Preventing Use-after-free with Dangling Pointers Nullification

Byoungyoung Lee† Chengyu Song† Yeongjin Jang† Tielei Wang†

Taesoo Kim† Long Lu* Wenke Lee†

†{blee, csong84, yeongjin.jang, tielei, taesoo}@gatech.edu, wenke@cc.gatech.edu
School of Computer Science,
Georgia Institute of Technology

*long@cs.stonybrook.edu
Department of Computer Science,
Stony Brook University

***Abstract*—Many system components and network applications are written in languages that are prone to memory corruption vulnerabilities. There have been countless cases where simple mistakes by developers resulted in memory corruption vulnerabilities and consequently security exploits. While there have been tremendous research efforts to mitigate these vulnerabilities, use-after-free still remains one of the most critical and popular attack vectors because existing proposals have not adequately addressed the challenging program analysis and runtime performance issues.**

In this paper we present DANGNULL, a system that detects temporal memory safety violations—in particular, use-after-free and double-free—during runtime. DANGNULL relies on the key observation that the root cause of these violations is that pointers are not nullified after the target object is freed. Based on this observation, DANGNULL automatically traces the object's relationships via pointers and automatically nullifies all pointers when the target object is freed. DANGNULL offers several benefits. First, DANGNULL addresses the root cause of temporal memory safety violations. It does not rely on the side effects of violations, which can vary and may be masked by attacks. Thus, DANGNULL is effective against even the most sophisticated exploitation techniques. Second, DANGNULL checks object relationship information using runtime object range analysis on pointers, and thus is able to keep track of pointer semantics more robustly even in complex and large scale software. Lastly, DANGNULL does not require numerous explicit sanity checks on memory accesses because it can detect a violation with implicit exception handling, and thus its detection capabilities only incur moderate performance overhead.

I. INTRODUCTION

Many system components and network applications are written in the unsafe C/C++ languages that are prone to memory corruption vulnerabilities. To address this problem, a large number of techniques have been developed to improve memory safety and prevent memory corruption bugs from being exploited [2, 4, 12, 19, 30–33, 38, 48]. However, the problem of detecting and preventing use-after-free bugs remains unsolved. Among the CVE identifiers of the Chromium browser that we collected from Oct. 2011 to Oct. 2013 in Table I, use-after-free vulnerabilities are not only 40x/3x more than stack and heap overflows in quantity, but also have more severe security impacts than traditional vulnerabilities: 88% of use-after-free bugs are rated critical or high in severity, while only 51% of heap overflows are considered as high severity. Not only are there many use-after-free vulnerabilities, they have also become a significant attack vector. In Pwn2Own 2014 [21], an annual contest among hackers and security research groups, the VUPEN team was awarded with the largest cash amount, $100,000, for a single use-after-free exploit that affects all major WebKit-based browsers.

Severity	Use-after-free	Stack overflow	Heap overflow	Others
Critical	13	0	0	0
High	582	12	107	11
Medium	80	5	98	12
Low	5	0	3	1
Total	680	17	208	24

Table I: The number of security vulnerabilities in the Chromium browser for two years (2011–2013), classified by types of vulnerabilities and their severity.

Compared with many other vulnerability types, including stack buffer overflows or heap buffer overflows, use-after-free is generally known as one of the most difficult vulnerability type to identify using static analysis. In modern C/C++ applications (especially under object-oriented or event-driven designs), the resource free (i.e., memory deallocation) and use (i.e., memory dereference) are well separated and heavily complicated. Statically identifying use-after-free vulnerabilities under this difficult conditions involves solving challenging static analysis problems (e.g., inter-procedural and point-to analysis while also considering multi-threading effects), and is therefore feasible only for small size programs [13, 34].

NDSS '15, 8-11 February 2015, San Diego, CA, USA
 ISBN 1-891562-38-X
http://dx.doi.org/10.14722/ndss.2015.23238

Category	Protection Technique	Explicit Checks	Liveness Support	False positive Rates	Bypassable	Bypassing Exploit Technique
Use-after-free detectors	DANGNULL	No	Yes	Low	No	N/A
	CETS [30]	Yes	No	High	No	N/A
	Undangle [6]	Yes	No	Low	No	N/A
	Xu et al. [48]	Yes	No	Low	No	N/A
Memory error detectors	AddressSanitizer [38]	Yes	No	Low	Yes	Heap manipulation
	Memcheck [32]	Yes	No	High	No	N/A
	Purify [19]	Yes	No	High	No	N/A
Control Flow Integrity	CCFIR [50]	Yes	No	Low	Yes	Abusing coarse grained CFI
	bin-CFI [51]	Yes	No	Low	Yes	or
	SafeDispatch [22]	Yes	No	Low	Yes	corrupting non-function pointers
Safe memory allocators	Cling [2]	No	No	Low	Yes	Heap manipulation
	DieHarder [33]	No	No	Low	Yes	Heap manipulation

Table II: Comparing the proposed system DANGNULL with other protection techniques detecting use-after-free. The `Explicit checks` column represents whether the technique explicitly instruments checks to detect use-after-free except via pointer propagation. The `Liveness support` column represents whether the technique may continuously run an instrumented application as if use-after-free vulnerabilities are patched. The `False positive rates` column represents whether the technique would generate the high/low number of false alarms on benign inputs, and high false positive rates imply that it would be difficult to be deployed for large scale software. The `Bypassable` column shows whether the protection technique can be bypassable with the following column's exploitation technique. Overall, while all other protection techniques show either high false positive rates or being bypassable, DANGNULL achieves both low false positive rates and being non-bypassable against sophisticated exploitation techniques. §VII describes more details on each protection technique.

Most research efforts to detect use-after-free vulnerabilities are relying on either additional runtime checks or dynamic analysis (listed in Table II). For instance, use-after-free detectors including [30, 48] have been proposed to address dangling pointer issues. By maintaining metadata for each pointer and tracking precise pointer semantics (i.e., which pointer points to which memory region), these tools can identify dangling pointers or prevent memory accesses through dangling pointers. However, precisely tracking runtime semantics on a per-pointer bases is non-trivial as there would be a huge number of pointers and their metadata in runtime, which may result in high false positive rates (i.e., identifying benign program behavior as use-after-free) or significant performance degradation. Such shortcomings would limit the potential for these techniques to be deployed for large scale software.

Memory error detectors [19, 32, 38] are also able to capture use-after-free bugs during the software testing phase. By maintaining the allocated/freed status of memory, these tools can prevent accesses to freed memory. However, these tools are not suitable for detecting real-world exploits against use-after-free vulnerabilities if attackers can partially control the heap memory allocation process, especially for web browsers. For example, by using Heap Spraying [10, 36] or Heap Fengshui [39] like techniques, attackers can force the target program to reuse certain freed memory.

In addition, Control Flow Integrity (CFI) tools can be used to prevent use-after-free vulnerabilities from being exploited to hijack the control-flow because the majority of vulnerability exploitations hijack the control flow to execute malicious code with Return-Oriented Programming (ROP) [35]. However, due to the inherent limitations of these tools, most of them only enforce coarse-grained CFI, which leaves some control-flows exploitable [7, 11, 15, 16]. Moreover, since control-flow hijacks are not the only method to compromise a program, it is still possible to bypass these techniques even if they can enforce perfect CFI, e.g., via non-control data attacks [8, 27].

Overall, all of the previous protection techniques show either high false positive rates or are bypassable using certain exploitation techniques. In other words, there is currently no use-after-free mitigation solution that works well for large scale software and can also stop all known forms of use-after-free exploitation techniques.

In this paper, we present DANGNULL, a system that prevents temporal memory safety violations (i.e., use-after-free and double-free) at runtime. As suggested by many secure programming books [37], a pointer should be set to `NULL` after the target object is freed. Motivated by the fact that dangling pointers obviously violate this rule, DANGNULL automatically traces the object relationships and nullifies their pointers when the object they pointed to is freed. In particular, rather than relying on a heavy dynamic taint analysis, DANGNULL incorporates a runtime object range analysis on pointers to efficiently keep track of both pointer semantics and object relationships. Based on the collected object relationship information, DANGNULL nullifies dangling pointers when the target memory is freed. After this nullification, any temporal memory safety violation (i.e., dereferencing the dangling pointers) turns into a null-dereference that can be safely contained.

This unique design choice of DANGNULL offers several benefits. First, since nullification immediately eliminates any possible negative security impacts at the moment dangling pointers are created, DANGNULL does not rely on the side effects from use-after-free or double-free, and thus cannot be bypassed by sophisticated exploit techniques. Second, a runtime object range analysis on pointers allows DANGNULL to efficiently keep track of pointer semantics. Instead of tracing complicated full pointer semantics, DANGNULL only tracks

abstracted pointer semantics sufficient to identify dangling pointers with the understandings on runtime object ranges. This allows DANGNULL to overcome the difficulty of pointer semantic tracking and scale even for complex and large software. Third, DANGNULL does not require any explicit sanity checks on memory accesses, which are a common performance bottleneck in other mitigation tools. Instead, it relies on implicit null-dereference exceptions, which are safely contained by DANGNULL. Lastly, DANGNULL can continue running correctly even after use-after-free exploits (i.e., as if a program is patched for the exploiting vulnerability) in some cases. Since nullified dangling pointers have identical semantics with existing null-pointer checks in programs, it can utilize such existing checks and survive use-after-free exploits.

We implemented DANGNULL and applied it to SPEC CPU 2006 and Chromium, and evaluated its effectiveness and performance overhead. In particular, for the Chromium browser, all seven real-world use-after-free exploits we tested were safely prevented with DANGNULL. Moreover, DANGNULL only imposed an average of 4.8% increase in overhead in JavaScript benchmarks and an average of 53.1% increase in overhead in rendering benchmarks, while a page loading time on Alexa top 100 websites showed a 7.4% slowdown on average. DANGNULL also showed no false positives while running more than 30,000 benign test cases (including both unit tests and end-to-end testing web pages).

To summarize, our contributions are as follows:

- We proposed DANGNULL, a system that detects temporal memory safety violations, and is resilient to sophisticated bypassing techniques.
- We implemented DANGNULL and applied it to the Chromium browser, a well known complex and large scale software.
- We thoroughly evaluated various aspects of DANGNULL: attack mitigation accuracy, runtime performance overheads, and compatibility.

The rest of this paper is organized as follows. §II describes general problems and challenges of dangling pointers. §III presents our mitigation solution DANGNULL for dangling pointers. §IV describes how DANGNULL is implemented, and §V evaluates various aspects of DANGNULL. §VI discusses implications and limitations of DANGNULL, and §VIII concludes the paper.

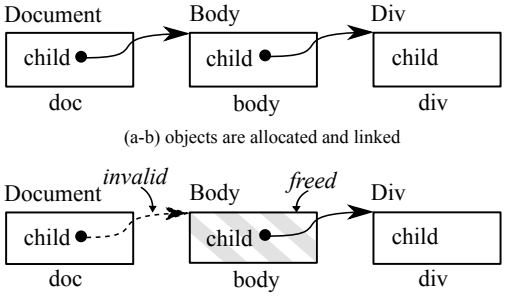
II. BACKGROUND

From dangling pointers to use-after-free. Dangling pointers refer to pointers that point to freed memory, and lead to memory safety errors when accessed. To be precise, a dangling pointer itself does not cause any memory safety problem, but accessing memory through a dangling pointer can lead to unsafe program behaviors and even security compromises, such as control-flow hijacking or information leakage.

For example, as illustrated in Example 1, `body` and `doc→child` pointers become dangling pointers after `body` is deleted; the object `body` is allocated, assigned to `doc→child`, and freed after the propagation. Since `doc→child` now points to the invalid memory region (which is freed or already reused by the memory allocator), if the memory is accessed by `doc→child`, a use-after-free error occurs. Unfortunately, use-after-free errors often lead to security exploits. For example, if an adversary can manipulate (directly or probabilistically) the memory region, he can then use the pointer to obtain sensitive data or even change the program's control flows.

```
1  class Div: Element;
2  class Body: Element;
3  class Document {
4    Element* child;
5  };
6
7  // (a) memory allocations
8  Document *doc = new Document();
9  Body *body = new Body();
10 Div *div = new Div();
11
12 // (b) using memory: propagating pointers
13 doc->child = body;
14 body->child = div;
15
16 // (c) memory free: doc->child is now dangled
17 delete body;
18
19 // (d) use-after-free: dereference the dangled pointer
20 if (doc->child)
21     doc->child->getAlign();
```

(i) Vulnerable code



(a-b) objects are allocated and linked

(c-d) body is freed (so dangled), and doc reads the invliad memory

(ii) Objects and their relations in the course of running (i)

Example 1: A running example of a use-after-free vulnerability. Document (`doc`), Body (`body`), and Div (`div`) are allocated and referencing each other. After `body` is freed by `delete`, `body` becomes a dangling pointer, pointing to an invalid memory region. Attempting to access `body` via `child` will lead to unsafe runtime behaviors.

Challenges in identifying dangling pointers. The example we described above is very simple. In practice, however, real-world use-after-free vulnerabilities can be complicated. First, the allocation, propagation, free, and dereference operations could all be located in separate functions and modules. Second, at runtime, the execution of these operations could occur in different threads. Especially for applications with event-driven designs and object-oriented programming paradigms in mind, the situation becomes even worse. For example, web browsers need to handle various events from JavaScript or DOM, UI applications need to handle user generated events, and server side applications implement event-loops to handle massive client requests. Given the complicated and disconnected component relationships, developers are prone to make mistakes when implementing object pointer operations and thus leave the door open for dangling pointer problems.

In addition, not all dangling pointers violate temporal memory safety. In our experiments (§V), we found numerous benign dangling pointers in practice (e.g., there were between 7,000 and 32,000 benign dangling pointers while visiting benign web pages in the Chromium browser). Thus, in order to clarify dangling pointer problems, we first define the following notions:

Definition 1. Dangling pointer. *A pointer variable p is a dangling pointer if and only if*

$$(x := allocate(size)) \wedge p \in [x, x + size - 1] \vdash deallocate(x)$$

for any pointer variables x and p, and a value size.

This definition simply captures the fact that a dangling pointer points to a freed memory area; if the variable p points to the freed memory area ($[x, x + size - 1]$), p is a dangling pointer. Without loss of generality, we assume that `allocate()` and `deallocate()` functions denote all memory allocation and deallocation functions, respectively. Based on this definition, we further define unsafe and benign dangling pointers.

Definition 2. Unsafe dangling pointer. *A pointer variable p is an unsafe dangling pointer if p is a dangling pointer and there exists memory read or write dereferences on p*

Definition 3. Benign dangling pointer. *A pointer variable p is a benign dangling pointer if p is a dangling pointer but not an unsafe dangling pointer.*

Note that only unsafe dangling pointers violate temporal memory safety and should thus be prevented or detected. On the other hand, an alarm on a benign dangling pointer is considered a false alarm (i.e., false positive).

Exploiting dangling pointers. In order to abuse unsafe dangling pointers for security compromises (e.g., to achieve control flow hijacking or information leak), an attacker needs to place useful data in the freed memory region where an unsafe dangling pointer points. Depending on how unsafe dangling pointers are subsequently used, different exploitation techniques are employed. For example, attackers can place a crafted virtual function table pointer in the freed region; and when a virtual function in the table is invoked later (i.e., memory read dereference on unsafe dangling pointers), a control flow hijacking attack is accomplished. As another example, if the attacker places a root-privileged flag for checking the access rights in the freed region, a privilege escalation attack is accomplished. Moreover, if the attacker places corrupted string length metadata in the freed region and the corresponding string is retrieved, an information leakage attack is accomplished.

It should be clear now that the exploitability of unsafe dangling pointers depends on whether an attacker can place crafted data where a dangling pointer points. Specifically, an attacker needs to place useful objects where the freed memory region is located (e.g., an extra memory allocation is one popular exploitation technique). This implies that the extra operations controlled by an attacker should be performed between free and use operations (e.g., between line 17 and 21 in Example 1) because it is the only time window that the freed memory region can be overwritten. In this sense, security professionals determine the exploitability of use-after-free or double-free based on whether an attacker can gain control between the free and use operations. For example, the Chromium security team uses this notion to determine the amount of bug bounty rewards [42].

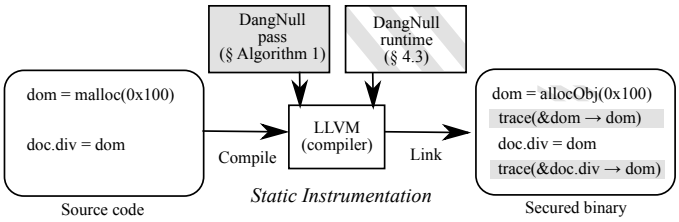


Figure 1: Overview of DANGNULL's design. DANGNULL consists of two main components, a static instrumentation (§III-B) and a runtime library (§III-C). At the static instrumentation stage, DANGNULL identifies propagation (their dependencies) of object pointers and inserts a call to the tracing routine to keep track of point-to relations between pointers and memory objects. At runtime, DANGNULL interposes all memory allocations to maintain the data structure for live objects (§III-C1), and all memory frees to nullify all the pointers pointing to the object that is about to be freed (§III-C2).

III. DESIGN

DANGNULL aims to detect unsafe dangling pointers without false positives and false negatives in practice, so that it cannot be bypassed with sophisticated exploitation techniques while supporting large scale software. To achieve this goal, DANGNULL addresses the root cause of the unsafe dangling pointer problem. As discussed in §II, the root cause of unsafe dangling pointers is that pointers—including both 1) the pointer that an allocated memory object is initially assigned to and 2) all pointers that are propagated from the initial pointer through direct copy and pointer arithmetic —are not nullified after the target memory is freed.

Based on this observation, DANGNULL is designed to 1) automatically trace the point-to relations between pointers and memory objects, and 2) nullify all pointers that point to a freed memory object. By nullifying pointers that would otherwise have become unsafe dangling pointers, DANGNULL not only prevents reads and writes of the freed memory, which may contain security-sensitive meta-data (e.g, function pointers or vector length variables), but also, in many cases, shepherds the execution of applications as if the detected use-after-free or double-free bug had already been patched.

A. System Overview

An overview of DANGNULL's design is illustrated in Figure 1. To generate a binary secured against use-after-free vulnerabilities, developers should compile the source code of the target program with DANGNULL. Given the source code, DANGNULL first identifies instructions that involve pointer assignments and then inserts a call to the tracing routine (a static instrumentation in §III-B). At runtime, with the help of instrumented trace instructions, DANGNULL keeps track of point-to relations in a thread-safe red-black tree, `shadowObjTree`, that efficiently maintains shadow objects representing the state of corresponding memory objects (a runtime library in §III-C).

```
1  for function in Program:
2     # All store instructions are
3     # in the LLVM IR form of 'lhs := rhs'.
4     for storeInstr in function.allStoreInstructions:
5        lhs = storeInstr.lhs
6        rhs = storeInstr.rhs
7
8        # Only insterested in a pointer on the heap.
9        if mustStackVar(lhs):
10          continue
11       if not isPointerType(rhs):
12          continue
13
14       new = createCallInstr(trace, lhs, rhs)
15       storeInstr.appendInstr(new)
```

Algorithm 1: The algorithm for static instrumentation. For every store instruction where the destination may stay on the heap, DANGNULL inserts `trace()` to keep track of the relation between the pointer and the object it points to.

On every memory allocation, DANGNULL initializes a shadow object for the target object being created. Upon freeing an object, DANGNULL retrieves all pointers that point to this object (from `shadowObjTree`) and nullifies those pointers, to prevent potential use-after-free or double-free.

Later in this section, we describe each component of DANGNULL (the static instrumentation and the runtime library), and explain how we maintain `shadowObjTree` with a concrete running example (Example 1).

B. Static Instrumentation

The static instrumentation of DANGNULL is done at the LLVM IR [24] level and is designed to achieve one primary goal: to monitor pointer assignments to maintain the point-to relations. To balance security, performance, and engineering efforts, only appropriate pointers are instrumented. More specifically, DANGNULL only tracks pointers located on the heap (e.g., `doc→child` in Example 1) but not on the stack (e.g., `doc` in Example 1). From our preliminary experiment on the Chromium browser, we found that stack-located pointers are unlikely to be exploitable, even though are many dangling pointers. This is because stack-located pointers tend to have a very short lifetime since the scope of stack variables are bounded by the scope of a function and accesses to those variables are limited in the programming language. Heap-located pointers generally have much a longer lifetime (i.e., the number of instructions between free and use is larger). In other words, unsafe dangling pointers located in the heap offer better controls between the free and dereference operations, and are thus are more likely to be exploited (§II). Therefore, to reduce performance overhead and keep our engineering efforts effective and moderate, we focus on heap-located pointers. Note that the nullification idea of DANGNULL has no dependencies on the pointer locations, and is generally applicable to both heap- and stack-located pointers.

The algorithm for the static instrumentation is described in Algorithm 1. At lines 1-4, all store instructions [1] in each function are iterated. With the pointer information obtained at lines 5-6, DANGNULL first opts out if `lhs` is a stack variable, using a intra-procedure backward data-flow analysis (line 9-10). Specifically, given a `lhs` variable, we leveraged a def-use chain provided by LLVM to see if this variable is allocated on the stack via the allocation statement. Since this analysis is conservative, it is possible that DANGNULL still instruments some pointer assignments in the stack. However, as DANGNULL does not instrument allocations and deallocations of stack variables, such assignments will be ignored by the runtime library. Next, DANGNULL performs a simple type signature check to see if `rhs` is not of a pointer type (line 11-12) [2]. With these two opt-out checks, DANGNULL ignores all uninteresting cases as the current version of DANGNULL only targets the heap located pointers. Because the location of a heap pointer cannot always be statically known due to pointer aliasing issues, store instructions are conservatively instrumented unless it is soundly determined to be a stack-located pointer. Any possible over-instrumentation due to this conservative choice will be handled using the runtime object range analysis, which we will describe in the next subsection (§III-C).

Once all these sanity checks are passed, a `trace()` function call is inserted after the store instruction. For example, to instrument `doc->child = body` in Example 1, DANGNULL inserts `trace(&doc->child, body)` after its assignment instruction. In this way, the DANGNULL's runtime library can later correctly trace the pointer references originating from `doc->child`.

Note that DANGNULL relies on the type signature of `C/C++`. Coercing type signatures in the code might cause some false negatives, meaning that DANGNULL can miss some propagation of pointers at runtime. In particular, if developers convert types of pointer objects (by casting) into a value of non-pointer types, then DANGNULL will not be able to trace the pointer propagation via that value. Moreover, if some libraries are not built using DANGNULL (e.g., proprietary libraries), DANGNULL would still be able to run them together, but the protection domain will be limited only to the instrumented code or modules.

C. Runtime Library

The runtime library of DANGNULL maintains all the object relationship information with an efficient variant of a red-black tree, called `shadowObjTree`. Object layout information (i.e., address ranges of an object) is populated by interposing all memory allocation and deallocation functions (e.g., `malloc` and `free`, `new` and `delete`, etc). Object relationships (i.e., an object refers to another object) are captured with the help of `trace()` added during the static instrumentation. Based on the collected object relationship information, the runtime library automatically nullifies all dangling pointers when the target memory is freed.

In this subsection, we first describe `shadowObjTree`, a data structure designed for maintaining the complex object relationships (§III-C1). We then further describe how these data structures are populated and how dangling pointers are immediately nullified during runtime (§III-C2).

1) Shadow Object Tree: DANGNULL records and maintains the relationships between objects[3] in `shadowObjTree`. It has a hierarchical structure because the object relationship itself

[1] In the LLVM IR, store instructions are always in the form of `lhs := rhs`.

[2] In the LLVM IR, the type of `lhs` of a store instruction is always the pointer of the `rhs`'s type.

[3] Since DANGNULL only tracks pointers stored on heap, the point-to relationship effectively becomes a relationship between heap objects.

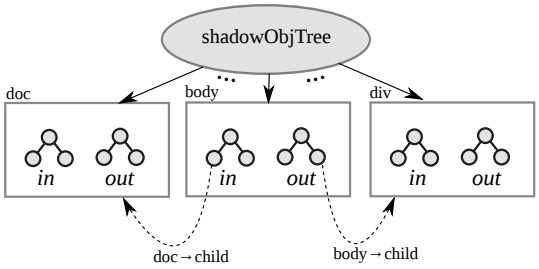


Figure 2: The shadow object tree and three shadow objects (`doc`, `body`, and `div`) corresponding to Example 1. To simplify the representation, only in- and out- bound pointers of the `body` shadow object are shown. `body` keeps the in-bound pointer for `doc→child`, which points to the shadow object of `doc`, and the out-bound pointer for `body->child`, which points to the shadow object of `div`.

is hierarchical: each running process has multiple objects, and each object has multiple in/out-bound pointers. Thus, `shadowObjTree` is composed of several sub-data structures as nodes, to better represent this hierarchical structure.

Figure 2 shows a structural view of `shadowObjTree`. A node of `shadowObjTree` is a shadow object, which holds the object's memory layout information (i.e., the object boundary with base and end addresses) and in/out-bound pointers (i.e., directed references between objects). To find a shadow object for the given pointer p, `shadowObjTree` searches for a shadow object such that the corresponding object's $base \leq p < end$. In other words, as long as the pointer p points to a corresponding object's address range, `shadowObjTree` returns the shadow object.

To efficiently support operations like insert, remove, and search, `shadowObjTree` uses a variant of the red-black tree as the underlying data structure, which generally excels if 1) the insertion order of keys shows random patterns and 2) the number of search operations is significantly more than that of insertion and remove operations. In `shadowObjTree`, the key is the address of the object, and the order of allocated objects' addresses eventually depends on the `mmap()` system call, which shows random behavior in modern ASLR enabled systems. Moreover, DANGNULL requires significantly more `find()` operations than `allocObj()` and `freeObj()` operations to soundly trace the object relationships.

Note that a hash table would not be an appropriate data structure for `shadowObjTree` because it cannot efficiently handle the size information of an object. To be specific, a find operation of `shadowObjTree` must answer range-based queries (i.e., finding a corresponding shadow object given the address, where the address can point to the middle of a shadow object), but a hash function of a hash table cannot be efficiently designed to incorporate such range information.

In addition, `shadowObjTree` has two sub-trees to maintain in/out-bound pointer information, and each sub-tree uses a red-black tree as the underlying data structure for the same reason described for `shadowObjTree`. As the pointer reference is directed, an in-bound reference of the object denotes that the object is pointed to by some other object and an out-bound reference denotes that it points to some other object. For example, the `body` object in Example 1 has `doc→child` as an in-bound sub-tree and `div` as an out-bound sub-tree.

2) Runtime Operations and Nullification: Upon running the instrumented binary, the runtime library of DANGNULL interposes all memory allocations and deallocations, and redirects their invocations to `allocObj()` and `freeObj()`. In addition, `trace()` instructions were inserted at pointer propagation instructions from the static instrumentation. As a running example, Example 2 illustrates how DANGNULL interposes and instruments the example code in Example 1 where `+` marked lines show the interposed or instrumented code.

The algorithm for the runtime library, which populates `shadowObjTree`, is described in Algorithm 2. Upon the memory allocation invocation, `allocObj()` first invokes corresponding real allocation functions (line 2). With the base pointer address from the real allocation, a shadow object is created and inserted to `shadowObjTree` as a node (lines 3-4). When `trace()` is invoked, the object relationship is added to the shadow objects. It first fetches two shadow objects representing `lhs` and `rhs` pointers, respectively (line 9-10). Next, with the concrete runtime values on pointers, DANGNULL uses the object range analysis to check whether `lhs` and `rhs` truly point to live heap objects (line 13). It is worth noting that this object range analysis not only helps DANGNULL avoid tracing any incorrect or unnecessary pointer semantics that are not related to dangling pointer issues, but also makes DANGNULL more robust on object relationship tracings, since it is based on concrete values and reasons about the correctness of pointer semantics with the liveness of source and destination heap objects. If the check passes, DANGNULL first removes an existing relationship, if there is any (line 14). It then inserts the shadow pointer to both shadow objects (line 16-17).

Note, by using `shadowObjTree`, DANGNULL does not need to handle pointer arithmetic to trace pointers. Specifically, because `shadowObjTree` contains information (`base` and `size`) of all live heap objects, given any pointer p, DANGNULL can locate the corresponding object through a search query (`shadowObjTree.find()`), i.e., finding object that has its `base` $\leq q <$ (`base + size`). For the same reason, although DANGNULL does not trace non-heap-located pointers (i.e., pointers in the stack or global variables), DANGNULL can still trace correctly when the pointer value is copied through them and brought back to the heap.

When an object is freed with `freeObj()`, the actual nullification starts. DANGNULL first fetches its shadow object (line 21). Next, it iterates over all in-bound pointers (pointing to the current object to be freed), and nullifies them with a pre-defined value (`NULLIFY_VALUE` at line 27). Note that these in-bound pointers are the pointers that would become dangling otherwise, and the pre-defined value can be set as any relatively small non-negative integer value (e.g., 0, 1, 2, ...). To avoid the erroneous nullification due to later deallocation of objects that the current object points to, DANGNULL also removes the current object from the sub-tree of the out-bound pointers (lines 29-31).

It is worth noting that DANGNULL nullifies not only unsafe dangling pointers, but also benign dangling pointers. In spite of this extra nullification, DANGNULL can still retain the same program behavior semantics because benign dangling pointers should not have any pointer semantics (i.e., never be used).

```
1  // (a) memory allocations
2  + Document *doc = allocObj(Document);
3  + Body *body = allocObj(Body);
4  + Div *div = allocObj(Div);
5
6  // (b) using memory: propagating pointers
7  doc->child = body;
8  + trace(&doc->child, body);
9
10 body->child = div;
11 + trace(&body->child, div);
12
13 // (c) memory free: unsafe dangling pointer, doc->child,
14 //                  is automatically nullified
15 + freeObj(body);
16
17 // (d) use-aftre-free is prevented, avoid dereferencing it
18 if (doc->child)
19     doc->child->getAlign();
```

Example 2: Instrumented running example of Example 1 (actual instrumentation proceeds at the LLVM Bitcode level). Memory allocations (`new`) and deallocations (`free`) are replaced with `allocObj()` and `freeObj()`, and `trace()` is placed on every memory assignment, according to the static instrumentation algorithm (Algorithm 1).

In most cases as we quantified in §V, DANGNULL behaves correctly without false positives. We have found one rare false positive case, described in detail in §VI.

In our secured binary (Example 2), `doc→child` is automatically nullified when `body` is freed: the shadow object representing `body` was created (line 3), propagated to `doc→child` (line 8), and nullified when the `body` is deallocated (line 15). As a result, depending on `NULLIFY_VALUE`, the example would raise the `SIGSEGV` exception (if `NULLIFY_VALUE` > 0) or continuously run (if `NULLIFY_VALUE` == 0), both of which safely mitigates negative security impacts by unsafe dangling pointers.

For the `SIGSEGV` exception cases, DANGNULL guarantees that the program securely ends in a safe-dereference, which is defined as follows.

Definition 4. Safe-dereference. *If a dereference instruction accesses the memory address in the range of [0, N] where it is preoccupied as non-readable and non-writable memory pages for a given constant N, such a dereference is a safe-dereference.*

A safe-dereference guarantees that a dereference on nullified unsafe dangling pointers turns into a secured crash handled either by the operating system or DANGNULL's `SIGSEGV` exception handler. In modern operating systems, it is common that the first several virtual memory pages are protected to avoid any potential null-dereference attacks (e.g., virtual address spaces from 0 to 64K are protected in Ubuntu [46]). In other words, DANGNULL can utilize this existing null address padding to guarantee safe-dereferences (64K in Ubuntu). Even if this null address padding is not supported by the operating system, DANGNULL can still pre-allocate these spaces using the `mmap()` system call to be non-readable and non-writable before any other code runs.

For continuously running cases, DANGNULL utilized the existing sanity check at line 18. This is because the semantic on invalid pointers is identical to both DANGNULL's nullification and typical programming practices. In other words, because it is common for developers to check whether the pointer value is null before accessing it, DANGNULL's nullification can utilize such existing checks and keep the application running as if there were no unsafe dangling pointers.

```
1  def allocObj(size):
2      ptr = real_alloc(size)
3      shadowObj = createShadowObj(ptr, size)
4      shadowObjTree.insert(shadowObj)
5      return ptr
6
7  # NOTE. lhs <- rhs
8  def trace(lhs, rhs):
9      lhsShadowObj = shadowObjTree.find(lhs)
10     rhsShadowObj = shadowObjTree.find(rhs)
11
12     # Check if lhs and rhs are eligible targets.
13     if lhsShadowObj and rhsShadowObj:
14         removeOldShadowPtr(lhs, rhs)
15         ptr = createShadowPtr(lhs, rhs)
16         lhsShadowObj.insertOutboundPtr(ptr)
17         rhsShadowObj.insertInboundPtr(ptr)
18     return
19
20 def freeObj(ptr):
21     shadowObj = shadowObjTree.find(ptr)
22
23     for ptr in shadowObj.getInboundPtrs():
24         srcShadowObj = shadowObjTree.find(ptr)
25         srcShadowObj.removeOutboundPtr(ptr)
26         if shadowObj.base <= ptr < shadowObj.end:
27             *ptr = NULLIFY_VALUE
28
29     for ptr in shadowObj.getOutboundPtrs():
30         dstShadowObj = shadowObjTree.find(ptr)
31         dstShadowObj.removeInboundPtr(ptr)
32
33     shadowObjTree.remove(shadowObj)
34
35     return real_free(ptr)
```

Algorithm 2: The Runtime library algorithm. All error handling and synchronization code is omitted for clarity. DANGNULL has a global data structure (thread-safe), `shadowObjTree`, to maintain object relations with shadow objects. `allocObj()` and `freeObj()` replaced the `malloc()` and `free()` (and their equivalence, `new` and `delete` in C++), and `trace()` will be inserted on every memory assignments as a result of the static instrumentation (§III-B).

This example is oversimplified for the purpose of clarifying the problem scope and showing how DANGNULL can nullify dangling pointers. In §V-A, we show that DANGNULL is effective when applied to real, complex use-after-free bugs in Chromium.

IV. IMPLEMENTATION

We implemented DANGNULL based on the LLVM Compiler project [43]. The static instrumentation module is implemented as an extra LLVM pass, and the runtime library is implemented based on LLVM compiler-rt with the LLVM Sanitizer code base. Table III shows the lines of code to implement DANGNULL, excluding empty lines and comments.

We placed the initialization function into `.preinit_array` as a `ELF` file format so that the initialization of DANGNULL is done before any other function[4]. In this function, all standard allocation and deallocation functions (e.g., `malloc` and `free`, `new` and `delete`, etc) are interposed. In total, DANGNULL interposed 18 different allocation functions in the current implementation, and any additional customized allocators for

[4]DANGNULL's prototype is implemented on a Linux platform. Although several implementation details are specific to Linux, these can be generally handled in other platforms as well.

Components	Lines of code
Static Instrumentation	389
Runtime Library	3,955
`shadowObjTree`	1,303
Red-black tree	476
Runtime function redirection	233
Others	1,943
Total	4,344

Table III: Components of DANGNULL and their complexities, in terms of their lines of code. All components are written in `C++`.

the target application can be easily added with one line of its function signature.

To avoid multi-threading issues when running DANGNULL, we used mutex locks for any data structures with the potential for data racing. One global mutex lock is used for `shadowObjTree`, and all shadow objects and their in/out–bound pointer sub-trees also hold their own mutex locks.

To retain the target program's original memory layout, DANGNULL uses a dedicated allocator from Sanitizer that has dedicated memory pages. All memory for metadata, including `shadowObjTree` and its pointer sub-trees, is allocated from this allocator. Thus, DANGNULL does not interfere with the original memory layout, and it can avoid any potential side effects by manipulating the original allocators [14].

We also modified the front-end of LLVM so that users of DANGNULL can easily build and secure their target applications with one extra compilation option and linker option. To build SPEC CPU 2006 benchmarks, we added one line to the build configuration file. To build the Chromium browser, we added 21 lines to the `.gyp` build configuration files.

V. EVALUATION

We evaluated DANGNULL on two program sets, the SPEC CPU2006 benchmarks [40] and the Chromium browser [41] [5]. First, we tested how accurately DANGNULL mitigates known use-after-free exploits (§V-A). Next, we measured how much overhead DANGNULL imposes during the instrumentation phase (§V-B) and the runtime phase (§V-C). Finally, we conducted a stress test to see if DANGNULL runs well without breaking compatibility (§V-D). All experiments were conducted on an Ubuntu 13.10 system (Linux Kernel 3.11.0) with a quad-core 3.40 GHz CPU (Intel Xeon E3-1245), 16 GB RAM, and 1 TB SSD-based storage.

A. Attack Mitigation

The goal of DANGNULL is to turn use-after-free or double-free attempts into safe-dereferences by nullifying dangling pointers. In order to test how DANGNULL accurately nullified unsafe dangling pointers and eventually protected the system from temporal memory safety violations, we tested the DANGNULL-hardened Chromium browser with real-world use-after-free exploits. Given the Chromium version instrumented (29.0.1457.65), we first collected all publicly available use-after-free exploits from the Chromium bug tracking system [42], which opens vulnerability information to the public after mitigation and includes a proof of concept exploit[6].

Table IV lists seven use-after-free vulnerabilities that existed in the targeted Chromium version. All of these were marked as high severity vulnerabilities by the Chromium team, which suggests that these have a high potential to be used for arbitrary code execution. Bug ID 162835 was specifically selected to later demonstrate that DANGNULL can mitigate this sophisticated exploit technique.

Before applying DANGNULL, all proofs-of-concept can trigger `SIGSEGV` exceptions at invalid addresses (`No-Nullify` column in Table IV). These invalid addresses are memory addresses that are dereferenced, i.e., the values of unsafe dangling pointers. Although we only present one value for each vulnerability, this value would randomly change between different executions due to ASLR and the order of memory allocations. These seemingly random `SIGSEGV` exceptions can be abused to launch control-flow hijacking attacks, information leaks, etc. They are particularly dangerous if the vulnerability offers a control between free and use (the right-most column, `Control b/w free and use`). For example, with this control, malicious JavaScript code can place crafted data in freed memory and turn the `SIGSEGV` exception (i.e., deference the unsafe dangling pointer) into control-flow hijacking attacks or information leakages depending on the context of dereference operations. Moreover, this control between free and use also implies that the attackers can bypass memory error detection tools (e.g., AddressSanitizer [38]) because it allows the attackers to force the reuse of a freed memory region (see more details in §VII and a concrete bypassing case (Example 5)).

Once Chromium is instrumented with DANGNULL, all of these cases were safely mitigated (`Nullify-value` column). Depending on the nullify value provided as a parameter, all 28 cases (7 rows by 4 columns) result in the following three categories: 1) integer values represent that DANGNULL securely contained `SIGSEGV` exceptions with safe-dereference; 2) *stopped by assertion* represents that DANGNULL re-utilized existing safe assertions in Chromium; and 3) check marks (✓) represent that Chromium continuously runs as if Chromium is already patched.

For the safe-dereference cases, it is worth noting that the dereferenced address values are quite small (at most 0x2e8). Although these seven exploits would not be enough to represent all use-after-free behaviors, we believe this implies that the moderate size of null address padding for safe-dereference (§III-C2) would be effective enough. DANGNULL's null address padding can be easily extended without actual physical memory overheads if necessary, and 64-bit x86 architectures can allow even more aggressive pre-mappings. Moreover, unlike the case before applying DANGNULL, these dereferenced addresses did not change between different executions. This indicates that unsafe dangling pointers were successfully nullified using

[5]The Chromium browser is the open source project behind the Chrome browser, and these two are largely identical.

[6]We have not found any double-free vulnerabilities for the given Chromium version. However, we believe DANGNULL would be equally effective against double-free exploits because DANGNULL nullifies exploit attempts where both use-after-free and double-free share common erroneous behaviors (i.e., at the moment when the unsafe dangling pointer is created).

DANGNULL and thus the random factor of the object's base address is canceled out.

In addition, the dereferenced address values show certain patterns depending on the nullify value. While address values are different when the nullify value is 0 (explained later in this section), address values generally show linear relationships with nullify values. For example, in Bug ID 261836, address values were {0x21, 0x22, 0x23} when nullify values were {1, 2, 3}, respectively. This pattern presents another clue that unsafe dangling pointers were actually nullified by DANGNULL because the difference of each address value and the nullify value is the same between different executions. We manually verified that the difference is actually the index offsets of the dereference operations by analyzing the crash call stacks and the source code. Bug ID 282088, which has 0xf0 for all different nullify values, was an exception. This is caused by aligned memory access in the dereference operation (i.e., the pointer is AND masked with 0xfffffff0).

As we mentioned in §III-C2, DANGNULL can also utilize existing sanity checking code. These cases are shown as a check mark (✓) indicating that Chromium correctly handled the exploit as if it was patched. Because program developers usually insert null pointer checks before pointer dereferences, unsafe dangling pointers nullified by DANGNULL did not flow into the dereference instructions which would cause safe-dereference. Instead, it was handled as an expected error and Chromium displayed the same output as a later patched version. We admit that it would be premature to argue that unsafe dangling pointers nullified by DANGNULL can always be handled in this way, but we believe this is an interesting direction for future work. Existing research [25] also showed that skipping erroneous null-dereference instructions can keep the target application safely running. Besides normal checks, DANGNULL was also able to re-utilize existing null pointer assertions, as demonstrated in Bug ID 162835.

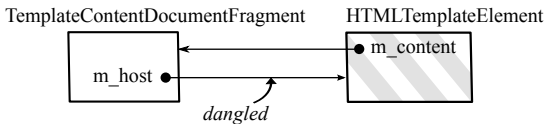
To clearly illustrate how DANGNULL mitigates these use-after-free exploits on the Chromium browser, Example 3 shows the simplified vulnerable code snippet and its object relationships for Bug ID 286975. Two class objects hold mutual relationships with their own member variables. The root cause of the vulnerability is in that the in-bound pointer (`m_host`) of `HTMLTemplateElement` is not nullified when it is freed (patches are shown as + marked lines). As a result, use-after-free occurs when `containsIncludingHostElements()` dereferences `m_host`. Under DANGNULL, it immediately eliminates the root cause of the vulnerability by nullifying `m_host` when the object is freed. Thus, DANGNULL not only mitigated Use-After-Free, but also utilized the existing null pointer checks at line 24 and helped the browser run as if it were patched.

B. Instrumentation

To see how DANGNULL's static instrumentation changes the output binary, we measured the number of inserted instructions and the file sizes of both the original and instrumented binaries. The number of inserted instructions depends on the number of pointer propagation instructions as described in Algorithm 1. Accordingly, file size increments in instrumented binaries are proportional to the number of inserted instructions (excepting the 370KB runtime library).

```
1  class TemplateContentDocumentFragment: public Node {
2    const Element* m_host;
3    // nullify its m_host
4  + void clearHost() { m_host = NULL;}
5  };
6
7  class Element: public Node;
8
9  class HTMLTemplateElement: public Element {
10   mutable RefPtr<Node> m_content;
11   ~HTMLTemplateElement() {
12     // nullify if m_content is alive
13 +   if (m_content)
14 +     m_content->clearHost();
15   }
16 };
17
18 class Node {
19   void containsIncludingHostElements(Node *node) {
20     TemplateContentDocumentFragment *t = \
21           (TemplateContentDocumentFragment*)node;
22
23     // null-pointer check
24     if (t->m_host)
25       // dereference dangling pointer
26       ((Node*)(t->m_host))->getFlags();
27   }
28 };
```

(i) Simplified code snippet. Lines marked + are the vulnerability patch.



(ii) Objects and their relationships while running (i)

Example 3: The vulnerable code and its object relationships for Bug ID 286975. `TemplateContentDocumentFragment` and `HTMLTemplateElement` have mutual references via `m_host` and `m_content`. In the vulnerable code, even after the `HTMLTemplateElement` object is freed, the `TemplateContentDocumentFragment` object still holds a reference to the object in `m_host`, resulting in a use-after-free vulnerability when `containsIncludingHostElements()` is invoked. With DANGNULL, use-after-free is mitigated, and Chromium continues to run correctly.

The results for SPEC CPU 2006 benchmarks are shown in Table V. A total of 16 programs were instrumented (eleven C programs and five C++ programs). The number of inserted instructions varied across each program. This variance is not only influenced by the total number of instructions, but also by a program's characteristics (some programs have more pointer propagation than others). For example, although mcf has fewer total instructions, DANGNULL inserted more than 10 times the number of instructions into mcf than lbm, a similarly-sized application. The file size increments were proportional to the number of inserted instructions as expected after subtracting the size of the runtime library.

The result for the Chromium browser is shown in Table VI. A total of 140,000 instructions were inserted, which is less than 1% of the whole program. This implies that pointer propagation instructions appeared with less than 1% probability. The file size is increased by about 0.5%, for a total size of 1,868 MB. Note that the Chromium build process uses static linking, and thus the resultant binary includes all shared libraries. We believe the file size increment (0.5%) should not be a concern for

Bug ID	CVE	Severity	No-Nullify	Nullify-value				Control b/w
				0	1	2	3	free and use
261836	-	High	0x7f27000001a8	0x2e8	0x21	0x22	0x23	yes
265838	2013-2909	High	0x1bfc9901ece1	✓	0x1	0x2	0x3	yes
279277	2013-2909	High	0x7f2f57260968	✓	0x1	0x2	0x3	yes
282088	2013-2918	High	0x490341400000	0xf0	0xf0	0xf0	0xf0	difficult
286975	2013-2922	High	0x60b000006da4	✓	0x15	0x16	0x17	yes
295010	2013-6625	High	0x897ccce6951	0x30	0x1	0x2	0x3	yes
162835	2012-5137	High	0x612000046c18	*stopped by assertion*				yes

Table IV: DANGNULL safely nullified all seven real-world use-after-free exploits for Chromium. Among these seven cases, three correctly run even after use-after-free exploits as if it was patched (represented as a ✓), and one is safely prevented as DANGNULL re-utilized existing assertions in Chromium (represented as *stopped by assertion*). Without DANGNULL, all exploits are potential threats, leading to control-flow hijacking attacks, information leakages, etc. To be concrete, we also include invalid pointers causing an exception with various nullification values (0-3), and their threat in terms of the chances of an attacker's control between free and use.

Name	Lan.	File Size (KB)			# of instructions		# of objects		# of pointers		# Nullify	Memory (MB)	
		before	after	incr.	inserted	total	total	peak	total	peak		before	after
bzip2	C	172	549	378	13	15,370	7	2	0	0	0	34	34
gcc	C	8,380	9,148	768	9,264	606,925	165k	3k	3167k	178k	104k	316	397
mcf	C	53	429	376	95	2,277	2	1	0	0	0	569	570
milc	C	351	737	386	71	24,024	38	33	0	0	0	2,496	2,500
namd	C++	1,182	1,564	382	45	77,434	964	953	0	0	0	44	114
gobmk	C	5,594	6,010	416	201	156,829	12k	47	0	0	0	23	28
soplex	C++	4,292	4,745	453	264	74,314	1k	88	14k	172	140	7	14
povray	C++	3,383	3,896	513	941	194,821	15k	9k	7923k	26k	6k	38	81
hmmer	C	814	1,210	396	94	60,832	84k	28	0	0	0	1	18
sjeng	C	276	662	386	17	22,836	1	1	0	0	0	171	171
libquantum	C	106	483	378	21	7,301	49	2	0	0	0	0	2
h264ref	C	1,225	1,646	420	154	115,575	9k	7k	906	111	101	44	208
lbm	C	37	411	374	9	2,341	2	1	0	0	0	408	409
astar	C++	195	574	378	54	8,220	130k	5k	2k	148	20	13	135
sphinx3	C	541	931	389	170	34,476	6k	703	814k	14k	0	46	62
xalancbmk	C++	48,538	51,010	2472	7,364	645,434	28k	4k	256k	18k	10k	7	76

Table V: Details of instrumented binaries (the left half) and their runtime properties (the right half) in SPEC CPU2006. The left half describes the details of incremented file size due to newly inserted instrumentation instructions. The runtime library of DANGNULL is about 370 KB; DANGNULL requires approximately 40 B per instrumentation to trace pointer propagation. The right half represents the details of the programs' runtime behavior (e.g., increase of memory usage and the number of pointers and objects in each benchmark). The increase of memory (due to `shadowObjTree`) depends on the number of objects and pointers created and freed in total; `bzip2`, which has minimal memory allocation, imposed no extra memory overhead, while `gcc`, which has many memory operations, imposes about 80 MB of extra memory overhead with DANGNULL.

Name	File Size (MB)			# of instructions	
	before	after	incr.	inserted	total
Chromium	1858	1868	10	140k	16,831k

Table VI: Static instrumentation results on Chromium

distribution or management of the instrumented binary.

C. Runtime Overheads

As DANGNULL must trace object relationships for nullification, it increases both execution time and memory usage. To determine how much runtime overhead DANGNULL imposes on target applications, we measured various runtime overheads of SPEC CPU2006 and the Chromium browser.

Figure 3 shows the runtime performance overheads of DANGNULL running SPEC CPU2006 benchmarks. The overheads largely depend on the number of objects and pointers that DANGNULL traced and stored in `shadowObjTree`. These metadata tracing measurements are shown in the right half of Table V. As we described in §V-B, each application has a different number of object allocations and degree of pointer propagation. Accordingly, each object allocation and pointer propagation would insert extra metadata into `shadowObjTree` unless it fails runtime range analysis. DANGNULL imposed an average performance overhead of 80%. DANGNULL caused more runtime overhead if the application had to trace a large number of pointers. For example, in the povray case, a total of 7,923,000 pointers were traced because it maintains a large number of pointers to render image pixels, and thus increased execution time by 270% with 213% memory overhead. On the other hand, in h264ref, only 906 pointers were traced and resulted in a 1% increase in execution time and 472% memory overhead.

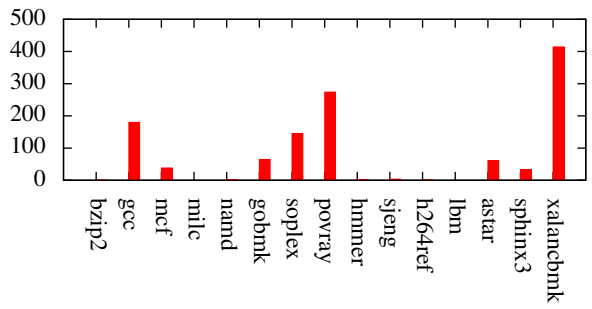


Figure 3: Runtime speed overheads when running SPEC CPU2006. y-axis shows slowdowns (%) of DANGNULL.

To obtain a practical measurement of performance impacts of DANGNULL, we also tested the DANGNULL-hardened Chromium browser to see how much DANGNULL would affect web navigation experiences. First, seven different browser benchmarks listed in Table VII are tested. Among them, Octane 2.0 [17], SunSpider 1.0.2 [47], and Dromaeo JS [28] evaluate the JavaScript engine performance. Balls [45], Dromaeo DOM, JS Lib, and html5 [45] evaluate rendering performance. For the JavaScript engine benchmarks, DANGNULL showed only 4.8% averaged overhead. This is because heavy JavaScript computations are mostly executed in the form of JIT-compiled code, which is not instrumented by DANGNULL. For rendering benchmarks, DANGNULL showed 53.1% overhead on average, which implies that rendering computations (i.e., DOM or HTML related computations) were affected by DANGNULL to some extent.

Note that the actual overhead for most end-users would be the combined overheads of both JavaScript and rendering computations. Therefore, we also measured the page loading time for the Alexa top 100 websites [3], as this would be representative of the actual overhead an end user would experience. To measure the page loading time, we developed a simple Chromium extension which computes the time difference between the initial fetch event and the end of page loading event. Each website is visited three times while the browser is newly spawned and the user profile directory is deleted before each visit to avoid page cache effects. On average, DANGNULL showed 7% increased load time. DANGNULL showed 2326 ms page loading time with 317.6ms standard deviation. Original Chromium showed 2165 ms with 377.9ms standard deviation. Due to the variability of network traversal and the response time of web servers, most performance impacts of DANGNULL introduce no greater magnitude of load time variability than those introduced by factors unrelated to DANGNULL.

To illustrate how DANGNULL behaves for web page loading, Table VIII shows detailed overheads. Four popular websites are visited, of which two were logged in using active accounts. When visiting `youtube.com`, page load time increased 32.8% as it renders many images. For login pages, DANGNULL maintained a similar overhead ratio even though it traced a relatively large number of objects and pointers.

D. Compatibility

Due to the nullification of benign dangling pointers, it is possible that DANGNULL may cause false positives by changing program execution semantics (i.e., dangling pointers that will not be dereferenced, but their address values will be use). To see how much negative impact DANGNULL would impose on such program execution semantics, we used the Chromium browser as an example and ran stress tests on DANGNULL with various benign inputs. These tests include base unit tests, LayoutTests, Acid3 tests, and visiting Alexa top 500 websites. The first three tests are well known tests to verify the correctness of browser implementations, and we additionally visited Alexa top 500 websites to see if DANGNULL causes any issues when browsing popular websites.

Base unit tests [9], which are distributed with the Chromium source code, are regression tests that directly communicate with the native interfaces of Chromium and check whether basic functions are working correctly. DANGNULL passed all 1,100 test cases.

LayoutTests [9], which are also distributed with the Chromium source code, include 30,120 test cases. These tests are web page files (e.g., `.html`, `.js`, etc), and check whether the browser renders various web pages as expected. Test results for Chromium secured with DANGNULL were identical to original Chromium. Both passed 30,035 tests, but failed the same 85 tests. For the failed cases, four tests crashed the browser, 70 tests failed to generate expected output, and 11 tests caused a timeout. We manually verified the four crashing tests, and all were to-be-fixed bugs in the Chromium version we evaluated. As the set of failed tests were the same for original and instrumented Chromium, we believe DANGNULL would match the rendering conformity of original Chromium.

Acid3 tests [44] check whether the browser conforms with published web standards. It runs various JavaScript code and compares the rendered output with a reference input to see if there are any differences. DANGNULL passed the test with the full score (100/100).

We also visited Alexa top 500 websites [3] to see if DANGNULL introduces any unexpected behavior. Each time the browser visits a website, we waited 10 seconds and checked whether DANGNULL raised false alarms during the visit. For all top 500 websites, DANGNULL displayed all web pages without false alarms.

Based on this compatibility evaluation, we believe that DANGNULL would not impact original execution semantics in practice. We found one rare false positive case while manually handling the browser, which is described in §VI.

VI. DISCUSSION

Usage scenarios of DANGNULL. Because this paper focuses on the correctness and effectiveness of DANGNULL against use-after-free and double-free vulnerabilities, we have not specifically restricted the possible usage scenarios of DANGNULL. In general, we believe DANGNULL can be applied to the following three scenarios:

- Back-end use-after-free detection: many zero-day attacks are based on use-after-free vulnerabilities [29]; DANGNULL can be used to detect these attacks. To the best of our knowledge, DANGNULL is the only mitigation system that can detect use-after-free

Benchmarks	JavaScript			Rendering			
(unit, [high or low])	**Octane** (score, high)	**SunSpider** (ms, low)	**Dromaeo JS** (runs/s, high)	**Balls** (fps, high)	**Dromaeo DOM** (runs/s, high)	**Dromaeo JS Lib** (runs/s, high)	**html5** (sec, low)
Original	13,874	320.0	1,602.1	11.6	857.8	216.0	10.1
DANGNULL	13,431	347.5	1,559.6	6.5	509.1	168.1	20.7
Slowdown	3.2%	8.6%	2.7%	44.1%	40.7%	22.2%	105.3%

Table VII: Chromium Benchmark results with and without DANGNULL. High/low denotes whether performance was higher or lower when compared to the unmodified test. For JavaScript benchmarks, DANGNULL imposes negligible overheads, varying from 2.7-8.6%. For rendering benchmarks requiring lots of memory operations (e.g., allocating DOM elements), DANGNULL exhibits 22.2%-105.3% overhead depending on type.

Website	Action	Page Complexity		Loading Time (sec)		# of objects		# of pointers		# Nullify	Memory (MB)	
		# Req	# DOM	Original	DANGNULL	total	peak	total	peak		before	after
gmail.com	visit	13	164	0.49	0.60 (22.4%)	123k	22k	32k	12k	7k	46	171
twitter.com	visit	14	628	1.05	1.16 (10.5%)	121k	23k	35k	13k	8k	48	178
amazon.com	visit	264	1893	1.37	1.60 (16.8%)	166k	25k	81k	28k	16k	57	200
youtube.com	visit	43	2293	0.61	0.81 (32.8%)	127k	23k	46k	16k	9k	49	178
gmail.com	login	177	5040	6.40	7.66 (19.7%)	295k	31k	165k	49k	32k	96	301
twitter.com	login	60	3124	2.16	2.77 (28.2%)	172k	27k	71k	23k	15k	98	276

Table VIII: Details of DANGNULL overhead when visiting four popular websites. The left half shows page complexities and page load time. The right half shows detailed runtime properties.

exploits carefully developed for complex and large scale software (e.g., Chromium).

- Runtime use-after-free mitigation for end users: if performance overhead is not the primary concern of end users, DANGNULL is an effective use-after-free mitigation tool with moderate performance overhead, especially for web browsers.
- Use-after-free resilient programs: we have shown that DANGNULL can utilize existing sanity check routines and survive use-after-free attempts. By integrating automatic runtime repair work [25], we believe DANGNULL can evolve to support use-after-free resilient programs in the future.

Performance optimization. We believe DANGNULL's performance overhead can be further improved, especially for performance critical applications. First of all, instrumentation phases can be further optimized by leveraging more sophisticated static analysis. For example, if it is certain that the original code already nullifies a pointer, DANGNULL would not need to nullify it again. Although we have not heavily explored this direction, this has to be done carefully because soundly learning this information involves pointer-aliasing problems, which are well-known difficult problems in program analyses, and any incorrect analysis results would lead to both false positives and false negatives.

Moreover, we identified that manipulation of `shadowObjTree` is the main performance bottleneck, and this can be optimized by 1) leveraging transactional memory [20] to enhance locking performance on `shadowObjTree`; 2) designing a software cache for `shadowObjTree`; 3) using alternative data-structures to implement `shadowObjTree` (e.g., alignment-based metadata storage by replacing the memory allocator [18]); or 4) creating a dedicated analyzing thread or process for `shadowObjTree` [23].

False negatives. DANGNULL's static instrumentation assumes that a pointer is propagated only if either the left– or right–hand side of a variable is a pointer type. This would not be true if the program is written in a manner such that the propagation is done between non-pointer-typed variables. Consider the example we introduced in Example 1. If the child member variable is typed as `long` (i.e., `long child` at line 4) and all the following operations regarding child are using type casting (i.e., `doc->child=(long)body` at line 13 and `((Elem*)doc->child)->getAlign()` at line 21), then such a pointer propagation would not be traced. DANGNULL would under-trace object relationships in this case, and there would be false negatives if child becomes an unsafe dangling pointer.

False positives. To stop dereferencing on unsafe dangling pointers, DANGNULL nullifies not only unsafe dangling pointers but also benign dangling pointers. This implies DANGNULL additionally nullifies benign dangling pointers, and it is possible it may cause some false positives, although these should not have any semantic meaning as they are "dangled".

While testing DANGNULL for SPEC CPU benchmarks and the Chromium browser, we found one rare false positive case. This false positive case sporadically occurs when a new tab is manually created inside the Chromium browser, and it is related to the unique pointer hash table design (Example 4).

We believe this false positive example would not be a critical concern for deploying DANGNULL due to its rareness. As we described in the compatibility evaluation in §V-D, DANGNULL passed more than 30,000 stress tests with the Chromium browser, a large scale and highly complicated application.

VII. RELATED WORK

Memory-related issues, including invalid memory accesses, memory leaks, and use-after-free bugs, have been studied for many years. Numerous methods have been proposed for C/C++

```
1  enum child_status {IN_USE=0, DELETED=1};
2  hash_map <Element*, child_status, ptrHash> allChilds;
3
4  Document *doc = new Document();
5  Element *elem = new Element();
6
7  // hold child reference
8  doc->child = elem;
9
10 // mark it as in-use in the hash_map
11 allChilds[elem] = IN_USE;
12
13 // delete child, nullified accordingly
14 delete doc->child;
15
16 // doc->child is nullified,
17 //   but Chromium relies on the stale pointer
18 allChilds[doc->child] = DELETED;
19
20 // makes sure all childs are deleted
21 for (it = allChilds.begin(); it != allChilds.end(); ++ it)
22    if (it->second == IN_USE)
23       delete it->first;
```

Example 4: A simplified false positive example of DANGNULL in the Chromium browser. This sporadically occurred when the tab is manually created inside the browser. If applications rely on the stale pointer (using the freed pointer as a concrete value, as `doc→child` in line 18), DANGNULL can cause a false positive. We fixed this issue for DANGNULL by switching the order of deletion and marking operations (switching line 14 and line 18).

programs. In this section, we categorize them and compare them with DANGNULL.

Use-after-free detectors. There is a line of research specifically focusing on detecting use-after-free vulnerabilities. In general, use-after-free vulnerabilities can be detected through both static and dynamic analysis. However, since 1) a dangling pointer itself is not erroneous behavior and 2) statically determining whether a dangling pointer will actually be used in the future requires precise points-to and reachability analyses across all possible inter-procedure paths, even state-of-the-art use-after-free detection tools based on the static analysis are only suitable for analyzing small programs [13, 34].

For this reason, most use-after-free detectors [6, 30, 48] are based on the runtime dynamic analysis. CETS [30] maintains a unique identifier with each allocated object, associates this metadata with pointers, and checks that the object is still allocated on pointer dereferences. To handle pointer arithmetic, CETS uses taint propagation (i.e., the resulting pointer will inherit the metadata from the base pointer of the corresponding arithmetic operation). Unfortunately, the assumption behind this design choice —the propagated pointer should point to the same object—does not always hold, which results in high false positive rates. From our experiments, CETS raised false alarms on 5 out of 16 tested programs while DANGNULL was able to correctly run all 16 programs. In addition to high false positive rates, CETS relies on explicit checks to guarantee the memory access validity for all memory operations, thus imposing higher performance overhead in SPEC CPU2006 compared to DANGNULL. For 4 programs (bzip2, milc, sjeng, h264ref, and lbm) that CETS was able to run[7], on average it incurred 40% slow down, while DANGNULL incurred 1% slow down.

Undangle [6] is another runtime dynamic analysis tool to detect use-after-free. It assigns each return value of memory allocation functions a unique label, and employs a dynamic taint analysis to track the propagation of these labels. On memory deallocation, Undangle checks which memory slots are still associated with the corresponding label, and then determines the unsafe dangling pointers based on the *lifetime* of dangling pointers (i.e., if the lifetime of a dangling pointer is higher than the certain threshold number, it is identified as an unsafe dangling pointer). While this approach can collect more complete pointer propagation information than DANGNULL (which would better help a bug triage or debugging process), a significant performance cost is required.

Control flow integrity. Control flow integrity (CFI) [1, 49–51] enforces indirect function calls to be legitimate (i.e., enforcing integrity of the control-flows). Similarly, SafeDispatch [22] prevents illegal control flows from virtual function call sites. Unlike use-after-free and memory error detectors, CFI makes use-after-free vulnerabilities difficult to exploit. Specifically, CFI only shepherds function pointers to guarantee legitimate control flows. In practice, however, most CFI implementations enforce coarse-grained CFI to avoid heavy performance overheads and false positive alarms, but recent research [7, 11, 15, 16] has demonstrated that all the aforementioned coarse-grained CFI implementations can be bypassed. Moreover, dangling pointers can also be abused to corrupt non-control data (e.g., vector length variables, user privilege bits, or sandbox enforcing flags) in objects [8], all of which are not function pointers, which makes CFI based protection techniques bypassable. For example, a recent attack [27] overwrote user permission bits in the metadata to bypass user authorizations, including all other defense mechanisms. As an another example, vector length variable corruption is one popular technique to exploit use-after-free vulnerabilities that lead to information leakage attacks or additional heap buffer overflows.

DANGNULL eliminates dangling pointers at the moment they are created. Thus, it can protect not only control flows but also any other security sensitive metadata in the objects from being abused by use-after-free or double-free vulnerabilities.

Memory error detectors. Memcheck (Valgrind) [32] and Purify [19] are popular solutions for detecting memory errors. Since their main goal is to help debugging, they are designed to be complete (incurring no false negatives) and general (detecting all classes of memory problems) in identifying memory-leak vulnerabilities, imposing very high memory and CPU overheads.

AddressSanitizer [38] is another popular tool developed recently that optimizes the method of representing and probing the status of allocated memory. However, due to an assumption to support this optimization (a quarantine zone that prevents reuse of previously freed memory blocks), it cannot detect use-after-free bugs if the assumption does not hold (i.e., heap objects are reallocated). Specifically, attackers can easily leverage various techniques to force reallocation of previously freed memory blocks, such as Heap Spraying [10, 36] and Heap Fengshui [39]. To clearly demonstrate this issue, we developed a proof-of-concept exploit bypassing the detection of AddressSanitizer (Example 5). However, with DANGNULL, all

[7] CETS failed to compile 7 programs out of 16 SPEC CPU2006 programs we tested.

```
function onOpened() {
  buf = ms.addSourceBuffer(...);
  // disconnect the target obj
  vid.parentNode.removeChild(vid);
  vid = null;
  // free the target obj
  gc();

+ var drainBuffer = new Uint32Array(1024*1024*512);
+ drainBuffer = null;
+ // drain the quarantine zone
+ gc();

+ for (var i = 0; i < 500; i ++) {
+   // allocate/fill up the landing zone
+   var landBuffer = new Uint32Array(44);
+   for (var j = 0; j < 44; j ++)
+     landBuffer[j] = 0x1234;
+ }

  // trigger use-after-free
  buf.timestampOffset = 100000;
}

ms = new WebKitMediaSource();
ms.addEventListener('webkitsourceopen', onOpened);

// NOTE.
//   <video id="vid"></video>
vid = document.getElementById('vid');
vid.src = window.URL.createObjectURL(ms);
```

Example 5: An example of exploits (Bug ID 162835) bypassing AddressSanitizer [38]. Lines with + marks show the bypassing routine, which keeps allocating the same sized memory to drain the quarantine zone of AddressSanitizer. Once the quarantine zone is drained, AddressSanitizer returns the previously freed memory block (i.e., an object is allocated in the previously freed memory region), which means it cannot identify memory semantic mismatches introduced by unsafe dangling pointers. Thus, AddressSanitizer cannot detect use-after-free exploits in this case, and this technique can be generalized to other use-after-free exploits with a different allocation size. However, DANGNULL detected this sophisticated exploit.

dangling pointers will be nullified upon the deallocation of their objects, rendering use-after-free vulnerabilities unexploitable, even with sophisticated manipulations.

Safe memory allocators. Many safe memory allocators have been proposed to prevent dangling pointer issues. Cling [2] can disrupt a large class of exploits targeting use-after-free vulnerabilities by restricting memory reuse to objects of the same type. Diehard and Dieharder [4, 33] mitigate dangling pointer issues by approximating an infinite-sized heap.

Smart pointers. A smart pointer is an abstract data type that encapsulates a pointer to support automatic resource management. Theoretically, an application would not suffer from use-after-free issues if all the pointers are represented with smart pointers (i.e., no raw pointers are used in the application code). However, it is common to expose raw pointers even in applications heavily using smart pointers. For example, in order to break the resource graph cycle connected with shared pointers (e.g., `std::shared_ptr` in C++11), browser rendering engines including WebKit [45] and Blink [5] usually expose a raw pointer instead of using weak pointers (e.g., `std::weak_ptr` in C++11) to avoid extra performance overheads and be compatible with legacy code [26], and these exposed raw pointers have been major use-after-free vulnerability sources for those engines. Note that automatically wrapping raw pointers with smart pointers is another challenging static analysis problem, as this requires understanding precise raw pointer semantics to be properly implemented.

VIII. CONCLUSIONS

In this paper, we presented DANGNULL, a system that detects temporal memory safety violations in runtime. We implemented DANGNULL, applied it to Chromium, and conducted a thorough evaluation showing the effectiveness and compatibility of DANGNULL. In particular, we demonstrated that DANGNULL can be applied to complex, large-scale software, such as the Chromium browser, to effectively mitigate use-after-free exploits with even the most sophisticated attack techniques. We believe DANGNULL can be used for a range of security applications: back-end use-after-free detection, runtime use-after-free mitigation, or use-after-free resilient programs.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers and our shepherd, Juan Caballero, for their help and feedback, as well as our operations staff for their proofreading efforts. This material is based upon work supported in part by the National Science Foundation under Grants No. CNS-1017265, CNS-0831300, and CNS-1149051, by the Office of Naval Research under Grant No. N000140911042, by the Department of Homeland Security under contract No. N66001-12-C-0133, and by the United States Air Force under Contract No. FA8650-10-C-7025. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Office of Naval Research, the Department of Homeland Security, or the United States Air Force.

REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] P. Akritidis, "Cling: A Memory Allocator to Mitigate Dangling Pointers," in *USENIX Security Symposium (Security)*, 2010.

[3] Alexa, "The Top 500 Sites on the Web," http://www.alexa.com/topsites, Aug 2014.

[4] E. D. Berger and B. G. Zorn, "DieHard: Probabilistic Memory Safety for Unsafe Languages," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[5] Blink : the rendering engine used by Chromium, http://www.chromium.org/blink, Aug 2014.

[6] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: Early Detection of Dangling Pointers in Use-after-free and Double-free Vulnerabilities," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2012.

[7] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses," in *USENIX Security Symposium (Security)*, 2014.

[8] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data Attacks Are Realistic Threats," in *USENIX Security Symposium (Security)*, 2005.

[9] Chromium Projects, "Running Tests at Home," http://www.chromium.org/developers/testing/running-tests, Aug 2014.

[10] M. Daniel, J. Honoroff, and C. Miller, "Engineering Heap Overflow Exploits with JavaScript," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2008.

[11] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose, "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection," in *USENIX Security Symposium (Security)*, 2014.

[12] D. Dhurjati and V. Adve, "Efficiently Detecting All Dangling Pointer Uses in Production Servers," in *International Conference on Dependable Systems and Networks (DSN)*, 2006.

[13] J. Feist, L. Mounier, and M.-L. Potet, "Statically detecting use after free on binary code," *Journal of Computer Virology and Hacking Techniques*, 2013.

[14] Flak, "Analysis of OpenSSL Freelist Reuse," http://www.tedunangst.com/flak/post/analysis-of-openssl-freelist-reuse, Aug 2014.

[15] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming Control-Flow Integrity," in *IEEE Symposium on Security and Privacy (Oakland)*, 2014.

[16] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard," in *USENIX Security Symposium (Security)*, 2014.

[17] Google, "Octane Benchmark," https://code.google.com/p/octane-benchmark, Aug 2014.

[18] Google, "Specialized memory allocator for ThreadSanitizer, MemorySanitizer, etc." http://llvm.org/klaus/compiler-rt/blob/7385f8b8b8723064910cf9737dc929e90aeac548/lib/sanitizer_common/sanitizer_allocator.h, Nov 2014.

[19] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *Winter 1992 USENIX Conference*, 1991.

[20] M. Herlihy and J. E. B. Moss, *Transactional memory: Architectural support for lock-free data structures*. ACM, 1993, vol. 21, no. 2.

[21] HP, "Pwn2Own 2014: A recap," http://www.pwn2own.com/2014/03/pwn2own-2014-recap, Aug 2014.

[22] D. Jang, Z. Tatlock, and S. Lerner, "SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks," in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[23] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis, "ShadowReplica: efficient parallelization of dynamic data flow tracking," in *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[24] LLVM Project, "LLVM Language Reference Manual," http://llvm.org/docs/LangRef.html.

[25] F. Long, S. Sidiroglou-Douskos, and M. Rinard, "Automatic Runtime Error Repair and Containment via Recovery Shepherding," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[26] Mads Ager, Erik Corry, Vyachslav Egorov, Kentaro Hara, Gustav Wibling, Ian Zerny, "Oilpan: Tracing Garbage Collection for Blink," http://www.chromium.org/blink/blink-gc, Aug 2014.

[27] Mallocat, "Subverting without EIP," http://mallocat.com/subverting-without-eip, Aug 2014.

[28] Mozilla, "DROMAEO, JavaScript Performance Testing," http://dromaeo.com, Aug 2014.

[29] S. Nagaraju, C. Craioveanu, E. Florio, and M. Miller, "Software Vulnerability Exploitation Trends," *Microsoft*, 2013.

[30] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "CETS: Compiler Enforced Temporal Safety for C," in *International Symposium on Memory Management (ISMM)*, 2010.

[31] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[32] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007, pp. 89–100.

[33] G. Novark and E. D. Berger, "DieHarder: Securing the Heap," in *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[34] H. Post and W. Küchlin, "Integrated static analysis for linux device driver verification," in *Integrated Formal Methods*. Springer, 2007, pp. 518–537.

[35] M. Prandini and M. Ramilli, "Return-oriented programming," *IEEE Security & Privacy*, 2012.

[36] P. Ratanaworabhan, B. Livshits, and B. Zorn, "NOZZLE: A Defense Against Heap-spraying Code Injection Attacks," in *USENIX Security Symposium (Security)*, 2009.

[37] R. Seacord, *Secure Coding in C and C++*, 1st ed. Addison-Wesley Professional, 2005.

[38] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *USENIX Conference on Annual Technical Conference (ATC)*, 2012.

[39] A. Sotirov, "Heap Feng Shui in JavaScript," *Black Hat Europe*, 2007.

[40] Standard Performance Evaluation Corporation, "SPEC CPU 2006," http://www.spec.org/cpu2006, Aug 2014.

[41] The Chromium Project, http://www.chromium.org/Home, Aug 2014.

[42] The Chromium Projects, "Chromium Issues," https://code.google.com/p/chromium/issues, Aug 2014.

[43] The LLVM Compiler Infrastructure, http://llvm.org, Aug 2014.

[44] The Web Standards Project, "Acid Tests," http://www.acidtests.org/, Aug 2014.

[45] The WebKit Open Source Project, http://www.webkit.org, Aug 2014.

[46] Ubuntu, "0-address protection in Ubuntu," https://wiki.ubuntu.com/Security/Features#null-mmap, Aug 2014.

[47] WebKit, "SunSpider 1.0.2 JavaScript Benchmark," https://www.webkit.org/perf/sunspider/sunspider.html, Aug 2014.

[48] W. Xu, D. C. DuVarney, and R. Sekar, "An efficient and backwards-compatible transformation to ensure memory safety of C programs," in *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2004.

[49] B. Zeng, G. Tan, and G. Morrisett, "Combining Control-flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing," in *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[50] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical Control Flow Integrity and Randomization for Binary Executables," in *IEEE Symposium on Security and Privacy (Oakland)*, 2013.

[51] M. Zhang and R. Sekar, "Control Flow Integrity for COTS Binaries," in *USENIX Security Symposium (Security)*, 2013.