HeapExpo: Pinpointing Promoted Pointers to Prevent Use-After-Free Vulnerabilities

Zekun Shen
New York University
New York, USA
zekun.shen@nyu.edu

Brendan Dolan-Gavitt
New York University
New York, USA
brendandg@nyu.edu

ABSTRACT

Use-after-free (UAF) vulnerabilities, in which dangling pointers remain after memory is released, remain a persistent problem for applications written in C and C++. In order to protect legacy code, prior work has attempted to track pointer propagation and invalidate dangling pointers at deallocation time, but this work has gaps in coverage, as it lacks support for tracking program variables promoted to CPU registers. Moreover, we find that these gaps can significantly hamper detection of UAF bugs: in a preliminary study with OSS-Fuzz, we found that more than half of the UAFs in real-world programs we examined (10/19) could not be detected by prior systems due to register promotion. In this paper, we introduce HeapExpo, a new system that fills this gap in coverage by parsimoniously identifying potential dangling pointer variables that may be lifted into registers by the compiler and marking them as volatile. In our experiments, we find that HeapExpo effectively detects UAFs missed by other systems with an overhead of 35% on the majority of SPEC CPU2006 and 66% when including two benchmarks that have high amounts of pointer propagation.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Dangling pointers, use-after-free, memory errors

ACM Reference Format:
Zekun Shen and Brendan Dolan-Gavitt. 2020. HeapExpo: Pinpointing Promoted Pointers to Prevent Use-After-Free Vulnerabilities. In *Annual Computer Security Applications Conference (ACSAC 2020), December 7–11, 2020, Austin, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3427228.3427645

1 INTRODUCTION

Use-after-free (UAF) vulnerabilities are widespread in C and C++ programs. Although some programming techniques can reduce the prevalence of use-after-free bugs, vulnerabilities are still regularly found: in 2017 and 2018, 289 and 303 vulnerabilities classified as UAFs were reported to the CVE Project, and 375 were found in the last year [8]. Use-after-free bugs are found in a wide variety of projects, including web browsers, utility programs and libraries.

The root cause of use-after-free vulnerabilities is dangling pointers which point to freed memory without being correctly cleared. If a dangling pointer is dereferenced, it may access memory of another object, leading to either information leak vulnerabilities (if the access is a read) or memory corruption (if the access is a write).

Use-after-free bugs are hard to detect and debug because they may not cause a crash immediately. Manual review for use-after-free vulnerabilities is time-consuming and does not scale to large programs. Analysis of heap-related code requires reviewers to understand the pointer propagation behavior throughout the entire code base. Although heap-related code is only a small portion of the whole program, there are many possible allocation and deallocation sequences, and use-after-free bugs may only manifest only under a small fraction of those sequences. As a result, it can take significant effort to manually uncover use-after-free vulnerabilities.

Prior work, such as FreeSentry [28], DangNull [16], DangSan [26], and pSweeper [13] has sought to eliminate dangling pointers by invalidating them automatically when releasing dynamic memory. This approach incurs less overhead than other tools like Address Sanitizer [25], which checks the validity of every memory read or write. The downside the of invalidation approach is its unwanted false-negatives: previous work cannot track local variables and function parameters which are promoted to registers. Because promoting stack memory to registers is a common compiler optimization, this renders many potential dangling pointers untrackable. In our analysis of 19 bugs from the OSS-Fuzz [9, 23] project (described in Section 4.1), we found that local variables and function parameters appear often in use-after-free bugs: 10 of the 19 bugs we examined are caused by variables that were promoted to registers by standard compiler optimization and would have been missed by prior work.

To close this gap in coverage, in this paper we present HeapExpo, a dangling pointer sanitizer that also tracks pointers in local variables and function arguments. As with previous works, we achieve pointer tracking by using LLVM infrastructure [15] to instrument pointer propagation instructions and provide a runtime library to track and manage metadata about allocations. Our analysis identifies pointer variables that may be optimized into registers by the compiler and marks them as volatile, forcing the compiler to keep them on the stack where they can be tracked by our runtime. However, because a naïve approach of marking every pointer variable adds prohibitive overhead, we additionally provide a static analysis that safely identifies pointer variables that can never be involved in use-after-free bugs, and allows these to be optimized freely.

Despite providing more comprehensive coverage, our optimized implementation has an overhead of 66% on the SPEC CPU2006

ACSAC 2020, December 7–11, 2020, Austin, USA

ACM ISBN 978-1-4503-8858-0/20/12…$15.00
https://doi.org/10.1145/3427228.3427645

benchmark [14], only a modest increase compared to the 46% overhead from the state-of-the-art dangling pointer sanitizer, DangSan [26].

We make the following contributions:

- We identified some major sources of dangling pointers that are not tracked by previous works due to register promotion by studying and categorizing UAF bugs discovered by the OSS-Fuzz project.
- We present a novel approach to identify and track pointers promoted to registers by marking them as volatile.
- We design an analysis that uses liveness and the program's function call graph to reduce the number of tracked stack variables, greatly reducing the overhead for comprehensive tracking.
- We provide a performance analysis that demonstrates our system incurs about 20% run-time overhead in addition to DangSan, with little extra memory overhead.

2 BACKGROUND

In this section, we walk through how previous work solves the dangling pointer problem, their limitations and the LLVM tool chain we studied and used. We also discuss the implementation of a state of the art dangling pointer sanitizer, DangSan, which helps the understanding of our design as a whole. Additionally, we discuss two threading models that influence our optimization.

2.1 LLVM Compiler

LLVM is a modular compiler tool chain. The front-end client, clang, translates the source code into an Intermediate Representation (LLVM IR) with a limited number of instructions. Optimizing passes process LLVM IR code and produce optimized LLVM IR. Common optimization techniques like dead code removal, function inlining, and alias analysis are applied in this phase. The LLVM IR is then handed to a target-dependent backend compiler to generate machine code for the host. Finally, the linker is invoked to combine all machine code files into a single executable.

LLVM Optimization. The LLVM front end first generates unoptimized code as in Listing 3 from the C code in Listing 2. At this point, all local variables and function arguments live on the stack. **a** lives in the stack location indicated by the **alloca** instruction at line 2. Then, the LLVM optimizer processes the unoptimized code with the `mem2reg` pass [10], which promotes memory references to registers. This is one of the very first optimization passes to obtain Single Source Assignment form. The end result is shown in Listing 4. LLVM provides options to instrument before or after the `mem2reg` pass by choosing an appropriate optimization stage.

The algorithm to determine whether a stack variable can be promoted is shown in Listing 1. The algorithm shows that a regular non-volatile pointer, either as a local variable or a function argument, is indeed promotable. Later, our analysis of OSS-Fuzz bugs (Section 4.1) follows the cases in this algorithm.

2.2 Temporal safety system design

Currently, there are four main approaches to ensure temporal safety: secure allocators, address-based checking, garbage collection, and dangling pointer invalidation. We compare the four approaches in Table 1. Our system focuses on the pointer invalidation approach because it is fast, has low memory overhead, and is hard to bypass. HeapExpo closes a major gap in coverage of dangling pointer detectors with only a modest increase in overhead.

```
1  bool isAllocaPromotable(AllocaInst *AI) {
2      if (hasVolatileStoreOrLoad(AI)):
3          return False;
4      else if (addressIsStoredSomewhere(AI))
5          return False;
6      else if (hasNonPtrCast(AI))
7          return False;
8      else if (isNontrivialStruct(AI))
9          return False;
10     else if (isArray(AI))
11         return False;
12     else if (isStruct(AI))
13         return False
14     return True;
15 }
```

Listing 1: llvm::isAllocaPromotable

Secure allocator [3, 6, 22]. This approach provides a custom allocator which restricts reuse of same memory region. This prevents a dangling pointer from pointing to other allocated objects, making it unexploitable. However, this approach can be bypassed if the memory reuse pattern can be learned by an attacker as discussed by Lee et al. [16].

Address-based checking [4, 19, 20, 25]. This type of temporal safety approach tries to invalidate the memory addresses of a heap object when it is released. Another often-used technique is to raise an alert when dangling pointers are used. This approach usually discourages memory reuse so that a dangling pointer does not point to a new object right away.

Garbage collection [1, 2, 27]. This is a passive reference counting technique that scans memory for potential pointers. Dynamic memory is released only when there are no pointer references. Therefore, this method can only mitigate use-after-free, not detect it. The runtime overhead is tied to the number of memory scans performed, so it often trades memory for speed.

Dangling pointer invalidation [13, 16, 26, 28]. This approach tracks the propagation of pointers inside memory. The propagation is tracked by taint analysis or monitoring certain instructions. When a heap object is released, the pointers that reference the object also get invalidated. The invalidation can be performed by setting the dangling pointer to kernel space or null. Dereference of a kernel space dangling pointer results in a crash immediately. Setting the pointer to null can let the program execute normally if there is null pointer check.

2.3 Limitations of Prior Work

Prior work [16, 26, 28] that keeps an active representation of memory is based on LLVM, with instrumentation done by LLVM passes. DangNull [16] only tracks pointers in the data and heap sections

Approach	Run-time	Memory	Bypassable	Detectable	False Negative	Worst Case
Secure allocator	Low	Low	Yes	Yes	Low	Allocation Intense
Address-based checking	High	High	Yes	Yes	Low	Memory Access Intense
Garbage collection	Low	Medium	No	No	Low	Allocation Intense
Dangling pointer invalidation	Low	Medium	No	Yes	**High**	Propagation Intense

Table 1: Comparison among Approaches Solving Use-after-free

Approach	Data	Heap	Stack	Promoted
DangNull	Yes	Yes	No	No
FreeSentry	Yes	Yes	Yes	No
DangSan	Yes	Yes	Yes	No
HeapExpo	Yes	Yes	Yes	Yes

Table 2: Trackable pointer sources among previous work. Promoted means registers promoted from stack locations.

```
1  int main() {
2      char* a = malloc(8);
3      free(a);
4      printf("%s\n", a);
5      return 0;
6  }
```

Listing 2: Sample C code

and does not track pointers on the stack or in registers. FreeSentry [28] and DangSan [26] optionally support tracking of pointers on the stack, but do not track those in registers. The authors of all three systems mention this limitation, but they do not provide an estimation of how many false negative it causes. We have summarized coverage among previous works in Table 2.

We have covered LLVM's `mem2reg` [10] optimization pass in the previous subsection. LLVM uses this pass to promote local variables from stack memory to registers. Since these works [16, 26, 28] instrument optimized code, they do not track promoted local variables and function arguments, which leaves a large portion of pointer code uninstrumented.

The C code in Listing 2 has a use-after-free vulnerability with its local variable **a**. Compiling the code with -O0 or -O2 can result in the following IR code in Listings 3 and 4. Past works instrument the IR after `mem2reg` pass; thus, they process the IR code in Listing 4, where **a** has been promoted from stack location to LLVM register. FreeSentry and DangSan cannot track **a** in this context, because it does not have stack location. To ensure temporal safety with low time overhead, they sacrifice completeness. In our work, we target tracking of local variables and function arguments.

2.4 DangSan Implementation

DangSan is a dangling pointer sanitizer implemented using the LLVM compiler tool chain. It consists of compile-time LLVM passes and run-time code linked to the final binaries. The LLVM pass instruments pointer write instructions with calls to tracking code. The run-time code overloads all allocator functions including `malloc`,

```
1  define i32 @main() {
2      %1 = alloca i8*
3      %2 = call @malloc(i64 8)
4      store i8* %2, i8** %1
5      %3 = load i8*, i8** %1
6      call @free(i8* %3)
7      %4 = load i8*, i8** %1
8      %5 = call @printf(i8* "%s\n", i8* %4);
9      ret i64 0
10 }
```

Listing 3: LLVM IR with -O0, local variable lives at stack address at line 2

```
1  define i32 @main() {
2      %1 = call malloc(i64 8)
3      call @free(i8* %1)
4      %2 = call @puts(i8* %1)
5  }
```

Listing 4: LLVM IR with -O2, local variable lives in %1 as LLVM register

`calloc`, `free`, `realloc`, `memalign`, `aligned_alloc`, `valloc`, `pvalloc` and `posix_memalign` to keep track of heap memory layout. The run-time code also provides functions to track pointer propagation so that instrumented calls can update the data structure.

2.4.1 Heap Data Structure. The heap is represented as a directed graph. Allocated memory blocks are nodes and pointers are directed edges that link memory blocks. Pointers to heap memory can exist in heap memory, the data section of the executable, stack memory, and registers. In DangSan's design, each dynamic memory block has associated metadata which tracks the sets of all inward pointers. DangSan chose not to record outward edges in the metadata for performance reasons.

The metadata of the heap objects is kept in a three-level shadow memory map. Querying the metadata of dynamic objects takes constant time with three memory reads.

The metadata consists of sets of pointers, and is implemented in the manner of a file system. When the direct log is filled up, an alternative hash map is used instead. The hash map is used to eliminate duplicates to prevent the log from growing without constraints.

```
1  void init_heapobj(void* ptr, size_t size) {
2      metadata *m = new_metadata(ptr, size);
3      insert_metadata_range(ptr, size, m);
4  }
5
6  void freeptr(void* ptr, size_t size) {
7      metadata *m = get_metadata(ptr);
8      for (uintptr_t loc : m->record) {
9          uintptr_t val = *(uintptr_t*)loc
10         if (val >= ptr &&
11             val < ptr + size ) {
12             invalidate_ptr(loc);
13         }
14     }
15     delete_metadata(m);
16 }
```

Listing 5: Alloc and Dealloc function hook

DangSan modifies gperftools' tcmalloc and adds hook functions to memory allocation and deallocation. All allocator functions are mapped to basic `malloc()` and `free()` functions. Because `memcpy()` is not tracked by DangSan, a `realloc()` is treated as a `malloc()` followed by a `free()`. As shown in Listing 5, allocation and deallocation hooks have the job of managing the three-level shadow memory. The allocation hook marks the region of the memory object with the metadata, while the deallocation hook clears the metadata. The deallocation hook has the additional task of managing dangling pointers. For all inward pointers in the metadata record, a value check is performed. If the pointer still refers to the freed object, the pointer is invalidated by setting its most significant bit to 1.

The pointer records are created with `regptr` calls that are instrumented by the LLVM pass. The `regptr` function checks if the pointer points to a recorded object and, if so, records the pointer in the metadata of the object. The pseudo-code is shown in Listing 6.

DangSan also includes another LLVM pass that checks the data section of the program and inserts a global constructor function that registers global variables in data sections.

2.4.2 Tracking pointers in memory. The compile-time instrumentation is designed as an LLVM pass that processes LLVM IR. Propagating pointers to data and heap sections guarantees a memory write instruction. LLVM IR uses the `store` instruction for memory writes, and it also indicates the type of stored values. Therefore, one only needs to instrument `store` instructions with pointer type:

store PtrTy* ptr_loc, PtrTy ptr_val

2.5 Threading Model

In this section, we discuss two essential threading models that affect our implementation: the global and thread-local assumptions. In the most general case, the **global model** assumes that a registered pointer may be invalidated by a `free()` in *any* thread. By contrast, **thread-local model** assumes that a registered pointer may only be invalidated by a `free()` in the *same* thread.

```
1  void regptr(uintptr_t loc, uintptr_t val){}
2      metadata* obj = meta_get(val);
3      if (obj) {
4          if (!in_lookback(obj, loc))
5              register(obj, loc);
6      }
7  }
```

Listing 6: Functions that provide pointer tracking

Our pointer invalidation system follows the global model by maintaining a synchronous data structure. We ensure the correctness of the heap representation under multiple threads. A `free()` invalidates pointers stored on the stack for every thread.

However, to enable further optimizations for local variable tracking, we rely on the thread-local model. This model allows further static analysis and covers most common situations. We note that this assumption is shared by prior work; DangSan and FreeSentry both rely on it for their loop optimization pass.

3 DESIGN AND IMPLEMENTATION

As shown in Figure 1, we designed HeapExpo on top of DangSan, keeping its instrumentation and runtime libraries. DangSan's transformation passes run at link time, when stack variables have already been promoted to registers. We therefore add another step before the `mem2reg` pass to preserve relevant stack variables. When DangSan's optimizations take place, these variables are still tracked as on stack.

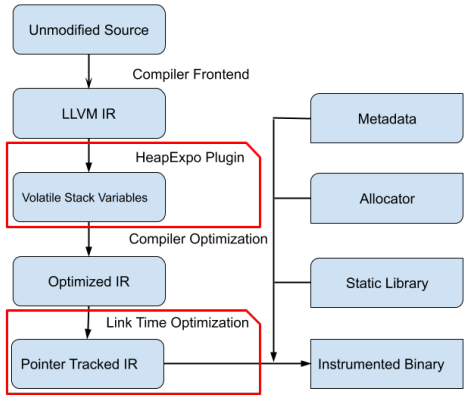

Figure 1: Overall Design of HeapExpo

3.1 Tracking Local Variable and Function Argument

As previously discussed, dangling pointers can be stored in the data section, on the heap, on the stack, and in registers. We have discussed how prior work achieved tracking of pointers in memory. In this section, we discuss how we track pointers promoted to registers.

```
1  void f(char *ptr) {
2      strcpy(ptr, "string1");
3      free(ptr);
4  }
5  void g(char *ptr) {
6      *ptr = 's';
7  }
8  int main() {
9      char *s1, *s2, *s3;
10     char c1, c2;
11     s1 = malloc(10);
12     regptr(&s1, s1); //reg s1
13     s2 = malloc(10);
14     f(s1);
15     s3 = malloc(10);
16     c1 = *s1;
17     g(s3);
18     c2 = *s3;
19 }
```

Listing 7: Local variable instrumentation example

Burden of register tracking. Computers have limited number of registers. Therefore, the LLVM back-end maps the unbounded set of LLVM registers to machine registers according to their liveness. Because LLVM passes operate at the IR level, our pass is not aware of machine registers and thus cannot easily track them. To ensure complete coverage, invalidating pointers in registers is required. To solve this problem, we propose a method that tracks local variables and function arguments without checking machine registers; instead, we identify pointer variables that need to be tracked and prevent them from being promoted to registers in the first place.

We run our LLVM pass before the `mem2reg` pass; at this point, local variables and function arguments are all stored on the stack. To track a local variable or function argument, we mark it as volatile, ensuring that further optimization will keep the variable on the stack. We do this by simply marking the relevant `store` and `load` instructions as volatile in our LLVM pass. We do not modify other variables and arguments and let the `mem2reg` pass promote them to registers.

When a memory block is released, we check all associated pointers, including volatilized pointers. If they still point to the region, we invalidate the pointers by making them point to invalid memory space. Dereference of the pointers thus causes a segmentation fault.

One downside of this strategy is that it results extra memory reads and writes, adding to the overhead to our system. To make HeapExpo practical, we must therefore develop optimization techniques that only track pointers that can potentially cause use-after-free. Overall, the optimizations can reduce number of instrumented instructions by half, leading to a significant speedup. We describe these optimizations next.

3.2 Liveness Analysis

Tracking local variables by keeping them on the stack has a performance impact because the compiler cannot optimize them into registers. To reduce number of tracked variables, we apply liveness analysis to local pointers and pointer arguments to functions based on control flow within each function.

We say that a local pointer becomes a *live* dangling pointer when (1) a function call invalidates it, and (2) it may be dereferenced later in the function execution. There is no need to instrument local pointers that are not live. In this way, we reduce the number of instrumentation added and the number of stack variables tracked.

First, we find all definitions and uses of local pointer variables and function parameters. Our LLVM pass takes place before the `mem2reg` pass, so every `store` instruction is a definition, and every `load` instruction is a use. At each function call, we make every live stack variables volatile. Specifically, we mark as volatile the last `store` before the call instruction and the first `load` after it.

In Listing 7, suppose we perform a check for **main** at line 14. We want to examine the liveness of local pointers s1, s2, and s3 at call to f(). s1 is defined at line 11 and used at line 16. It is possible that the value of s1 is invalidated inside f(). Hence we make s1 volatile by making the store and load volatile. Also, we register the stack location of s1 for tracking. s2 is defined at line 13 but never used after f(). Thus, we do not need to track s2, as it cannot cause a use-after-free. Although s3 is used at line 18, it is always guarded by the definition at line 15. s3 may be invalidated in f(), but the definition will always overwrite the old address before use. We thus do not need to track s3 either.

In the example above, we should repeat the above check with every function call in **main**, but we will show next that the check at g() is not necessary.

With the control flow graph of the function, we can easily assess the liveness of a variable. If there is a path from the function entry point to some definition of a local pointer, and then to the call instruction, it means that the pointer is potentially defined at the call instruction. Similarly, if there is a path from the call instruction to a usage, without encountering any definitions, it means the local pointer is potentially dereferenceable after the call instruction. If the local pointer is both potentially defined before the function call, and potentially dereferenceable after the function call, we say the pointer variable is live. We repeat the process with every call instruction in the function to obtain a set of live local pointers to track. Any local variables and function parameters that are not live can be safely promoted to LLVM registers and optimized.

3.3 Call Graph Analysis

In the previous subsection, we examined the liveness of every local pointer variable at each function call, assuming that the variable may be freed inside the called function. Here, we introduce a call graph analysis that can reduce the number of liveness checks.

Under the thread-local assumption, a stack pointer can only be invalidated inside its thread, which means the function the pointer is in must call free() to release memory. A call to a function that does not release any memory cannot invalidate any stack variables. Therefore, we can avoid adding checks at those calls.

We conservatively gather this information about functions with the following algorithm. We conservatively regard external functions and indirect jumps as possibly calling free. In order to make library function calls precise, we can provide C/C++ library calls

that never call free in our model. This way, we can eliminate some call instructions to instrument. In our running example, we do not need to add a check at the call to g() in Listing 7 because g() does not release any memory.

3.4 Volatile Dropping Optimization

Combined liveness and call graph analysis can help us identify stack variables that may introduce UAF. We then mark these stack variables as volatile for later instrumentation. The volatilization must occur before other passes like `mem2reg` optimize the code during compilation. However, because the each source file is compiled independently, library functions from other files are not available until link time. Therefore, we perform a second pass of call graph analysis at link time and drop unnecessary volatile instructions. This pass is implemented as a link time optimization pass. We explicitly invoke `mem2reg` after volatile instruction dropping to ensure newly optimized stack locations are lifted to registers.

4 EVALUATION

In this section, we evaluate the effectiveness of HeapExpo by analyzing OSS-Fuzz bugs [9, 23], and its performance by benchmarking its time and space overheads compared to previous work.

4.1 Survey of UAF Bugs Discovered by OSS-Fuzz

To estimate how many bugs may be missed by prior work that cannot track pointers through registers, we study use-after-free bugs found in the OSS-Fuzz database [23], and manually categorize the sources of dangling pointers. Our process is similar to debugging where we identify the source of each dangling pointer as a heap pointer, global pointer, local variable pointer, etc.

Reproducing older bugs from OSS-Fuzz is not completely straightforward, as individual projects in OSS-Fuzz are continually updated to their latest versions, and it is nontrivial to identify the correct revision for both the project and its dependencies that will allow the bug to manifest. Once the correct version is found, we can then build the fuzzer component and use the provided test case to reproduce the crash. To identify the correct versions and analyze the bug, we:

- Find a commit of project source that references the bug reported by OSS-Fuzz
- Find corresponding commits of upstream dependencies
- Find the commit of OSS-Fuzz at the time
- Build the executable and libraries with the correct versions of their source code
- Use the OSS-Fuzz test case to reproduce the crash
- Rebuild with debugging flags "-O0 -g"
- Perform dynamic analysis with a debugger

We selected projects with reports of use-after-free bugs from OSS-Fuzz database. We chose projects with an eye toward ease of bug reproduction, based on the number of dependencies and the complexity involved in compiling the fuzzer. Large projects usually have complicated build scripts, making it difficult to turn off optimizations. Without disabling optimization, the source of dangling pointers can be ambiguous. In the end, we were able to reproduce 19 bugs from 11 projects as shown in Table 4.

Source of Dangling Pointer	Occurrence
Global Pointer	1
Heap Pointer	6
Local Variable Pointer	5
Function Argument Pointer	5
Reference of Pointer	1
Transformed Pointer	1

Table 3: Occurrence of UAF Bugs by Sources

```
1  char* ptr;
2  void func1(char* a) {
3      free(a);
4      *a = 0;
5  }
6  void func2(char* a) {
7      *a = 0;
8  }
9  int main() {
10     ptr = malloc(0x10);
11     func1(ptr);
12     func2(ptr);
13 }
```

Listing 8: Category Example

We note that our selection process may bias our corpus toward smaller, simpler projects, and may not be representative of all UAF bugs in the wild. However, we argue that even this relatively small convenience sample indicates the importance of tracking local variables and function arguments.

We categorize the reproduced bugs by the source of dereferenced dangling pointers, manually reviewing reproduced crashes. Referencing the source code with the crash logs, we are able to identify the source of each dangling pointer. The result of our manual review is shown in Table 3. **Reference of pointer** is the case where the function argument is a pointer (reference) to a pointer. If the compiler inlines the function and promotes the referenced pointer, prior work cannot track the pointer. **Transformed pointer** means that before the dangling pointer is used, it is stored as non-pointer data. In the case listed in the Table, the pointer is divided by two before being stored to memory to save one bit of space.

Even if a dereferenced dangling pointer is stored in local variables or function arguments, it is not necessarily the source of the dangling pointer. The underlying reason is that an invalidated dangling pointer can be propagated to these variables. To avoid such mis-categorization, we check whether recent pointer store in function control flow propagates a valid pointer or an invalidated dangling pointer. If the recent pointer store propagates an invalidated pointer, as in Listing 8, the source of the UAF in **func1** is identified as a function argument, while that of **func2** is identified as global variable, because the dangling global variable is propagated to **func2**'s argument.

OSS-Fuzz ID	Project	Source	Promoted
10304	libxml2	Heap Variable	×
17737	libxml2	Heap Variable	×
9975	openvswitch	Reference of Pointer	✓
10796	openvswitch	Local Variable	✓
3569	proj4	Function Argument	✓
3619	proj4	Function Argument	✓
3630	proj4	Function Argument	✓
13878	systemd	Local Variable	✓
13882	systemd	Local Variable	✓
11752	yara	Heap Variable	×
11753	yara	Heap Variable	×
18004	usrsctp	Function Argument	✓
18080	usrsctp	Function Argument	✓
4349	bloaty	Heap Variable	×
10200	libaom	Transformed Pointer (on Heap)	×
5921	wireshark	Global Variable	×
17953	curl	Local Variable	✓
17954	curl	Local Variable	✓
16884	libhevc	Heap Variable	×

Table 4: Summary of Reproduced OSS-Fuzz Bugs

4.2 Effectiveness of UAF Detection

In this section, we test the effectiveness of HeapExpo against use-after-free bugs in a real-word program and our own crafted examples. We tested HeapExpo and DangSan with a QuickJS exploit [5, 24]. The fact that the dangling pointer comes from a local variable makes DangSan incapable of detecting the exploit. On the other hand, HeapExpo is able to track and invalidate the dangling pointer in the local variable and correctly triggers a crash when the invalidated pointer is dereferenced.

Besides manually crafted tests, we also created sample test code by extracting snippets of buggy code from the use-after-free bugs discovered by OSS-Fuzz, which makes it easier to build with DangSan and HeapExpo. This sample code also allow us to easily compare between the two systems. The details can be found in our GitHub repository.

Source	DangSan	HeapExpo
Global Pointer	Yes	Yes
Heap Pointer	Yes	Yes
Volatile Stack Pointer	Yes	Yes
Local Variable Pointer	No	Yes
Function Argument Pointer	No	Yes
Reference of Pointer	Partially	Yes
Transformed Pointer	No	No

Table 5: DangSan vs. HeapExpo Coverage by Type. When promoted by function inlining pass, reference of pointer is not protected by DangSan .

The results from the sample code show that HeapExpo correctly tracks dangling pointers in local variables and function arguments. As shown in Table 3, we observed that the major sources of dangling pointers are heap variables, function arguments and local variables. According to the result in Table 5, HeapExpo is able to track two of the major sources where DangSan fails. Neither DangSan nor HeapExpo can track the transformed pointer case in libaom [18]. We closely studied the pointer reference case, and we believe that it is unlikely to be protected by DangSan, because the function is small and the referenced pointer is a function argument. By contrast, HeapExpo succeeds in this case by preserving the stack location of the function argument.

The exploitability of use-after-free vulnerability usually depends on how attackers could reliably arrange the heap. Limiting the attack within a function's lifespan makes it hard to align dynamic memory, but we have seen that the QuickJS exploit [24] leverages a dangling local variable. Furthermore, we cross checked OSS-Fuzz and CVE database to search for use-after-free bugs caused by dangling **local variables**. Although only a few OSS-Fuzz bugs were also present in the CVE database, we found two such examples, CVE-2019-17534 (OSS-Fuzz id: 16796) and one of the CVE-2018-1000039 bugs (OSS-Fuzz id: 5492), with high risk scores of 8.8 and 7.8 respectively. This demonstrates that the bugs missed by prior systems can indeed be exploitable.

During the course of our evaluation we also found a minor bug in DangSan's pointer packing methods that led to it missing some stack variables. The sample code registers two stack pointer locations, but DangSan fails to register and invalidate the second registered location. By reviewing DangSan's packing code, we found that the packed value is never written back. After we fixed the bug, we are able to correctly track stack variables with DangSan.

4.3 Performance and Memory Overhead

We tested the performance of HeapExpo on CPU-intensive benchmarks, comparing the run time of HeapExpo and DangSan with the common unprotected baseline. We mainly used the SPEC CPU2006 benchmark [14], which was also used to evaluate DangSan. We found several benchmarks that executed correctly under DangSan but not HeapExpo (`gcc`, `perlbench` and `libquantum`). We believe that these failures are caused by bugs in the underlying benchmark programs, as DangSan's authors also reported having to patch some programs in the CPU2006 benchmark suite in order to fix UAFs. We compare the time overhead between HeapExpo and DangSan on the remaining benchmarks.

Our experiments were run on a server with two Intel Xeon X5690 CPUs @3.47GHz. We ran the benchmarks 3 times and report the median. The result of individual time overhead is shown in Figure 2. Although we could not directly benchmark FreeSentry and DangNull because of unpublished optimizations and closed sourced code, we estimated and included the results reported in their papers. Aside from a few missing benchmarks, HeapExpo still performed as well as those conventional approach, let alone the extra coverage of local pointers. We computed the geometric means of the time overheads from DangSan and HeapExpo. Our geometric mean over common benchmarks is 1.66, with 66% overhead from baseline, while DangSan has an overhead of 46%. Our overhead on top of DangSan is therefore 20%. Without propagation intensive programs omnetpp and xalancbmk, HeapExpo has geometric mean of 35% run-time overhead from non-instrumented baseline among 12 benchmarks.

In the experiment, we found runtime overhead is moderately correlated to number of pointer propagations that occurred with a correlation score of $r = 0.61$. The extra pointer propagation from newly tracked sources adds extra overhead. Therefore, the runtime overhead of pointer propagation intense benchmarks increases moderately when there are many new propagations. For example, in xalancbmk, there are 7.2 trillion pointer propagations tracked by HeapExpo versus 2.4 trillion tracked by DangSan. Although this causes the runtime overhead to increase from 2.2x to 3.5x, about three times as many pointer propagations are tracked by HeapExpo.

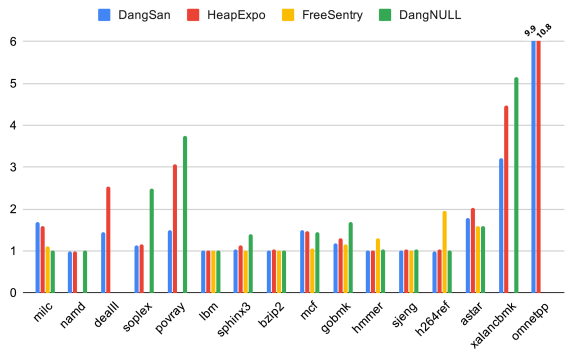

Figure 2: HeapExpo vs. DangSan Run-time Overhead

We also measured the memory overhead of HeapExpo and DangSan under SPEC CPU2006 benchmarks. As shown in Figure 3, the memory overhead of HeapExpo is not significantly different from DangSan. We did not change the underlying data structures of DangSan, but we did increase the number of source type to track, so a small increase in memory overhead is expected. Due to the drastically increased pointer propagation behavior in xalancbmk benchmark, the memory overhead is increased as well. Among the benchmarks shared by all approaches, HeapExpo has a geometric mean of 100%, while DangSan has 87% and DangNull has 137% overhead. Although it requires slightly more memory than DangSan, HeapExpo consumes less memory than DangNull, which offers less protection.

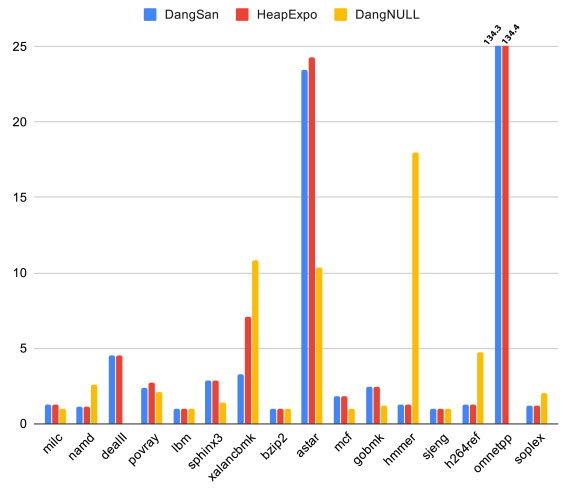

Figure 3: HeapExpo vs. DangSan Memory Overhead

4.4 Impact of Optimization

Here, we show how our liveness analysis and call graph analysis affect the performance of HeapExpo. In Table 6, we compare the number of instrumented instructions among different builds. "HeapExpo (unopt)" has no static analysis optimization, and marks every stack pointer as volatile. The "HeapExpo" build has static analysis turned on and only marks stack pointers that may result in UAF bugs as volatile. Finally, the "DangSan" build does not mark any stack pointers volatile. Table 6 indicates that HeapExpo eliminates about half the pointers that need instrumentation in the SPEC CPU2006 benchmarks compared to our base build, but still has about 1–2 times more instrumentation than DangSan as a result of our improved coverage. Figure 4 demonstrates that our optimizations greatly lower the amount of instrumentation and thus reduce runtime overhead.

According to the number of instructions, we can also see that stack pointers like local variables and function arguments make up a large portion of pointer propagations overall. Comparing the number of instrumented instructions between unoptimized HeapExpo build and the DangSan build, we argue that there are about 4 times more pointer propagation instructions present in the program than those tracked by DangSan.

Benchmark	HeapExpo (unopt)	HeapExpo	HeapExpo LTO	DangSan
perlbench	20224	15193	15193	7698
bzip2	237	107	72	35
mcf	259	173	145	130
gobmk	1795	1134	779	278
hmmer	3255	2433	1250	498
sjeng	118	101	47	14
h264ref	1916	950	669	406
omnetpp	23794	8749	7886	5673
astar	527	151	136	90
xalancbmk	175707	35666	27229	20338

Table 6: Number of Instrumented Instructions among Different Builds

These local variables and function arguments are not trackable with any prior dangling pointer invalidation system. Promoted to registers, these pointers are pushed to the stack during function calls and popped when the function returns. Since no prior work tracks pushed stack locations, exploits that make use of these dangling pointers could evade detection.

The straightforward solution is to track all pointer variables. However, that would incur 413% more instrumentation, and make the overall overhead unacceptable. After applying our liveness and call graph analyses, we reduce the instrumentation overhead to 156%. This overhead is acceptable, but we can still do better by removing further unnecessary instrumentation using the call graph gathered during link time optimization. The number of instrumented instructions is consequently lowered to 86% more than DangSan, and we still guarantee the safety of all pointer variables under the thread-local assumption.

5 LIMITATIONS

The design of our work follows earlier research including FreeSentry, DangNull and DangSan. For this reason, our work shares their common limitations. We discuss these limitations in this section.

Our system and other previous works are transformation passes based on LLVM tool chain, which means that source code is required. Recalling other approaches in literature, source code is often required to perform meaningful sanitizing with reasonable overhead. After compiler and linker optimizations, binary code is very different from its source, and meaningful information such as variable types is lost during the process. Although it is possible to retrieve a portion of the information through static analysis of binary code, the information would not be helpful enough for protection because of the lack of completeness. We agree with prior work that the limitation of requiring source code is an appropriate trade-off.

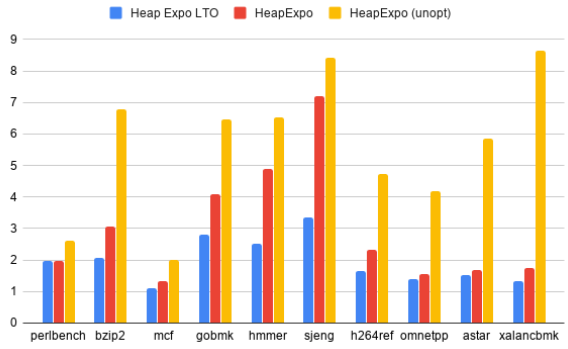

Figure 4: Number of Instrumented Instructions Compared to DangSan

Like most sanitizers, HeapExpo can only detect temporal safety violations with instrumented code. That means the library source code is also required to detect use-after-free bugs in libraries. The linked libraries need to be compiled by the sanitizer as well. In practice, this is implemented in OSS-Fuzz where all library code as well as project code is compiled with the sanitizer flags.

We share another limitation with DangSan, where we do not track pointers that are copied in type-unsafe ways. For example, **memcpy** and **memmove** in glibc are not tracked because instrumenting them can result in non-trivial overhead. Tracking these functions is achievable with some overhead by using an implementation that also records outwards pointers. Although our prototype does not implement this, we think the type information could be recovered more easily with a pass closer to the front-end where all type information remains.

We also note that the fact that LLVM often optimizes the **memcpy** function to trackable instructions like **store** makes this false negative less significant. In Listing 9, LLVM interprets line 11 as a **memcpy** call which copies data from a local to a global, so global.data is not registered at line 11. Thus, when the memory is released at line 15, global.data is not invalidated by our instrumentation. The use of global.data at line 16 does not raise an alert. However, LLVM will transform the **memcpy** call to two simpler instructions shown in lines 13–14, where line 13 is instrumented with a call to regptr in line 12. This occurs because we have a relatively small structure, so the cost for calling **memcpy** exceeds the two simple instructions. The idea is that one can customize the **memcpy** optimization to recover more pointer propagation. We did not implement this because of the limited number of bugs in this case, as von der Kouwe et al. discuss [26].

```
1  struct str {
2      char* data;
3      int len;
4  };
5  struct str global;
6  int main () {
7      struct str local;
8      local.data = malloc(10);
9      strcpy(local.data, "string");
10     local.len = strlen(local.data);
11     global = local;
12     //regptr(&global.data,local.data);
13     //global.data = local.data
14     //global.len = local.len
15     free(local.data);
16     printf("%s\n", global.data);
17 }
```

Listing 9: False negative example

Finally like all current dangling pointer invalidation approaches, our work does not support custom pointer manipulation. Comparison between a regular pointer with an invalidated pointer is broken, but that between invalidated pointers is supported. Custom storage for pointers such as compacting pointers to smaller sizes is not trackable because type information is lost during the transformation. However, we believe this kind of direct pointer manipulation is relatively rare: we only see one such case (in libaom [18]) among our reproduced bugs from OSS-Fuzz.

6 RELATED WORK

In this section, we discuss how our work relates to prior work in the literature. HeapExpo is an improvement on the dangling pointer invalidation approach for detecting use-after-free vulnerabilities. Although that is not the only approach, its combination of low overhead and strong protection compared with other methods makes it attractive.

Dangling Pointer Invalidation. The previous works using this approach are DangNull [16], FreeSentry [28], DangSan [26] and pSweeper [13]. Based on the LLVM toolchain, they all instrument pointer propagation instructions at the IR level. The earliest works, DangNull and FreeSentry, are conceptually similar but implemented with different data structures. They both instrument `store` instructions in the LLVM IR as an LLVM optimization pass. DangNull keeps a more complete data structure which records both inward and outward pointers, but it only tracks pointers in global data and on the heap. FreeSentry only registers inward pointers of an object, but it extends dangling pointer detection to pointers stored on the stack as well. Finally, DangSan extends FreeSentry by improving the data structures used and supporting multi-threading; DangSan uses a log-like data structure inspired by log-structured filesystems. They also support multithreading by managing thread-local data structures. In terms of overhead, DangSan is currently the state of the art.

These three works all manage pointer propagation inline, while pSweeper dedicates an extra thread to to manage the pointer metadata structures. It bookmarks the location of pointers and keeps a list of pointers as a simple data structure. Concurrently, it runs a separate thread that cycles through the bookmarked location to register pointers and to clean out dangling ones. The use of extra threads leverages idle CPU cycles in multicore systems to speed up dangling pointer invalidation. However, it uses more computing power overall due to extra worker threads.

The instrumentation strategy has remained the same across the four previous works, so none of them are able to track stack pointers promoted to registers. We studied a number of bugs in OSS-Fuzz database and found local variables and function arguments are the sources of more than half of the use-after-free bugs. Previous works cannot deal with such cases, so we target these false negatives in this paper.

Secure allocators. To ensure temporal safety, some prior work focuses on creating a more sophisticated allocator that can detects access to freed memory. Dhurjati and Adve [11] put each allocation on a different virtual page, so that reads and writes to released pages raise an error. DieHarder [22] marks freed chunks of memory and suppresses the reuse of those memory blocks so that dereferencing dangling pointers is likely to cause an exception. Cling [3] is an allocator that introduces type-safe memory reuse that prevents corruption of control addresses by inferring object types from the allocation stack trace, so that dangling pointers will end up pointing to the same type of object even if the allocation is reused.

Address-based checking. In this approach, metadata of pointers and memory addresses are stored and checked at dereference time. Previous projects in the category include CETS [12] and SafeC [4]. CETS is another compiler-based use-after-free detection. It also supports checking for local variables. CETS associates pointers with the root object by pointing them to a key that indicates the validity of the memory object. When memory objects are released, these keys are invalidated. Access of the dangling pointer would raise an alert during a key check. When combined with SoftBound [19], it can achieve both spatial and temporal safety. As von der Kouwe et al. [26] discussed, however, this approach has more run-time overhead and compatibility issues than dangling pointer invalidation approach. Lee et al. [16] also pointed that it has high false positive rate, raising false alarms in 5 of 16 tested programs.

Safe C [4] is source-level instrumentation that adds extra metadata to raw pointers, including the actual raw pointer, size, and the memory section it points to. It is able to perform bounds checking and check temporal safety using these attributes. However, source-level instrumentation can reduce the opportunities the compiler has to optimize the program, and has a significant time overhead.

Debugging tools for use-after-free. The most widely used tools for detecting memory errors are Valgrind [21] and Address Sanitizer [25]. They also detect at the time of pointer dereference. They are generally comprehensive, but come with high performance overhead. Valgrind is built on a dynamic binary translation (DBT) framework. It translates and instruments binary code one block at a time. However, at the machine language level, the type of a memory location or register is ambiguous. Although we know a referenced register value has to be a pointer, pointers may also be used in arithmetic. Thus, pointer checking can often require a

search through all of program memory, which is inefficient. Another possible approach using DBT would be to use taint analysis to track all the pointers. However, we argue that this approach has an extra overhead from taint analysis, and is unsuitable for runtime protection.

Address Sanitizer [25] is a very effective tool for detecting memory-related errors. The essence of Address Sanitizer is using shadow memory to mark the validity of every address. It postpones the reuse of freed memory, so that use of a dangling pointer produces access to memory marked invalid. Address Sanitizer overloads the allocator functions and adds padding to memory blocks so that access to the padding also raises alerts. Address Sanitizer shows the effectiveness of shadow memory, but it cannot prevent sophisticated attacks which force the reuse of memory. Moreover, the run-time and memory overhead make it unsuitable for online protection use. Hardware Assisted Address Sanitizer hwasan is an optimization for memory usage on AArch64 and SPARC architectures. With a low false-negative rate (99.61%), it probabilistically detects UAF. Furthermore, its optimization makes dynamic memory layout easier to predict, and thus the system is more exploitable if probabilistic checking misses a dereference.

Garbage collection. MarkUs [2] introduces many optimization techniques on top of garbage collection to tailor it for use-after-free mitigation. During garbage sweeping, it scans memory for potential pointers. Dynamic memory blocks are freed only when there is no potential pointer referencing them. It is incapable of detecting use-after-free by design and works only as a mitigation technique. Unable to distinguish pointer types from raw data, conservative garbage collection adds extra attack vector for memory exhaustion where an attacker can put random heap addresses in controlled buffers. Li and Tang [17] launched such attack against MemGC [1, 27] to exploit Microsoft Edge.

Dangling pointer detection. Undangle [7] focuses on dangling pointer detection that may result in use-after-free vulnerabilities. Undangle works offline by processing the execution trace and allocation log of the program. It applies taint analysis to track pointer propagation during execution. Taint analysis significantly increases the time overhead because every instruction must be instrumented to propagate taint information, meaning that this approach can only be used offline. We can provide similar functionality online by recording and reporting invalidated pointers.

7 CONCLUSION

In conclusion, we improve on prior work on temporal safety analysis. Focusing on dangling pointer invalidation, we support tracking of a broader range of pointers including local variables and function arguments. Through a review of 19 real-world UAF bugs from the OSS-Fuzz database, we found that 10 are ultimately caused by dangling pointers stored in local variables and function arguments, indicating that existing systems have significant gaps in coverage.

To close this gap, we introduced a novel approach that forces such pointers to be stored on the stack, allowing them to be tracked. However, tracking all such pointer variables and arguments can introduce unacceptably high overheads, so we applied a static analysis that identifies pointers that can never be involved in use-after-free bugs and excludes them from instrumentation.

HeapExpo successfully closes an important gap in detection of dangling pointers in C/C++ programs, and does so with reasonable additional overhead compared to prior work.

To aid in future research, we choose to open source our prototype HeapExpo at https://github.com/messlabnyu/heap-expo.

ACKNOWLEDGMENTS

This research was supported in part by National Science Foundation (NSF) Award 1657199. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the NSF.

REFERENCES

[1] 2016. Triaging the exploitability of IE/EDGE crashes. https://msrc-blog.microsoft.com/2016/01/12/triaging-the-exploitability-of-ieedge-crashes/
[2] S Ainsworth and TM Jones. [n.d.]. MarkUs: Drop-in Use-After-Free Prevention for Low-Level Languages. In *2020 IEEE Symposium on Security and Privacy (SP)*. 860–860.
[3] Periklis Akritidis. 2010. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *Proceedings of the 19th USENIX Conference on Security* (Washington, DC) *(USENIX Security'10)*. USENIX Association, Berkeley, CA, USA, 12–12. http://dl.acm.org/citation.cfm?id=1929820.1929836
[4] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. 1994. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (Orlando, Florida, USA) *(PLDI '94)*. ACM, New York, NY, USA, 290–301. https://doi.org/10.1145/178243.178446
[5] Fabrice Bellard. 2019. QuickJS Javascript Engine. https://bellard.org/quickjs/.
[6] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. *SIGPLAN Not.* 41, 6 (June 2006), 158–168. https://doi.org/10.1145/1133255.1134000
[7] 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* (Minneapolis, MN, USA) *(ISSTA 2012)*. ACM, New York, NY, USA, 133–143. https://doi.org/10.1145/2338965.2336769
[8] CVE. 2019. CVE - Common Vulnerabilities and Exposures. https://cve.mitre.org/.
[9] Google developers. 2019. https://github.com/google/oss-fuzz.
[10] LLVM developers. 2019. https://llvm.org/docs/Passes.html#mem2reg-promote-memory-to-register.
[11] Dinakar Dhurjati and Vikram Adve. 2006. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks*. IEEE Computer Society, Washington, DC, USA, 269–280. https://doi.org/10.1109/DSN.2006.31
[12] Dinakar Dhurjati and Vikram Adve. 2006. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks*. IEEE Computer Society, Washington, DC, USA, 269–280. https://doi.org/10.1109/DSN.2006.31
[13] Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. 2019. Position Paper: Progressive Memory Safety for WebAssembly. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 4.
[14] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737
[15] 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* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673
[16] 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*.
[17] Henry Li and Jack Tang. 2017. Cross the Wall - Bypass All Modern Mitigations of Microsoft Edge. https://www.blackhat.com/asia-17/briefings.html#cross-the-wall-bypass-all-modern-mitigations-of-microsoft-edge
[18] libaom developers. 2019. https://github.com/mozilla/aom.
[19] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. *SIGPLAN Not.* 44, 6 (June 2009), 245–258. https://doi.org/10.1145/1543135.1542504
[20] Nicholas Nethercote and Julian Seward. 2007. How to Shadow Every Byte of Memory Used by a Program. In *Proceedings of the 3rd International Conference*

on Virtual Execution Environments (San Diego, California, USA) *(VEE '07)*. ACM, New York, NY, USA, 65–74. https://doi.org/10.1145/1254810.1254820

[21] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. *SIGPLAN Not.* 42, 6 (June 2007), 89–100. https://doi.org/10.1145/1273442.1250746

[22] Gene Novark and Emery D. Berger. 2010. DieHarder: Securing the Heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '10)*. ACM, New York, NY, USA, 573–584. https://doi.org/10.1145/1866307.1866371

[23] OSS-Fuzz 2019. OSS-Fuzz: Fuzzing the planet. https://bugs.chromium.org/p/oss-fuzz.

[24] QuickJS 0day Contest 2019. QuickJS 0day Contest. http://rce.party/cracksbykim-quickJS.nfo.

[25] 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* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 28–28. http://dl.acm.org/citation.cfm?id=2342821.2342849

[26] 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* (Belgrade, Serbia) *(EuroSys '17)*. ACM, New York, NY, USA, 405–419. https://doi.org/10.1145/3064176.3064211

[27] Mark Yason. 2015. MemGC: Use-After-Free Exploit Mitigation in Edge and IE on Windows 10. https://securityintelligence.com/memgc-use-after-free-exploit-mitigation-in-edge-and-ie-on-windows-10/

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