14.4.2	formats.c:245

Table 4. llvm-mpx and BOGO validation.

[4]For `soplex`, we manually modified the pointer manipulations that violate standard memory model, and made their bounds checking always succeed. See discussion in [47] Section 4.4.

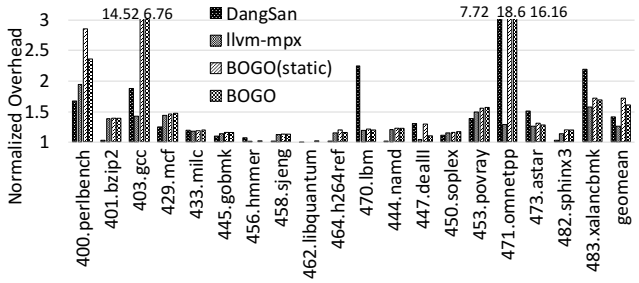

Figure 7. Performance Overhead.

Application	Partial	Full	Page Fault	Application	Partial	Full	Page Fault
400.perlbench	816926.10	0.49	2875.50	470.lbm	0.03	0	0.01
401.bzip2	0.34	0	0.04	444.namd	4.74	0	0.15
403.gcc	29474.74	0.25	201114.61	447.dealII	770810.54	9.63	0.05
429.mcf	0.03	0	18.30	450.soplex	1432.80	0	641.12
433.milc	18.41	0	0.01	453.povray	17648.83	0	5318.45
445.gobmk	1808.81	0	1386.05	471.omnetpp	73419.43	0.35	214350.09
456.hmmer	4426.50	0	0.28	473.astar	13613.71	0.10	1321.86
458.sjeng	0.01	0	0.01	482.sphinx3	38908.55	0.58	985.62
462.libquantum	0.45	0	0.01	483.xalancbmk	607803.72	0.44	0.25
464.h264ref	514.04	0	599.12				

Table 5. Frequency of Partial, Full, and Page Fault Scans (per second). The sum of Partial and Full Scans represents `free` frequency.

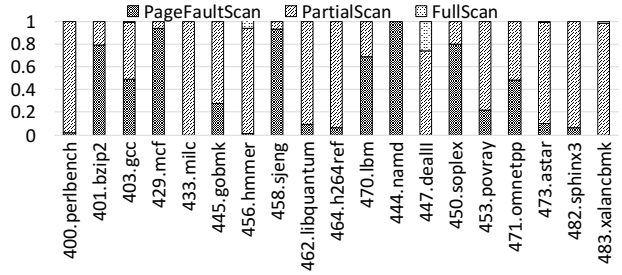

Figure 8. *BOGO* Performance Overhead Breakdown.

We also found out that Oleksenko et al. [47] did not apply LTO for both `gcc-mpx` and `icc-mpx`. However, it turns out that applying LTO after the instrumentation does not improve the performance significantly. This is confirmed by the small height gap between the 3rd (Inst→O2) the 4th (Inst→O2→LTO) bars in Figure 6. We believe that the same phenomenon will be observed in `gcc-mpx` and `icc-mpx` because LTO is restricted anyway by the inserted MPX bound checks. Thus, we conclude that the reason for the poor performance of `gcc-mpx` and `icc-mpx` is mainly due to their unconformity of optimize-before-instrument convention.

9.2 Security Evaluation

`llvm-mpx`'s Spatial Memory Safety. *BOGO*'s ability to detect use-after-free hinges on spatial memory safety solution. Thus, it is critical to validate whether `llvm-mpx` is sound. For a fair and accurate comparison, we picked LLVM-based SoftBound [41] as baseline, instead of comparing across different compilers (e.g., `llvm-mpx` vs. `gcc-mpx`). To this end, we collected the number of dynamic bounds checks performed on a subset of tested applications with reference input; the open source version of SoftBound [41] currently works for only 6 SPEC applications, all of which we tested. We confirmed that `llvm-mpx` performs a higher number of bounds checks than SoftBound because the former supports more instructions (§8.1). Oleksenko et al. [47] also report that `gcc-mpx` leads to a much higher instruction count than `icc-mpx` (~3x vs. ~1.5x – see their Figure 10). To further validate `llvm-mpx`, we tested real-world applications with buffer overflow bugs[5]. As in Table 4, the first 5 cases are from BugBench [36] followed by 9 CVEs. `llvm-mpx` detected all without false positives. Note that `llvm-mpx` also detected known bugs in SPEC [7].

Based on the above validation steps, we conclude that our `llvm-mpx` implementation is credible thus being able to serve as a solid basis for *BOGO*'s temporal safety enforcement.

***BOGO*'s Temporal Memory Safety.** We empirically evaluated *BOGO*'s implementation for temporal memory safety. First, we inspected *BOGO*'s detection capability for 32 cases from NIST/Juliet (CWE416, Use-After-Free) [4], as listed in Table 4. *BOGO* soundly detected them all. Second, *BOGO* detected all use-after-free vulnerabilities in 4 tested CVEs.

9.3 SPEC CPU 2006 Benchmark

9.3.1 Performance Overhead.

Figure 7 shows the performance overhead normalized to the baseline without memory safety. For each application, there are four bars to compare: DangSan (temporal-only), llvm-mpx (spatial-only), *BOGO* (static), and *BOGO*. The first bar is for DangSan, the state-of-the-art temporal memory safety only solution. It does not support use-after-return. We evaluated it using the open source version from GitHub [3]. On average, DangSan incurs 1.41x slowdown. The second bar is for `llvm-mpx`. On average, `llvm-mpx` incurs 1.26x slowdown. The next two bars show the performance overhead of *BOGO* (static) and *BOGO* which provide both spatial (via `llvm-mpx`) and temporal memory safety. The third bar, i.e., *BOGO* (static), reports the best per-app result selected by profiling with varying HPQ size (the sensitivity study on HPQ is shown in §9.3.4). The last bar, i.e., *BOGO*, shows that its scan cost-based dynamic adaptation of HPQ size (§6.3) outperforms the best static configuration, i.e., *BOGO* (static). We observed significant improvements for `perlbench`, `gcc` where *BOGO* could dynamically adapt to the phase changes of runtime execution behaviors. Note, the dynamic adaptation scheme can offer similar or better performance without prior profiling or knowledge of the program behavior. On average, *BOGO* incurs 1.6x slowdown for spatial and temporal safety.

We then present detailed performance overhead analysis for *BOGO*. First, Table 5 shows the frequencies of PartialScan,

[5]RIPE [64] is not used as it does not support the 64-bit system.

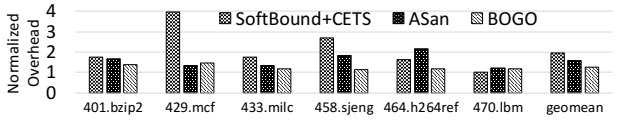

Figure 9. Performance Overhead of Full Memory Safety Solutions.

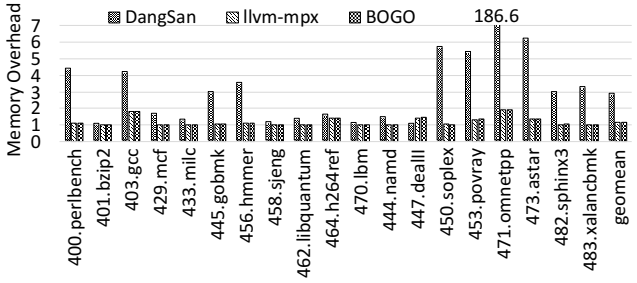

Figure 10. Memory Overhead.

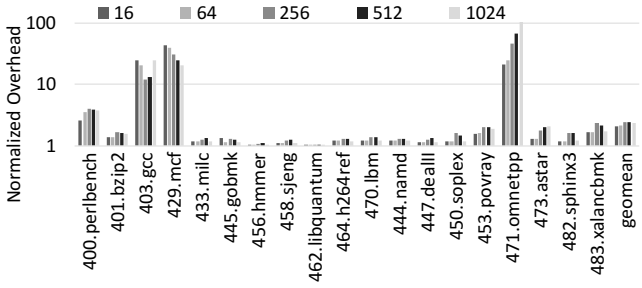

Figure 11. Sensitivity study: varying HPQ, fixed-size FAQ.

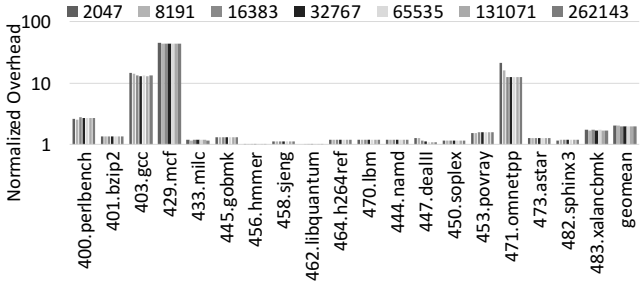

Figure 12. Sensitivity study: varying FAQ, fixed-size HPQ.

FullScan, and PageFaultScan (numbers/sec). Overall, the frequency of FullScan is very small shows the benefit of PartialScan+PageFaultScan. We found that naive FullScan only approach incurs more than 10x slowdown (not shown in Figure 7). Note that the sum of Partial and Full Scan frequencies represents `free` frequency, and it varies significantly across different applications (up to 817K/sec). This implies that the SPEC benchmarks cover a broad spectrum of the deallocation behaviors which affect *BOGO*'s performance overhead. Later, we stress-test *BOGO* with malloc-free intensive benchmarks (§9.4) and evaluate it with real-world applications (§9.5).

Figure 8 reports *BOGO*'s performance overhead breakdown of time spent for three scans. Two applications incur relatively high *BOGO* overhead: `gcc` and `omnetpp`. It turns out that `gcc` and `omnetpp` have frequent Partial and Page Fault Scans as shown in Table 5. In general, applications with higher scan frequencies (e.g., `perlbench`, `xalancbmk`) incur higher overhead compared to the others. `gcc` and `omnetpp` also suffer from scanning larger dataset in HPQ and FAQ, According to Figure 8, `gcc` and `omnetpp` spent about 50:50 on PartialScan and PageFaultScan, showing the effectiveness of HPQ dynamic scheme for applications with such high overhead.

9.3.2 Other Full Memory Safety Techniques

Overhead would add up when combining a temporal memory safety solution (e.g. DangSan) with another spatial memory safety solution (e.g., `llvm-mpx`). As shown in Figure 7, DangSan and `llvm-mpx` incur 1.41x and 1.26x slowdown, respectively. When combined, the total runtime overhead would be similar to *BOGO* (1.6x), yet *BOGO* is more memory efficient (§9.3.3).

SoftBound+CETS [38, 42] keeps separate per-pointer metadata for spatial memory safety (SoftBound) and temporal memory safety (CETS). Figure 9 highlights the performance of *BOGO* compared to other full memory safety solutions for 6/19 SPEC 2006 applications; the latest open source version of SoftBound+CETS [6] is broken for the remaining 13 applications. For those 6 applications, *BOGO* (1.25x slowdown) significantly outperforms SoftBound+CETS (1.94x); (Their paper reports 1.75x overhead for 9/19 SPEC 2006 and 8/16 SPEC 2000 applications). It seems that the open source version might be less optimized. Although SoftBound+CETS supports use-after-return detection, it is disabled for fair comparison. **AddressSanitizer** [53] maintains per-object metadata. Their paper reports 1.73x slowdown for SPEC 2006, higher than *BOGO* (1.6x). When we run it, with use-after-return disabled, for the same 6 applications in Figure 9, it incurs 1.57x slowdown which is still higher than *BOGO* (1.25x) but lower than the SoftBound+CETS (1.94x). AddressSanitizer quarantines freed memory and defer actual reclamation to support use-after-free detection, causing memory bloat. To avoid high memory overhead, it does actual free periodically, thereby sacrificing soundness.

9.3.3 Memory Overhead

Figure 10 shows the memory usage of DangSan (temporal-only), `llvm-mpx` (spatial-only), and *BOGO*. It is measured by taking the average of resident memory (VmRSS) and normalized to the baseline without memory safety. On average, DangSan incurs 2.84x memory overhead, which is slightly higher than what the paper reports (2.4x) [59], while llvm-mpx and *BOGO* incur 1.16x and 1.17x, respectively. Thus, *BOGO* adds very small memory overhead for full memory safety. When DangSan is combined with MPX for the full

Figure 13. Malloc-free benchmark performance. The bar graphs shows the normalized overhead (left y-axis). The line graph shows the free frequency (right y-axis).

Application	LOC	Test method	Free Freq.
aget	1K	download 4GB file, 8 threads	0.45
pfscan	2K	search 4GB file, 8 threads	27
pbzip2	6K	compress 4GB file, 8 threads	5,280
transmission	116K	download file 3.8GB	13,354
memcached	18K	YCSB [18], workload ABCDEF	0.35
cherokee	102K	ab [2], 8 conc. clients, 100K req.	27,509
nginx	166K	ab [2], 8 conc. clients, 100K req.	115,113
apache	270K	ab [2], 8 conc. clients, 100K req.	486
mysql	1,473K	sysbench [5], 8 conc. clients, 100K req.	677,774

Table 6. Real-world application test methods. Top four applications are utilities/clients, while the bottom fives are servers.

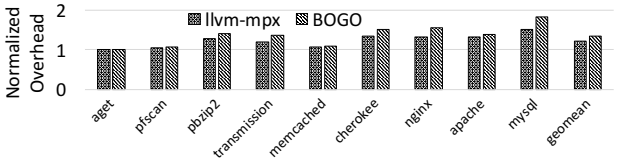

Figure 14. Real-world application performance.

safety, the total space overhead would add up. For `omnetpp`, DangSan suffers from 186.62x (134.65x according to the paper [59]) overhead, while *BOGO* incurs only 1.92x overhead compared to llvm-mpx (1.91x). Although DangSan maintains a huge pointer graph for performance, it still incurs 7x slowdown for `omnetpp` as shown in Figure 7.

9.3.4 Sensitivity Study

Dynamic adaptation is disabled in this study. Figures 11 and 12 show the performance sensitivity studies with respect to the sizes of HPQ and FAQ, respectively. Figure 11 shows the result of varying HPQ with the fixed-size (65535) FAQ. As discussed in §6.2, the size of HPQ affects PartialScan and PageFaultScan in opposing directions. Each application favors different sizes of HPQ. For example, `omnetpp` prefers small HPQ, `mcf` favors larger HPQ, and `gcc` works best on the middle size HPQ. This justifies *BOGO*'s dynamic HPQ size adaptation, and Figure 7 confirms its effectiveness. On the other hand, a bigger FAQ is in general preferable as it reduces the number of FullScans. For example, `omnetpp` in Figure 12 definitely favors a larger FAQ. We found that the rest applications are not very sensitive to the size of FAQ, though there is some fluctuation. For this reason, *BOGO* does not currently adjust FAQ on the fly.

9.4 Malloc/Free Benchmark

Stress-testing *BOGO* with malloc/free intensive applications [31, 46] shows higher runtime overhead (Figure 13) than SPEC: on average, 2.7x slowdown for *BOGO*, and 1.27x for `llvm-mpx` only. The reason is two-fold: (1) the huge amount of malloc/free (up to 5.8M/s) puts significant pressure on *BOGO*'s page scan mechanism; and (2) the execution time of the applications is very short (less than 2 seconds) even with the largest input, and the majority of the entire execution time is spent allocating/deallocating numerous objects. Thus, *BOGO*'s activity ends up taking a significant portion. However, except for `perl` (5.8M/s) and `espresso` (3.9M/s), the overhead of the remaining applications is under 80%, and the overhead added by *BOGO* upon `llvm-mpx` is only 34%.

9.5 Real-World Applications

We evaluated *BOGO* with 9 real-world applications using the test cases listed in Table 6. The five servers including `apache` and `mysql` are set up with default configuration. While `nginx` is a single-threaded multi-process server, all the rest 8 are multithreaded applications. As discussed in §7, *BOGO* does not guarantee soundness for multithreaded applications as with others [41, 42, 47, 59]. Thus, this experiment is just for performance evaluation and compatibility demonstration purposes. We note that all instrumented applications behave correctly. Despite no soundness guarantee, as reported in §9.2, *BOGO* (llvm-mpx) could detect a buffer overflow bug in `memcached-1.4.5`. Given the workloads, the free frequency varies up to 678K per second. As shown in Figure 14, the runtime overhead of *BOGO* ranges from 1x to 1.83x, with a geomean of 1.34x, which is less than that of more CPU-intensive and malloc/free-frequent SPEC applications.

10 Conclusion

This paper presents *BOGO*, seamlessly adding temporal memory safety to the spatial memory safety on Intel MPX. *BOGO* scans bound metadata to find dangling pointers, invalidates their bounds, and detects temporal memory safety violations as spatial safety violations. This frees *BOGO* from maintaining separate metadata for temporal memory safety, saving both runtime and space overhead. Our evaluation shows that *BOGO* supports full memory safety at comparable runtime overhead and much less memory overhead than other state-of-the-art solutions. All the source code of *BOGO* and `llvm-mpx` is available at https://github.com/lzto/bogo .

Acknowledgments

The authors would like to thank anonymous reviewers for their valuable feedback. This paper is based upon work supported by the National Science Foundation under Grant No. CSR-1750503 and CSR-1814430.

References

[1] [n. d.]. 2011 CWE/SANS Top 25 Most Dangerous Software Errors. ([n. d.]). http://cwe.mitre.org/top25/.

[2] [n. d.]. ab - Apache HTTP server benchmarking tool. ([n. d.]). https://httpd.apache.org/docs/2.4/programs/ab.html.

[3] [n. d.]. DangSan Open Source Implementation. ([n. d.]). https://github.com/vusec/dangsan.

[4] [n. d.]. NIST Software Assurance Reference Dataset Project. ([n. d.]). https://samate.nist.gov/SARD.

[5] [n. d.]. Scriptable database and system performance benchmark. ([n. d.]). https://github.com/akopytov/sysbench.

[6] [n. d.]. SoftBound+CETS Open Source Implementation. ([n. d.]). https://github.com/santoshn/softboundcets-34.

[7] [n. d.]. Spec2006 AddressSanitizer Patch. ([n. d.]). https://github.com/google/sanitizers/ blob/master/address-sanitizer/spec/spec2006-asan.patch.

[8] [n. d.]. Struct Bound Narrowing. ([n. d.]). https://gcc.gnu.org/wiki/Intel MPX support in the GCC compiler#Narrowing.

[9] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-compatible Defense Against Out-of-bounds Errors. In *Proceedings of the 18th Conference on USENIX Security Symposium (SSYM'09)*. 51–66.

[10] Todd M Austin, Scott E Breach, and Gurindar S Sohi. 1994. *Efficient detection of all pointer and array access errors*. Vol. 29. ACM.

[11] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 30–40.

[12] Nathan Burow, Derrick McKee, Scott A Carr, and Mathias Payer. 2017. CUP: Comprehensive User-Space Protection for C/C++. *arXiv preprint arXiv:1704.05004 (to appear in AsiaCCS'18)* (2017).

[13] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 2012. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 133–143.

[14] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented Programming Without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10)*. ACM, New York, NY, USA, 559–572. https://doi.org/10.1145/1866307.1866370

[15] Xi Chen, Asia Slowinska, and Herbert Bos. 2016. On the Detection of Custom Memory Allocators in C Binaries. *Empirical Softw. Engg.* 21, 3 (June 2016), 753–777. https://doi.org/10.1007/s10664-015-9362-z

[16] David Chisnall, Colin Rothwell, Robert NM Watson, Jonathan Woodruff, Munraj Vadera, Simon W Moore, Michael Roe, Brooks Davis, and Peter G Neumann. 2015. Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 117–130.

[17] Tzi-cker Chiueh and Fu-Hau Hsu. 2001. RAD: A compile-time solution to buffer overflow attacks. In *Distributed Computing Systems, 2001. 21st International Conference on*. IEEE, 409–417.

[18] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[19] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks.. In *Usenix Security*, Vol. 98. 63–78.

[20] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. 2007. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. ACM, New York, NY, USA, 351–366. https://doi.org/10.1145/1294261.1294295

[21] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. 103–114.

[22] Dinakar Dhurjati and Vikram Adve. 2006. Backwards-compatible Array Bounds Checking for C with Very Low Overhead. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. 162–171.

[23] Dinakar Dhurjati and Vikram Adve. 2006. Efficiently detecting all dangling pointer uses in production servers. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*. IEEE, 269–280.

[24] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. 2006. SAFECode: Enforcing Alias Analysis for Weakly Typed Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 144–157. https://doi.org/10.1145/1133981.1133999

[25] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. 2005. Memory Safety Without Garbage Collection for Embedded Applications. *ACM Trans. Embed. Comput. Syst.* 4, 1 (Feb. 2005), 73–111. https://doi.org/10.1145/1053271.1053275

[26] Gregory J Duck and Roland HC Yap. 2017. EffectiveSan: Type and Memory Error Detection using Dynamically Typed C/C++. *arXiv preprint arXiv:1710.06125 (to appear in PLDI'18)* (2017).

[27] Google. 2017. AddressSanitizerUseAfterReturn. (2017). https://github.com/google/sanitizers/wiki/AddressSanitizerUseAfterReturn

[28] Rajiv Gupta. 1993. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)* 2, 1-4 (1993), 135–150.

[29] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C.. In *USENIX Annual Technical Conference, General Track*. 275–288.

[30] Richard W M Jones, Paul H J Kelly, Most C, and Uncaught Errors. 1997. Backwards-compatible bounds checking for arrays and pointers in C programs. In *in Distributed Enterprise Applications. HP Labs Tech Report*. 255–283.

[31] Changhee Jung, Sangho Lee, Easwaran Raman, and Santosh Pande. 2014. Automated Memory Leak Detection for Production Use. In *Proceedings of the 36th International Conference on Software Engineering*.

[32] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. 2018. Delta Pointers: Buffer Overflow Checks Without the Checks. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, New York, NY, USA, Article 22, 14 pages. https://doi.org/10.1145/3190508.3190553

[33] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. 75–.

[34] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing Use-after-free with Dangling Pointers Nullification.. In *NDSS*.

[35] Zhengyang Liu and John Criswell. 2017. Flexible and Efficient Memory Object Metadata. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management (ISMM 2017)*. ACM, New York, NY, USA, 36–46. https://doi.org/10.1145/3092255.3092268

[36] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. 2005. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, Vol. 5.

[37] S.S. Muchnick. 1997. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers.

[38] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. 2015. Everything you want to know about pointer-based checking. In

LIPIcs-Leibniz International Proceedings in Informatics, Vol. 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[39] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. 189–200.

[40] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2014. WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. 175:175–175:184.

[41] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. 245–258.

[42] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *ACM Sigplan Notices*, Vol. 45. ACM, 31–40.

[43] Santosh Ganapati Nagarakatte. 2012. *Practical low-overhead enforcement of memory safety for c programs*. Ph.D. Dissertation. University of Pennsylvania.

[44] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27, 3 (2005), 477–526.

[45] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. 89–100.

[46] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. 2009. Efficiently and precisely locating memory leaks and bloat. In *Proc. of the 30th PLDI*.

[47] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches. *CoRR* abs/1702.00719 (2017). arXiv:1702.00719 http://arxiv.org/abs/1702.00719

[48] Marco Prandini and Marco Ramilli. 2012. Return-oriented programming. *IEEE Security & Privacy* 10, 6 (2012), 84–87.

[49] Feng Qin, Shan Lu, and Yuanyuan Zhou. 2005. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*. IEEE, 291–302.

[50] Olatunji Ruwase and Monica S Lam. 2004. A Practical Dynamic Buffer Overflow Detector.. In *NDSS*, Vol. 2004. 159–169.

[51] Pawel Sarbinowski, Vasileios P Kemerlis, Cristiano Giuffrida, and Elias Athanasopoulos. 2016. VTPin: practical VTable hijacking protection for binaries. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 448–459.

[52] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat* (2015), 7–9.

[53] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker.. In *USENIX Annual Technical Conference*. 309–318.

[54] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 28–28. http://dl.acm.org/citation.cfm?id=2342821.2342849

[55] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. 2016. HDFI: Hardware-Assisted Data-flow Isolation. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 1–17.

[56] Yulei Sui, Ding Ye, Yu Su, and Jingling Xue. 2016. Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions. *IEEE Transactions on Reliability* 65, 4 (2016), 1682–1699.

[57] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 48–62.

[58] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. 2011. On the Expressiveness of Return-into-libc Attacks. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID'11)*. Springer-Verlag, Berlin, Heidelberg, 121–141. https://doi.org/10.1007/978-3-642-23644-0_7

[59] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. DangSan: Scalable Use-after-free Detection. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 405–419.

[60] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 1675–1689. https://doi.org/10.1145/2976749.2978406

[61] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. 2007. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*. IEEE, 273–284.

[62] Xiaoxiao Wang, Mohammad Tehranipoor, and Jim Plusquellic. 2008. Detecting malicious inclusions in secure hardware: Challenges and solutions. In *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*. IEEE, 15–19.

[63] John Wilander and Mariam Kamkar. 2003. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention.. In *NDSS*, Vol. 3. 149–162.

[64] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. 2011. RIPE: Runtime Intrusion Prevention Evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC '11)*. ACM, New York, NY, USA, 41–50. https://doi.org/10.1145/2076732.2076739

[65] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE, 457–468.

[66] Wei Xu, Daniel C DuVarney, and R Sekar. 2004. An efficient and backwards-compatible transformation to ensure memory safety of C programs. *ACM SIGSOFT Software Engineering Notes* 29, 6 (2004), 117–126.

[67] Kaiyuan Yang, Matthew Hicks, Qing Dong, Todd Austin, and Dennis Sylvester. 2016. A2: Analog malicious hardware. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 18–37.

[68] Ding Ye, Yu Su, Yulei Sui, and Jingling Xue. 2014. WPBound: Enforcing spatial memory safety efficiently at runtime with weakest preconditions. In *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*. IEEE, 88–99.

[69] Ding Ye, Yu Su, Yulei Sui, and Jingling Xue. 2014. WPBOUND: Enforcing Spatial Memory Safety Efficiently at Runtime with Weakest Preconditions. In *Proceedings of the 2014 IEEE 25th International Symposium on Software Reliability Engineering (ISSRE '14)*. IEEE Computer Society, Washington, DC, USA, 88–99. https://doi.org/10.1109/ISSRE.2014.20

[70] Suan Hsi Yong and Susan Horwitz. 2003. Protecting C programs from attacks via invalid pointer dereferences. In *ACM SIGSOFT Software Engineering Notes*, Vol. 28. ACM, 307–316.

[71] Yves Younan. 2015. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers.. In *NDSS*.