SoK: Sanitizing for Security

Dokyung Song, Julian Lettner, Prabhu Rajasekaran,
Yeoul Na, Stijn Volckaert, Per Larsen, Michael Franz
University of California, Irvine
{dokyungs,jlettner,rajasekp,yeouln,stijnv,perl,franz}@uci.edu

***Abstract*—The C and C++ programming languages are notoriously insecure yet remain indispensable. Developers therefore resort to a multi-pronged approach to find security issues before adversaries. These include manual, static, and dynamic program analysis. Dynamic bug finding tools—henceforth “sanitizers”—can find bugs that elude other types of analysis because they observe the actual execution of a program, and can therefore directly observe incorrect program behavior as it happens.**

A vast number of sanitizers have been prototyped by academics and refined by practitioners. We provide a systematic overview of sanitizers with an emphasis on their role in finding security issues. Specifically, we taxonomize the available tools and the security vulnerabilities they cover, describe their performance and compatibility properties, and highlight various trade-offs.

I. Introduction

C and C++ remain the languages of choice for low-level systems software such as operating system kernels, runtime libraries, and browsers. A key reason is that they are efficient and leave the programmer in full control of the underlying hardware. On the flip side, the programmer must ensure that every memory access is valid, that no computation leads to undefined behavior, etc. In practice, programmers routinely fall short of meeting these responsibilities and introduce bugs that make the code vulnerable to exploitation.

At the same time, memory corruption exploits are getting more sophisticated [1]–[4], bypassing widely-deployed mitigations such as Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP). Code-reuse attacks such as Return-Oriented Programming (ROP) corrupt control data such as function pointers or return addresses to hijack the control-flow of the program [1]. Data-only attacks such as Data-Oriented Programming (DOP) leverage instructions that can be invoked on legal control-flow paths to compromise a program by corrupting only its non-control data [4].

As a first line of defense against bugs, programmers use analysis tools to identify security problems before their software is deployed in production. These tools rely on either static program analysis, dynamic program analysis, or a combination. Static tools analyze the program source code and produce results that are conservatively correct for all possible executions of the code [5]–[9]. In contrast, dynamic bug finding tools—often called “sanitizers”—analyze a single program execution and output a precise analysis result valid for a single run only.

Sanitizers are now in widespread use and responsible for many vulnerability discoveries. However, despite their ubiquity and critical role in finding vulnerabilities, sanitizers are often not well-understood, which hinders their further development and adoption. In fact, although there is a significant body of research in the area, only a few of them have seen adoption, leaving many types of vulnerabilities unsanitized. This paper provides a systematic overview of sanitizers with an emphasis on their role in finding security vulnerabilities. We taxonomize the available tools and the security vulnerabilities they cover, describe their performance and compatibility properties, and highlight various trade-offs. Based on our findings, we point to deployment directions for developers as well as research directions aimed at (i) finding vulnerabilities that elude existing tools, (ii) improving compatibility with real-world programs, and (iii) ways to find vulnerabilities more efficiently.

TABLE I
Exploit Mitigations vs. Sanitizers

	Exploit Mitigations	Sanitizers
The goal is to ...	Mitigate attacks	Find vulnerabilities
Used in ...	Production	Pre-release
Performance budget ...	Very limited	Much higher
Policy violations lead to ...	Program termination	Problem diagnosis
Violations triggered at location of bug ...	Sometimes	Always
Tolerance for FPs is ...	Zero	Somewhat higher
Surviving benign errors is ...	Desired	Not desired

The rest of the paper is organized as follows. We start with a high-level comparison of sanitizers and exploit mitigations (Section II). Next, we describe the low-level vulnerabilities in C/C++ (Section III) and taxonomize techniques to detect them (Section IV). We then continue with a description of two key implementation aspects of sanitizers: program instrumentation techniques (Section V) and metadata management (Section VI). We then briefly discuss how to drive the program being sanitized so as to maximize the effectiveness of the sanitizer (Section VII). Next, we present a summary of sanitizers that are being actively maintained or that were published at academic conferences with a focus on their precision, compatibility, and performance/memory costs (Section VIII). We also survey the deployment landscape of these tools (Section IX). We conclude the paper with future research directions (Section X).

II. Exploit Mitigations vs. Sanitizers

Sanitizers are similar to many well-known exploit mitigations insofar that they may instrument the program in similar ways, e.g., by inserting inlined reference monitors (IRMs). Despite such similarities, exploit mitigations and sanitizers significantly differ in their goals and use cases. We summarize key differences in Table I.

The biggest difference between the two types of tools lies in the type of security policy they enforce. Exploit mitigations deploy a policy aimed at detecting or preventing attacks, whereas sanitizers aim to pinpoint the precise locations of buggy program statements. Control-Flow Integrity (CFI) [10], [11], Data-Flow Integrity (DFI) [12] and Write Integrity Testing (WIT) [13] are examples of exploit mitigations because they detect deviations from legal control or data flow paths, which usually happen as a consequence of a bug's exploitation, but do not necessarily happen at the precise locations of vulnerable program statements. Bounds checking tools, in contrast, could be considered sanitizers because violations of their policies trigger directly at the locations of vulnerable statements.

Some tools selectively apply sanitization techniques, possibly combined with exploit mitigation techniques. Code-Pointer Integrity (CPI), for example, only performs bounds checks (a sanitization technique used in many sanitizers) when the program directly or indirectly accesses sensitive code pointers [14]. We therefore consider CPI an exploit mitigation rather than a sanitizer because CPI only detects a fraction of all bugs that could be detected using bounds checks.

Exploit mitigations are meant to be deployed in production, thus put stringent requirements on various design aspects. First, exploit mitigations rarely see real-world adoption if they incur non-negligible run-time overhead [15]. Sanitizers have less stringent performance requirements because they are only used for testing. Second, false positive detections in an exploit mitigations are unacceptable because they terminate the program. Sanitizers may tolerate false alerts to the extent that developers are willing to review false bug reports. Finally, surviving benign errors (e.g., writes to padding) is allowed and often desired in production systems for reliability and availability reasons, whereas sanitizers aim to detect these bugs precisely since their exploitability is unknown.

III. Low-level Vulnerabilities

Given the wide range of security-related bugs, we focus on bugs that have specific security implications in C/C++. This includes not only undefined behavior, but also well-defined behaviors that are potentially dangerous in the absence of type and memory safety. We briefly describe the bugs and how they can be exploited to leak information, escalate privilege, or execute arbitrary code.

A. Memory Safety Violations

A program is *memory safe* if pointers in the program only access their *intended referents*, while those intended referents are valid. The intended referent of a pointer is the object from whose base address the pointer was derived. Depending on the type of the referent, it is either valid between its allocation and deallocation (for heap-allocated referents), between a function call and its return (for stack-allocated referents), between the creation and the destruction of its associated thread (for thread-local referents), or indefinitely (for global referents).

Memory safety violations are among the most severe security vulnerabilities and have been studied extensively in the literature [15], [16]. Their exploitation can lead to code injection [17], control-flow hijacking [1], [18], [19], privilege escalation [20], information leakage [21], and program crashes.

1) Spatial Safety Violations: Accessing memory that is not (entirely) within the bounds of the intended referent of a pointer constitutes a spatial safety violation. Buffer overflows are a typical example of a spatial safety violation. A buffer overflow happens when the program writes beyond the end of a buffer. If the intended referent of a vulnerable access is a subobject (e.g., a struct field), and if an attacker writes to another subobject within the same object, then we refer to this as an *intra-object overflow*. Listing 1 shows an intra-object overflow vulnerability which can be exploited to perform a privilege escalation attack.

```
struct A { char name[7]; bool isAdmin; };
struct A a; char buf[8];
memcpy(/* dst */ a.name, /* src */ buf, sizeof(buf));
```

Listing 1. Intra-object overflow vulnerability which can be exploited to overwrite security-critical non-control data

2) Temporal Safety Violations: A temporal safety violation occurs when the program accesses a referent that is no longer valid. When an object becomes invalid, which usually happens by explicitly deallocating it, all the pointers pointing to that object become *dangling*. Accessing an object through a dangling pointer is called a *use-after-free*. Accessing a local object outside of its scope or after the function returns is referred to as *use-after-scope* and *use-after-return*, respectively. This type of bug becomes exploitable when the attacker can reuse and control the freed region, as illustrated in Listing 2.

```
struct A { void (*func)(void); };
struct A *p = (struct A *)malloc(sizeof(struct A));
free(p);   // Pointer becomes dangling
...
p->func(); // Use-after-free
```

Listing 2. Use-after-free vulnerability which can be exploited to hijack the control-flow of the program

B. Use of Uninitialized Variables

Variables have an *indeterminate value* until they are initialized [22], [23]. C++14 allows this indeterminate value to propagate to other variables if both the source and destination variables have an unsigned narrow character type. Any other use of an uninitialized variable results in undefined behavior. The effects of this undefined behavior depend on many factors, including the compiler and compiler flags that were used to compile the program. In most cases, indeterminate values are in fact the (partial) contents of previously deallocated variables that occupied the same memory range as the uninitialized variable. As these previously deallocated variables may sometimes hold security-sensitive values, reads of uninitialized memory may be part of an information leakage attack, as illustrated in Listing 3.

```
struct A { int data[2]; };
struct A *p = (struct A *)malloc(sizeof(struct A));
p->data[0] = 0; // Partial initialization
send_to_untrusted_client(p, sizeof(struct A));
```

Listing 3. Use of a partially-initialized variable which becomes vulnerable as the uninitialized value crosses a trust boundary

C. Pointer Type Errors

C and C++ support several casting operators and language constructs that can lead memory accesses to misinterpret the data stored in their referents, thereby violating type safety. Pointer type errors typically result from unsafe casts. C allows all casts between pointer types, as well as casts between integer and pointer types. The C++ `reinterpret_cast` type conversion operator is similarly not subject to any restrictions. The `static_cast` and `dynamic_cast` operators do have restrictions. `static_cast` forbids pointer to integer casts, and casting between pointers to objects that are unrelated by inheritance. However, it does allow casting of a pointer from a base class to a derived class (also called *downcasting*), as well as all casts from and to the `void*` type. *Bad-casting* (often referred to as *type confusion*) happens when a downcast pointer has neither the run-time type of its referent, nor one of the referent's ancestor types.

```
class Base { virtual void func(); };
class Derived : public Base { public: int extra; };
Base b[2];
Derived *d = static_cast<Derived *>(&b[0]); // Bad-casting
d->extra = ...; // Type-unsafe, out-of-bounds access, which
                // overwrites the vtable pointer of b[1]
```

Listing 4. Bad-casting vulnerability leading to a type- and memory-unsafe memory access

To downcast safely, programmers must use the `dynamic_cast` operator, which performs run-time type checks and returns a null pointer if the check fails. Using `dynamic_cast` is entirely optional, however, and introduces additional run-time overhead.

Type errors can also occur when casting between function pointer types. Again, C++'s `reinterpret_cast` and C impose no restrictions on casts between incompatible function pointer types. If a function is called indirectly through a function pointer of the wrong type, the target function might misinterpret its arguments, which leads to even more type errors. Finally, C also allows type punning through `union` types. If the program reads from a union through a different member object than the one that was used to store the data, the underlying memory may be misinterpreted. Furthermore, if the member object used for reading is larger than the member object used to store the data, then the upper bytes read from the union will take unspecified values.

D. Variadic Function Misuse

C/C++ support *variadic functions*, which accept a variable number of *variadic* function arguments in addition to a fixed number of regular function arguments. The variadic function's source code does not specify the number or types of these variadic arguments. Instead, the fixed arguments and the function semantics encode the expected number and types of variadic arguments. Variadic arguments can be accessed and simultaneously typecast using `va_arg`. It is, in general, impossible to statically verify that `va_arg` accesses a valid argument, or that it casts the argument to a valid type. This lack of static verification can lead to type errors, spatial memory safety violations, and uses of uninitialized values.

```
char *fmt2; // User-controlled format string
sprintf(fmt2, user_input, ...);
// prints attacker-chosen stack contents if fmt2 contains
// too many format specifiers
// or overwrites memory if fmt2 contains %n
printf(fmt2, ...);
```

Listing 5. Simplified version of CVE-2012-0809; user-provided input was mistakenly used as part of a larger format string passed to a printf-like function

E. Other Vulnerabilities

There are other operations that may pose security risks in the absence of type and memory safety. Notable examples include overflow errors which may be exploitable when such values are used in memory allocation or pointer arithmetic operations. If an attacker-controlled integer value is used to calculate a buffer size or an array index, the attacker could overflow that value to allocate a smaller buffer than expected (as illustrated in Listing 6), or to bypass existing array index checks, thereby triggering an out-of-bounds access.

```
// newsize can overflow depending on len
int newsize = oldsize + len + 100;
newsize *= 2;
// The new buffer may be smaller than len
buf = xmlRealloc(buf, newsize);
memcpy(buf + oldsize, string, len); // Out-of-bounds access
```

Listing 6. Simplified version of CVE-2017-5029; a signed integer overflow vulnerability that can lead to spatial memory safety violation

C/C++ do not define the result of a signed integer overflow, but stipulate that unsigned integers wrap around when they overflow. However, this wrap-around behavior is often unintended and potentially dangerous.

Undefined behaviors such as signed integer overflows pose additional security risks when compiler optimizations are enabled. In the presence of potential undefined behavior, compilers are allowed to assume that the program will never reach the conditions under which this undefined behavior is triggered. Moreover, the compiler can perform further optimization based on this assumption [24]. Consequently, the compiler does not have to statically verify that the program is free of potential undefined behavior, and the compiler is not obligated to generate code that is capable of recognizing or mitigating undefined behavior. The problem with this rationale is that optimizations based on the assumption that the program is free from undefined behavior can sometimes lead the compiler to omit security checks. In CVE-2009-1897, for example, GCC infamously omitted a null pointer check from one of the Linux kernel drivers, which led to a privilege escalation vulnerability [25]. Compiler developers regularly add such aggressive optimizations to their compilers. Some people therefore refer to undefined behavior as *time bombs* [26].

```
struct sock *sk = tun->sk; // Compiler assumes tun is not
                           // a null pointer
if (!tun) // Check is optimized out
  return POLLERR;
```

Listing 7. Simplified version of CVE-2009-1897; dereferencing a pointer lets the compiler safely assume that the pointer is non-null

IV. Bug Finding Techniques

We now review the relevant bug finding techniques. We begin each subsection with an informal description of the bug finding policy, followed by a description of mechanisms that implement (or approximate) that policy.

A. *Memory Safety Violations*

Memory safety bug finding tools detect dereferences of pointers that either do not target their intended referent (i.e., spatial safety violations), or that target a referent that is no longer valid (i.e., temporal safety violations). There are two types of tools for detecting these bugs. We summarize their high-level goals and properties here, and then proceed with an in-depth discussion of the techniques these tools can employ to detect memory safety bugs.

Location-based Access Checkers*:* Location-based access checkers detect memory accesses to invalid memory regions. These checkers have a metadata store that maintains state for each byte of (a portion) of the addressable address space, and consult this metadata store whenever the program attempts to access memory to determine whether the memory access is valid or not. Location-based access checkers can use red-zone insertion [27]–[31] or guard pages [32], [33] to detect spatial safety violations. Either of these techniques can be combined with reuse delay to additionally detect temporal safety violations [27]–[29], [31]–[36]. Location-based access checkers incur low run-time performance overheads, and are highly compatible with uninstrumented code. The downside is that these tools are imprecise, as they can only detect if an instruction accesses valid memory, but *not if the accessed memory is part of the intended referent of the instruction*. These tools generally incur high memory overhead.

Identity-based Access Checkers*:* Identity-based access checkers detect memory accesses that do not target their intended referent. These tools maintain metadata (e.g., bounds or allocation status) for each allocated memory object, and have a mechanism in place to determine the intended referent for every pointer in the program. Metadata lookups can happen when the program calculates a new pointer using arithmetic operations to determine if the calculation yields a valid pointer and/or upon pointer dereferences to determine if the dereference accesses the intended referent of the pointer. Identity-based access checkers can use per-object bounds tracking [34], [37]–[43] or per-pointer bounds tracking [44]–[55] to detect spatial safety violations, and can be extended with reuse delay [55], lock-and-key checking [46], [47], [56], or with dangling pointer tagging [57]–[60] to detect temporal safety violations. Identity-based checkers are more precise than location-based access checkers, as they cannot just detect accesses to invalid memory, but also accesses to valid memory outside of the intended referent. These tools do, however, incur higher run-time performance overhead than location-based checkers. Identity-based checkers are generally not compatible with uninstrumented code. They also have higher false positive detection rates than location-based checkers.

1) *Spatial Memory Safety Violations:*

Red-zone Insertion*:* Location-based access checkers can insert so-called *red-zones* between memory objects [27]–[31]. These red-zones represent out-of-bounds memory and are marked as invalid memory in the metadata store. Any access to a red-zone or to an unallocated memory region triggers a warning. Purify was the first tool to employ this technique [27]. Purify inserts the red-zones at the beginning and the end of each allocation. Purify tracks the state of the program's allocated address space using a large shadow memory bitmap that stores two bits of state per byte of memory. Valgrind's Memcheck uses the same technique but reserves two bits of state for every bit of memory [28]. Consequently, Memcheck can detect access errors with bit-level precision, rather than byte-level precision.

Light-weight Bounds Checking (LBC) similarly inserts red-zones, but adds a fast path to the location-based access checks to reduce the overhead of the metadata lookups [30]. LBC does this by filling the red-zones with a random pattern and compares the data read/overwritten by every memory access with the fill pattern. If the data does not match the fill pattern, the access is considered safe because it could not have targeted a red-zone. If the data does happen to match the fill pattern, LBC performs a secondary slow path check that looks up the state of the accessed data in the metadata store, and triggers a warning if the accessed data is a red-zone.

Location-based access checkers that use red-zone insertion generally incur low run-time performance overhead, but have limited precision as they can only detect illegal accesses that target a red-zone. Illegal accesses that target a valid object, which may or may not be part of the same allocation as the intended referent, cannot be detected. Red-zone insertion-based tools also fail to detect intra-object overflow bugs because they do not insert red-zones between subobjects. While technically feasible, inserting red-zones between subobjects would lead to excessive memory overhead and it would change the layout of the parent object. Any code that accesses the parent object or one of its subobjects would therefore have to be modified, which would also break compatibility with external code that is not aware of the altered data layout.

Guard Pages*:* Location-based access checkers can insert inaccessible guard pages before and/or after every allocated memory object [32], [33]. Out-of-bound reads and writes that access a guard page trigger a page fault, which in turn triggers an exception in the application. This use of the paging hardware to detect illegal accesses allows location-based access checkers to run without instrumenting individual load and store instructions. Using guard pages does, however, incur high memory overhead, making the technique impractical for applications with large working sets. Microsoft recognized this problem and added an option to surround memory objects with guard blocks instead of full guard pages in PageHeap [33]. PageHeap fills these guard blocks with a fill pattern, and verifies that the pattern is still present when a memory object is freed. This technique is strictly inferior to red-zone insertion, as it only detects out-of-bounds writes (and not reads), and it does not detect the illegal writes until the overwritten object is freed.

Per-pointer Bounds Tracking: Identity-based access checkers can store bounds metadata for every pointer [44]–[55]. Whenever the program creates a pointer by calling `malloc` or by taking the address of an object, the tracker stores the base and size of the referent as metadata for the new pointer. The tracker propagates this metadata when the program calculates new pointers through arithmetic and assignment operations. Spatial memory safety violations are detected by instrumenting all pointer dereferences and checking if a pointer is outside of its associated bounds when it is dereferenced.

Identity-based access checkers that use per-pointer bounds tracking can provide complete spatial memory violation detection, including detection of intra-object overflows. SoftBound [48] and Intel Pointer Checker [49] detect intra-object overflows by narrowing the pointer bounds to the bounds of the subobject whenever the program derives a pointer from the address of a subobject (i.e., a struct field). The primary disadvantage of per-pointer bounds tracking is poor compatibility, as the program generally cannot pass pointers to uninstrumented libraries because such libraries do not propagate or update bounds information correctly. Another disadvantage is that per-pointer metadata propagation adds high run-time overheads. CCured reduces this overhead by identifying "safe" pointers, which can be excluded from bounds checking and metadata propagation [50]. However, even with such optimizations, per-pointer bounds checking remains expensive without hardware support [61].

Per-object Bounds Tracking: Identity-based access checkers can also store bounds metadata for every memory object, rather than for every pointer [34], [37]–[43].

This approach—pioneered by Jones and Kelly (J&K)—solves some of the compatibility issues associated with per-pointer bounds tracking [34]. Per-object bounds trackers can maintain bounds metadata without instrumenting pointer creation and assignment operations. The tracker only needs to intercept calls to memory allocation (i.e., `malloc`) and deallocation (i.e., `free`) functions, which is possible even in programs that are not fully instrumented. Since bounds metadata is maintained only for objects and not for pointers, it is difficult to link pointers to their intended referent. While the intended referent of an in-bounds pointer can be found using a range-based lookup in the metadata store, such a lookup would not return the correct metadata for an out-of-bounds (OOB) pointer. J&K therefore proposed to instrument pointer arithmetic operations, and to invalidate pointers as they go OOB. Any subsequent dereference triggers a fault, which can then be caught to output a warning.

J&K's approach, however, breaks many existing programs that perform computations using OOB pointers. In light of this, CRED supports the creation and manipulation of OOB pointers by tracking their referent information [37]. CRED links OOB pointers to so-called *OOB objects* which store the address of the original referent for each OOB pointer.

Baggy Bounds Checking (BBC) eliminates the need to allocate dedicated OOB objects by storing the distance between the OOB pointer and its referent into the pointer's most significant bits [39]. Tagging the most significant bits also turns OOB pointers into invalid user-space pointers, such that dereferencing them causes a fault. BBC compresses the size of the per-object metadata by rounding up all allocation sizes to the nearest power of two, such that one byte of metadata suffices to store the bounds.

Low-fat pointer (LFP) bounds checkers improve BBC by making allocation sizes configurable, which results in lower performance and memory overheads [42], [43]. The idea is to partition the heap into equally-sized subheaps that each supports only one allocation size. Thus, the allocation size for any given pointer can be obtained by looking up the allocation size supported by that heap. The base address of the pointer's referent can be calculated by rounding it down to the allocation size. LFP also differs from BBC in handling of OOB pointers. For better compatibility with uninstrumented libraries, LFP does not manipulate the pointer representation to encode the referent of an OOB pointer. Instead, LFP recomputes the referent of each pointer whenever the pointer is given to a function as input either explicitly (e.g., a pointer given as an argument) or implicitly (e.g., a pointer loaded from memory). However, this requires LFP to enforce an invariant that pointers are within bounds whenever they are given as input to other functions, which is likely too strict for real-world programs (cf. Section VIII-A).

Per-object bounds trackers can support near-complete spatial safety vulnerability detection. However, techniques such as BBC and LFP do sacrifice precision for better run-time performance, as they round up allocation sizes and check allocation bounds rather than object bounds (cf. Section VIII-B).

Per-object bounds tracking has other downsides too. First, per-object bounds trackers do not detect intra-object overflows (cf. Section VIII-B). Second, marking pointers as OOB by pointing them to an OOB object, or by writing tags into their upper bits might impact compatibility with external code that is unaware of the bounds checking scheme used in the program. Specifically, external code is unable to restore OOB pointers into in-bounds pointers even when that is the intent.

2) Temporal Memory Safety Violations:

Reuse Delay: Location-based access checkers can mark recently deallocated objects as invalid in the metadata store by replacing them with red-zones [27]–[29], [31], [34] or with guard pages [32], [33], [35], [36]. Identity-based checkers can similarly invalidate the identity of deallocated objects [55]. The existing access checking mechanism can then detect dangling pointer dereferences as long as the deallocated memory or identity is not reused. If the program does reuse the memory or identity for new allocations, this approach will erroneously allow dangling pointer dereferences to proceed. Some reuse delay-based tools reduce the chance of such detection failures by delaying the reuse of memory regions or identities until they have "aged" [27]–[29], [31], [34], [55]. This leads to a trade-off between precision and memory overhead as longer reuse delays lead to higher memory overhead, but also to a higher chance of detecting dangling pointer dereferences.

Dhurjati and Adve (D&A) proposed to use static analysis to determine exactly when deallocated memory is safe to reuse [35]. D&A allocate every memory object on its own virtual memory page, but allow objects to share physical memory pages by aliasing virtual memory pages to the same physical page. When the program frees a memory object, D&A convert its virtual page into a guard page. D&A also partition the heap into pools, leveraging a static analysis called Automatic Pool Allocation [62]. This analysis can infer when a pool is no longer accessible (even through potentially dangling pointers), at which point all virtual pages in the pool can be reclaimed. Dang et al. proposed a similar system that does not use pool allocation, and can therefore be applied to source-less programs [36]. Similar to D&A, Dang et al. allocate all memory objects on their own virtual pages. Upon deallocation of an object, Dang et al. unmap that object's virtual page. This effectively achieves the same goal as guard pages, but allows the kernel to free its internal metadata for the virtual page, which reduces the physical memory overhead. To prevent reuse of unmapped virtual pages, Dang et al. propose to map new pages at the high water mark (i.e., the highest virtual address that has been used in the program). While this does not rule out reuse of unmapped virtual pages completely, the idea is that reuse is unlikely to happen given a 64-bit address space.

Lock-and-key: Identity-based checkers can detect temporal safety violations by assigning unique allocation identifiers—often called keys—to every allocated memory object and by storing this key in a *lock location* [46], [47], [56]. They also store the lock location and the expected key as metadata for every pointer. The checker revokes the key from the lock location when its associated object is deallocated. Lock-and-key checking detects temporal memory safety violations when the program dereferences a pointer whose key does not match the key stored in the lock location for that pointer. Assuming unique keys, this approach provides complete coverage of temporal safety violations [56]. Since this technique stores per-pointer metadata, it naturally complements identity-based checkers that also detect spatial violations using per-pointer bounds tracking. The drawbacks of lock-and-key checking are largely the same as those for per-pointer bounds tracking: compatibility with uninstrumented code is poor because uninstrumented code will not propagate the metadata correctly, and the run-time overhead is significant because maintaining metadata for every pointer is expensive.

Dangling Pointer Tagging: The most straightforward way to tag dangling pointers is to nullify either the value or the bounds associated with pointers that are passed to the `free` function [49]. A spatial memory safety violation detection mechanism would then trigger a warning if such pointers are dereferenced at a later point in time. The disadvantage of this approach is that it does not tag copies of the dangling pointer, which may also be used later.

Several tools tag not only the pointer passed to `free`, but also copies of that pointer by maintaining auxiliary data structures that link all memory objects to any pointers that refer to them [57]–[60]. Undangle uses taint tracking [63]–[65] to track pointer creations, and to maintain an object-to-pointer map [57]. Whenever the program deallocates a memory object, Undangle can query this pointer map to quickly find all dangling pointers to the now deallocated object. Undangle aims to report not only the use but also the existence of dangling pointers. It has a configurable time window where it considers dangling pointers latent but not unsafe, e.g., a transient dangling pointer that appears during the deallocation of nested objects. Undangle reports a dangling pointer when this window expires, or earlier if the program attempts to dereference the pointer.

DangNull [58], FreeSentry [59], and DangSan [60] steer clear of taint tracking and instrument pointer creations at compile time instead. These tools maintain pointer maps by calling a runtime registration function whenever the program assigns a pointer. Whenever the program deallocates a memory object, the tools look up all pointers that point to the object being deallocated, and invalidate them. Subsequent dereferences of an invalidated dangling pointer result in a hardware trap.

Dangling pointer tagging tools that are not based on taint tracking have some fundamental limitations. First, they require the availability of source as it relies on precise type information to determine which operations store new pointers. Second, they fail to maintain accurate metadata if the program copies pointers in a type-unsafe manner (e.g., by casting them to integers). Third, and most importantly, they can only link objects to pointers stored in memory, and is therefore unaware of dangling pointers stored in registers. Taint tracking-based tools such as Undangle, have none of these disadvantages, but incur significant performance and memory overheads.

B. *Use of Uninitialized Variables*

These tools detect uses of uninitialized values.

Uninitialized Memory Read Detection: Location-based access checkers can be extended to detect reads of uninitialized memory values by marking all memory regions occupied by newly allocated objects as uninitialized in the metadata store [27]. These tools instrument read instructions to trigger a warning if they read from uninitialized memory regions, and they instrument writes to clear the uninitialized flag for the overwritten region. Note that marking memory regions as uninitialized is not equivalent to marking them as a red-zone, since both read and write accesses to red-zones should trigger a warning, whereas accesses to uninitialized memory should only be disallowed for reads.

Uninitialized Value Use Detection: Detecting reads of uninitialized memory yields many false positive detections, as the C++14 standard explicitly allows uninitialized values to propagate through the program as long as they are not used. This happens, for example, when copying a partially uninitialized struct from one location to the other. Memcheck attempts to detect only the uses of uninitialized values by limiting error reporting to (i) dereferences of pointers that are (partially) undefined, (ii) branching on a (partially) undefined value, (iii) passing undefined values to system calls, and (iv) copying uninitialized values into floating point registers [28]. To support this policy, Memcheck adds one byte of shadow state for every partially initialized byte in the program memory.

This allows Memcheck to track the definedness of all of the program's memory with bit-level precision. Memcheck approximates the C++14 semantics but produces false negatives (failing to report illegitimate uses of uninitialized memory) and false positives (reporting legitimate uses of uninitialized memory), which are unavoidable given that Memcheck operates at the binary level, rather than the source code level. MemorySanitizer (MSan) implements fundamentally the same policy as Memcheck, but instruments programs at the compiler Intermediate Representation (IR) level [66]. The IR code carries more information than binary code, which makes MSan more precise than Memcheck. MSan produces no false positives (provided that the entire program is instrumented) and few false negatives. Its performance overhead is also an order of magnitude lower than Memcheck.

C. Pointer Type Errors

These tools detect bad-casting or dereferencing of pointers of incompatible types.

Pointer Casting Monitor: Pointer casting monitors detect illegal downcasts through the C++ `static_cast` operator. Illegal downcasts occur when the target type of the cast is not equal to the run-time type (or one of its ancestor types) of the source object. UndefinedBehaviorSanitizer [67] (UBSan) and Clang CFI [68] include checkers that verify the correctness of `static_cast` operations by comparing the target type to the run-time type information (RTTI) associated with the source object. This effectively turns `static_cast` operations into `dynamic_cast`. The downside is that RTTI-based tools cannot verify casts between non-polymorphic types that lack RTTI.

CaVer [69] and TypeSan [70] do not rely on RTTI to track type information, but instead maintain metadata for all types and all objects used in the program. This way, they can extend the type-checking coverage to non-polymorphic types. At compile time, these tools build per-class type metadata tables which contain all the valid type casts for a given pointer type. The type tables encode the class inheritance and composition relationships. Both tools also track the effective run-time types for each live object by monitoring memory allocations and storing the allocated types in a metadata store. To perform downcast checking, the tools retrieve the run-time type of the source object from the metadata store, and then query the type table for the corresponding class to check if the type conversion is in the table (and is therefore permissible). HexType similarly tracks type information in disjoint metadata structures, but provides a more accurate run-time type tracing [71]. HexType also replaces the default implementation of `dynamic_cast` with its own optimized implementation, while preserving its run-time semantics, i.e., returning NULL for failing casts.

Pointer Use Monitor: C/C++ support several constructs to convert pointer types in potentially dangerous ways; C-style casts, `reinterpret_cast`, and unions can all be used to bypass compile-time and run-time type checking. Extending pointer casting monitoring to these constructs can result in false positives, however. This is because programmers can legitimately use such constructs as the language standard allows it. For this reason, one might opt for pointer dereference/use monitoring over pointer casting monitoring.

Loginov et al. proposed a pointer use monitor for C programs [72]. The tool maintains and verifies run-time type tags for each memory location by monitoring load and store operations. A tag contains the scalar type that was last used to write to its corresponding memory location. Aggregate types are supported by breaking them down into their scalar components. The tool stores the tags in shadow memory. Whenever a value is read from memory, the tool checks if the type used to load the value matches the type tag.

LLVM Type Sanitizer (TySan) also maintains a type tag store in shadow memory and verifies the correctness of load instructions [73]. Contrary to Loginov et al.'s tool, however, TySan does not require that the types used to store and load from a memory location match exactly. Instead, TySan only requires type compatibility, as defined by the aliasing rule in the C/C++ standard. TySan leverages metadata generated by the compiler frontend (Clang) which contains the aliasing relationship between types. This metadata is used at run time to allow, for example, all loads through a character pointer type, even if the target location was stored using a pointer to a larger type. Loginov et al.'s tool would detect this as an error, but this behavior is explicitly permitted by the language standard.

EffectiveSan is another pointer use monitor that performs type checks as well as bounds checks at pointer uses [74]. EffectiveSan instruments each allocation site to tag each allocated object with its statically-determined type. It uses declared variable types for stack and global variables, as well as objects allocated using the C++ `new` operator. For objects allocated using `malloc`, it uses the type of the first lvalue usage of the object. EffectiveSan also generates type layout metadata at compile time, which contains the layout information of all nested subobjects for each type. At every pointer dereference, both type compatibility and object bounds are checked, using the object type tag in conjunction with the type layout metadata. EffectiveSan's bounds checking supports detection of intra-object overflows by using type layout information to derive subobject bounds at run time.

Several tools also detect pointer type errors in indirect function calls, that is, calling functions through a pointer of a type incompatible with the type of the callee [67], [68], [75]. Function-signature-based forward-edge control flow integrity mechanisms such as Clang CFI [68] can be viewed as sanitizers that detect such function pointer misuses. Since all the function signatures are known at compile time, these tools can detect mismatches between the pointer type and the function type without maintaining run-time tags.

D. Variadic Function Misuse

These tools detect memory safety violations and uninitialized variable uses specific to variadic functions.

Dangerous Format String Detection: The most prominent class of variadic function misuse bugs are format string vulnerabilities. Most efforts therefore focus solely on detection of dangerous calls to `printf`. Among these efforts are tools that restrict the usage of the `%n` qualifier in the format string [76], [77]. This qualifier may be used to have `printf` write to a caller-specified location in memory. However, this dangerous operation [2] is specific to the `printf` function, so the aforementioned tools' applicability is limited.

***Argument Mismatch Detection**:* FormatGuard prevents `printf` from reading more arguments than were passed by the caller [78]. FormatGuard does this by redirecting the calls to a protected `printf` implementation that increments a counter whenever it retrieves a variadic argument through `va_arg`. If the counter surpasses the number of arguments specified at the call site, FormatGuard raises an alert. HexVASAN generalizes argument counting to all variadic functions, and also adds type checking [79]. HexVASAN instruments the call sites of variadic functions to capture the number and types of arguments passed to the callee and saves this information in a metadata store. The tool then instruments `va_start` and `va_copy` operations to retrieve information from the metadata store, and it instruments `va_arg` operations to check if the argument being accessed is within the given number of arguments and of the given type.

E. Other Vulnerabilities

These tools detect other undefined behavior or well-defined but potentially unintended and dangerous behavior.

***Stateless Monitoring**:* UndefinedBehaviorSanitizer (UBSan) is a dynamic tool that detects undefined behavior we have not covered so far [67]. The undefined behaviors UBSan detects currently include signed integer overflows, floating point or integer division by zero, invalid bitwise shift operations, floating point overflows caused by casting (e.g., casting a large double-precision floating point number to a single-precision float), uses of misaligned pointers, performing an arithmetic operation that overflows a pointer, dereferencing a null pointer, and reaching the end of a value-returning function without returning a value. Most of UBSan's detection features are stateless, so they can be turned on collectively without interfering with each other. UBSan can also detect several kinds of well-defined but likely unintended behavior. For example, the language standard dictates that unsigned integers wrap around when they overflow. This well-defined behavior is often unexpected and a frequent source of bugs, however, so UBSan can optionally detect these unsigned integer wrap-arounds.

V. Program Instrumentation

Sanitizers implement their bug finding policy by embedding inlined reference monitors (IRMs) into the program. These IRMs monitor and mediate all program instructions that can contribute to a vulnerability. Such instructions include (but are not limited to) memory loads and stores, stack frame (de)allocations, calls to memory allocation functions (e.g., `malloc`), and system calls. IRMs can be embedded using a compiler, linker, or an instrumentation framework.

A. Language-level Instrumentation

Sanitizers can be embedded at the source code or abstract syntax tree (AST) level. The source code and AST are language-specific and typically contain full type information, language-specific syntax, and compile time-evaluated expressions such as `const_cast` and `static_cast` type conversions. This language-specific information is typically discarded when the compiler lowers the AST into Intermediate Representation (IR) code. Language-level instrumentation is recommended (or even necessary) for sanitizers that detect pointer type errors through pointer cast monitoring.

An additional advantage of instrumenting at the language level is that the compiler preserves the full semantics of the program throughout the early stages of the compilation. The sanitizer can therefore see the semantics intended by the programmer. At later stages in the compilation, the compiler may assume that the program contains no undefined behavior, and it may optimize the code based on this assumption (e.g., by eliminating seemingly unnecessary security checks). The disadvantage of instrumenting at the language level is that the entire source code of the application must be available and the code must be written in the expected language. Thus, this approach does not work for applications that link against closed-source libraries, nor does it work for applications that contain inline assembly code [80].

B. IR-level Instrumentation

Sanitizers can also be embedded later stage in the compilation, when the AST has been lowered into IR code. Compiler backends such as LLVM support IR-level instrumentation [81]. This approach is more generic than source-level transformation in that the compiler IR is (mostly) independent of the source language. Thus, by instrumenting at this level, the sanitizer can automatically support multiple source languages. An additional advantage is that the compiler backend implements various static analyses and optimization passes that can be used by the sanitizer. Sanitizers can leverage this infrastructure to optimize the IRMs they embed into the program (e.g., by removing redundant or provably safe checks).

The disadvantage of IR-level instrumentation is largely similar to that of language-level instrumentation, i.e., the lack of support for closed-source libraries and inline assembly code (Section V-A). Exceptionally, AddressSanitizer (ASan) does provide limited support for inline x86 assembly code by instrumenting `MOV` and `MOVAPS` instructions found in inline assembly blocks [31]. This approach is architecture-specific, however, and needs to be reimplemented or duplicated for every supported architecture.

C. Binary Instrumentation

Dynamic binary translation (DBT) frameworks allow instrumentation of a program at run time [82]–[84]. They read program code, instrument it, and translate it to machine code while the program executes and expose various hooks to influence execution. The main advantage of DBT-based tools over compiler-based tools is that they work well for closed-source programs. Moreover, DBT frameworks offer complete instrumentation coverage of user-mode code regardless of its origin. DBT frameworks can instrument the program itself, third party code (that may be dynamically loaded), and even dynamically generated code.

The main disadvantage of DBT is the (much) higher run-time performance overhead compared to static instrumentation tools (see Section VIII-E). This overhead can be primarily attributed to run-time decoding and translation of instructions. This problem can be partially addressed by instrumenting binaries statically using a Static Binary Instrumentation (SBI) framework. However, both SBI and DBT-based sanitizers must operate on binaries that contain virtually no type information or language-specific syntax. It is therefore impossible to embed a pointer type error sanitizer at this stage. Information about stack frame and global data section layouts is also lost at the binary level, which makes it impossible to insert a fully precise spatial memory safety sanitizer using binary instrumentation.

D. Library Interposition

An alternative, albeit very coarse-grained, method is to intercept calls to library functions using library interposers [85]. A library interposer is a shared library that, when preloaded into a program [86], can intercept, monitor, and manipulate all inter-library function calls in the program. Some sanitizers use this method to intercept calls to memory allocation functions such as `malloc` and `free`.

The advantage of this approach is that, similarly to DBT-based instrumentation, it works well for COTS binaries in that no source or object code is required. Contrary to DBT, however, library interposition incurs virtually no overhead. One disadvantage is that library interposition only works for *inter*-library calls. A call between two functions in the same library cannot be intercepted. Another disadvantage is that library interposition is highly platform and target-specific. A sanitizer that uses library interposition to intercept calls to `malloc` would not work for programs that have their own memory allocator, for example.

VI. Metadata Management

One important aspect of a sanitizer design is how it stores and looks up metadata. This metadata typically captures information about the state of pointers or memory objects used in the program. Although run-time performance is not a primary concern for sanitizer developers or users, the sheer quantity of metadata most sanitizers store means that even small inefficiencies in the storage scheme can make the sanitizer unacceptably slow. The metadata storage scheme also by and large determines whether two sanitizers can be used in conjunction. If two independent sanitizers both use a metadata scheme that changes the pointer and/or object representation in the program, they often cannot be used together.

A. Object Metadata

Some sanitizers use object metadata storage schemes to store state for all allocated memory objects. This state may include the object size, type, state (e.g., allocated/deallocated, initialized/uninitialized), allocation identifier, etc.

Embedded Metadata: An obvious way to store metadata for an object is to increase its allocation size and to append or prepend the metadata to the object's data. This mechanism is popular among modern memory allocators which, for example, store the length of a buffer in front of the actual buffer. Tools can modify memory allocators to transparently reserve memory for metadata in addition to the requested buffer size, and then return a pointer into the middle of this allocation so that the metadata is invisible to clients. Allocation-embedded metadata is used in ASan [31] and Oscar [36], among others. ASan embeds information about the allocation context in each allocated object. Oscar stores each object's canonical address as embedded metadata.

Direct-mapped Shadow: The direct-mapped shadow scheme maps every block of n bytes in the application's memory to a block of m bytes of metadata via the formula:

```
// shadow_base is the base address of the metadata store
// block_addr is the address of the memory block
metadata_addr = shadow_base + m * (block_addr / n)
```

ASan [31], for example, stores 1 byte of metadata for every 8 bytes of application memory. In this case, the shadow mapping formula can be simplified to:

```
metadata_addr = shadow_base + (block_addr >> 3)
```

The direct-mapped shadow scheme is easy to implement and insert into an application. It is generally also very efficient since only one memory read is needed to retrieve the metadata for any given object. There are cases where it can lead to poor run-time performance too, however, as it can worsen memory fragmentation (and thus spatial locality) in already fragmented address spaces. It is also wasteful in terms of allocated memory, since the shadow memory area is contiguous and must be big enough to mirror all allocated memory blocks (from the lowest virtual addresses to the highest).

Multi-level Shadow: The multi-level shadow scheme can reduce the memory footprint of metadata store by introducing additional layers of indirection in the form of directory tables. These directory tables can store pointers to metadata tables or other directory tables. Each metadata table directly mirrors a portion of the application memory, similar to the direct-mapped shadow scheme. As a whole, the multi-level shadow scheme resembles how modern operating systems implement page tables. Having additional layers of indirection allows metadata stores to allocate metadata tables on demand. They only have to allocate the directory tables themselves, and can defer allocation of the metadata tables until they are needed. This is particularly useful for systems that have limited address space (e.g., 32-bit systems), where sanitizers that implement direct-mapped shadow schemes (e.g., ASan [31]) often exhaust the available address space and cause program termination.

Tools that require per-object metadata (in contrast to per-byte metadata) can use a *variable-compression-ratio* multi-level shadow mapping scheme, where the directory table maps variable-sized objects to constant-sized metadata. This scheme can help the tools to optimize their shadow memory usage and allocation-time performance [87].

The main characteristic of this scheme is that each metadata access requires multiple memory accesses: one for each level of directory tables and another one for the corresponding metadata table. This significantly affects performance, especially for tools that frequently look up metadata, e.g., a bounds checking tool which requires metadata access for most memory accesses. TypeSan [70], for example, is a good fit for the two-level variable-compression-ratio scheme, as the type metadata is per-object and constant-sized and metadata lookup is infrequent.

Custom Data Structure*:* In addition to variations of the previously presented metadata schemes, some tool authors have opted for a range of custom data structures and tool-specific solutions to store metadata. Bounds checkers such as J&K, CRED, and D&A employ splay trees [34], [37], [38]. UBSan and CaVer use an additional hash table as a cache to store the most recent results of type checking [67], [69]. DangNull utilizes a thread-safe red-black tree to encode the relationship between objects [58]. Note that, when using a data structure without support for concurrent access it must be protected by explicit locks in a multi-threaded setting. For thread-local or stack variables, per-thread metadata is also a choice, e.g., CaVer's per-thread red-black tree for stack and global objects.

B. Pointer Metadata

Fat Pointers*:* Some sanitizers replace standard machine pointers with fat pointers. Fat pointers are structures that contain the original pointer value, as well as metadata associated with the original pointer. A fairly straightforward fat pointer layout, used in many per-pointer bounds tracking tools is:

```
struct fat_pointer {
  void* value; // Original pointer value
  void* base;  // Base address of the intended referent
  size_t size; // Size of the referent
};
```

The primary advantage of using fat pointers is that they do not add much additional cache pressure compared to regular pointers, and that they can store an arbitrary amount of metadata. The disadvantages are that they require extensive instrumentation of the program, they change the calling conventions for functions that accept pointer arguments (fat pointers occupy more than one register when passed as a function argument), and that they cannot be used in programs that interact with uninstrumented third-party libraries. Without instrumentation, these libraries do not interpret the fat pointer correctly, nor are they able to update the fat pointer when its embedded inner pointer value changes.

Tagged Pointers*:* A less invasive way to store per-pointer metadata is to replace regular machine pointers with tagged pointers. A tagged pointer embeds metadata in the pointer itself, without changing its size. This technique provides better compatibility than fat pointers. Passing tagged pointers as function arguments does not require changes to the standard calling conventions, for example. Another advantage is that tagged pointers do not introduce any additional cache pressure compared to regular machine pointers. Tools that rely on tagged pointers generally store the tag in the unused bits of the original pointer. The amount of information that a tagged pointer can encode is therefore limited by size of the unused address space for a given target platform. Most AMD64 platforms, including Linux/x86_64, for example, only use the lowest 256TiB of virtual address space for user-mode applications. The upper 16 bits of any valid user-mode pointer are therefore guaranteed to be zero. These 16 bits can store per-pointer metadata. Baggy Bounds Checking uses the spare bits to store the distance between an OOB pointer and its intended referent [39]. In 32-bit SGX enclaves running on 64-bit processors, the upper 32 bits of any pointer are zero. SGXBounds uses these 32 bits to store the upper bound of the pointer's referent [54].

CUP goes one step further and uses the entire pointer width to store tags, thus discarding the original pointer value altogether [55]. One of the tags CUP stores in the tagged pointer is an offset into a metadata table that contains the original pointer value.

Note that tagged pointers usually cannot be dereferenced directly. CUP needs to retrieve the original pointer value of a tagged pointer before it can be dereferenced, while most other tools need to mask out the tag(s) prior to dereferencing a tagged pointer. This is also necessary for tagged pointers that escape to external uninstrumented libraries. The one exception is low-fat pointer-based tools, which store the tag implicitly in the pointer value, and can be dereferenced directly [42], [43].

Disjoint Metadata*:* Storing metadata in a disjoint metadata store instead of embedding it in the pointer representation improves compatibility over the aforementioned approaches. In contrast to per-object metadata, however, sanitizers usually do not use direct-mapped shadow stores to maintain per-pointer metadata (cf. Section VI-A). The portion of memory occupied by pointers is usually small and the size of pointer metadata (e.g., bounds) tends to exceed the size of the pointer itself, resulting in wasteful consumption of address space. For this reason, even at the cost of additional memory accesses, tool authors have favored multi-level structures for maintaining per-pointer metadata. CETS utilizes a two-level lookup trie (similar to page tables) using the pointer location as the key to store the allocation identifier and the lock address of the referent [56]. Intel Pointer Checker [49] and Intel MPX [88] also use a two-level structure to maintain pointer bounds.

The main disadvantage of disjoint metadata compared to in-pointer metadata is that the sanitizer must explicitly propagate the metadata whenever the program copies a pointer to a new memory location. If the program calls `memcpy` to copy a data structure containing pointers, for example, then the sanitizer must update the metadata store for the pointers in the target data structure. With in-pointer metadata, by contrast, the metadata always travels with the pointer.

C. Static Metadata

Some sanitizers require certain information discarded by the compiler to perform their checks at run time. To make the required compile-time information available at run time, these sanitizers usually embed static metadata into the compiled program. For example, bad-casting sanitizers create a type hierarchy table at compile time to facilitate type casting checks at run time. HexVASAN, a variadic function call sanitizer, builds static metadata for each variadic call site to encode the number of arguments and their types. At run time, the instrumented caller pushes the static metadata onto a custom stack, which is used by the callee to check the validity of the supplied arguments.

VII. Driving a Sanitizer

Dynamic analysis tools, including sanitizers, only detect bugs on code paths that execute during testing. Increasing path coverage therefore increases bug finding opportunities. Program execution can be driven by unit or integration test suites, automated test case generators, alpha and beta testers, or any combination thereof.

Unit testing and integration testing are already best practices in software engineering. Writing these tests has traditionally been a manual process. While indispensable in general, using hand-written tests does have some drawbacks when used to sanitize a program. First, manually written tests often focus on positive testing using *valid* inputs to check expected behavior. Security bugs are typically exploited by feeding the program *invalid* inputs, however. Second, manually written tests hardly ever cover *all* code paths.

Developers can use automated test case generators to alleviate these problems. One option is to use symbolic execution, which systematically explores all possible execution paths to generate concrete program inputs [6]–[9]. These inputs can then be fed into sanitized programs to find bugs. However, this approach in general does not scale due to the path explosion problem and the cost of constraint solving. A more scalable option is to run a fuzzer on the program being sanitized [89]–[94]. Fuzzers are testing tools that run programs on automatically generated inputs, typically using light-weight dynamic program analysis such as coverage feedback. Fuzzers perform negative testing, because they tend to provide invalid inputs to the program, and can find security bugs relatively quickly, especially if the bugs are triggered on code paths that are easily accessible.

Finally, developers can ship sanitization-enabled programs to beta testers and to collect and transmit any sanitizer output back to the developer. The main advantage here is that beta testers can distribute the testing load, therefore allowing developers to locate bugs more quickly. One disadvantage is that beta testers will inadvertently focus on testing the program's main usage scenarios. Another disadvantage is that sanitizers can slow down the program to the point where it becomes unusable, thus reducing the chance that beta testers will thoroughly test the programs. Lettner et al. [95] demonstrated that *partitioned sanitization*, where sanitization is turned on and off at run time based on criteria such as coverage and execution speed, can alleviate this concern to the extent that sanitizers compose.

VIII. Analysis

Table II summarizes the features of sanitizers that are either being actively maintained (as open source projects or commercial products), or that were published at academic conferences. Some of the tools we included were originally designed as an exploit mitigation, and therefore do not have a built-in error reporting mechanism. However, these tools do fit our definition of a sanitizer (cf. Section II) as they can pinpoint the exact location of the vulnerable code, and they can provide useful feedback if used in conjunction with a debugger. We excluded Intel Inspector [96], ParaSoft Insure++ [75], Micro Focus DevPartner [97], and UNICOM Global PurifyPlus [98], because the lack of public information about these sanitizers does not permit an accurate comparison.

For every sanitizer, the table shows which bugs it finds, which techniques it uses to find those bugs, and which metadata storage scheme (if any) it uses. The pie marks represent our assessment of how effective the sanitizer is, and how efficient it is in terms of run-time and memory overheads. A colored cell indicates that the sanitizer is known to produce false positives. We discuss the reasons for these false positives in Section VIII-A. We verified the reported performance numbers for 10 of these tools (i.e., those that have their performance overhead cells in Table II marked with an asterisk) by running them on the same experimental platform using the same set of benchmarks. We report the exact performance numbers for these tools in Appendix A.

A. False Positives

The practicality of a sanitizer primarily depends on how accurately it reports bugs. A developer that uses a sanitizer wants to minimize the time spent on reviewing its bug reports. The most desirable property for a sanitizer is therefore that it reports no false positives (i.e., all bugs it reports are truly bugs), while false negatives (i.e., the sanitizer finds all possible bugs) are a secondary concern. We identified the following recurring problems that can lead to false positive detections.

The most frequently recurring problem is that sanitizers often implement a bug finding policy or mechanism which is stricter than either the language standard or the de facto standard. The de facto standard includes widely-followed programming practices that do not necessarily comply with the language standard, even though they result in bug-free code [99]. One could therefore argue that reporting behavior that does not comply with the de facto standard as a bug constitutes a false positive detection.

Older per-object bounds trackers such as J&K disallow the creation of OOB pointers that point beyond the end of an array [34]. This design decision is compatible with the language standard, but not with the de facto standard. Creating OOB pointers is common in real-world programs, and does not lead to problems unless the program dereferences the OOB pointer. Subsequent per-object bounds trackers such as CRED allow programs to create OOB pointers.

Programs that store temporarily OOB pointers can also cause problems for dangling pointer checkers [58]–[60]. These tools

Sanitizers			IV. Bug Finding Techniques													V. Instr.				VI. Metadata Mgmt.							VIII. Analysis										
	Year Published	Actively Maintained	Red-zone Insertion	Guard Pages	Reuse Delay	Per-pointer Bounds Tracking	Per-object Bounds Tracking	Lock-and-key	Dangling Pointer Tagging	Uninit. Mem. Read Detection	Uninit. Value Use Detection	Pointer Casting Monitor	Pointer Use Monitor	Variadic Arg. Mismatch Detection	Stateless Monitoring	Language-level Instr.	IR-level Instr.	Binary Instr.	Library Interposition	Embedded Metadata	Direct-mapped Shadow	Multi-level Shadow	Custom Data Structure	Fat/Tagged Pointers	Disjoint Per-pointer Metadata	Static Metadata	Spatial Safety Violation (III-A1)	Temporal Safety Violation (III-A2)	Use of Uninit. Variables (III-B)	Bad-casting (III-C)	Load Type Mismatch (III-C)	Func. Call Type Mismatch (III-C)	Variadic Func. Misuse (III-D)	Signed Integer Overflow (III-E)	Other Undef. Behavior (III-E)	Performance Overhead	Memory Overhead
Purify [27]	’92		✔		✔					✔								✔			✔						◔	◑	● 🟥							◕	●
Memcheck [28]	’05	✔	✔		✔						✔							✔				✔					◔	◑	◑							○*	n/a
Dr. Memory [29]	’11	✔	✔		✔						✔							✔				✔					◔	◑	◑							○	n/a
LBC [30]	’12		✔													✔						✔					◕									◕	◑
ASan [31]	’12	✔	✔		✔												✔				✔						◕	◑								◕*	◑
Electric Fence [32]	’93			✔	✔														✔								◔	◕								n/a	○
PageHeap [33]	’00			✔	✔														✔								◔	◕								n/a	○
D&A Dangling [35]	’06			✔	✔												✔											◕								◕	n/a
Oscar [36]	’17			✔	✔														✔	✔								◕								◕	●
RTCC [45]	’92					✔										✔								✔			◕									◑	n/a
Safe-C [46]	’94					✔		✔								✔								✔	✔		● 🟥	● 🟥								◑	●
P&F [47]	’97					✔		✔								✔									✔		● 🟥	● 🟥								◑	●
MSCC [52]	’04					✔		✔								✔									✔		◕ 🟥	◕ 🟥								◕	●
SoftBound+CETS [48], [56]	’10					✔		✔									✔								✔		● 🟥	● 🟥								◕	●
Intel Pointer Checker [49]	’12	✔				✔			✔								✔								✔		● 🟥	◔								◕	●
SGXBounds [54]	’17					✔											✔			✔				✔			◕ 🟥									◕	●
CUP [55]	’18				✔	✔											✔							✔	✔		◕ 🟥	◑ 🟥								◕	n/a
J&K [34]	’97						✔									✔							✔				◕ 🟥									◑	n/a
CRED [37]	’04						✔									✔							✔				◕ 🟥									◕	n/a
D&A Bounds [38]	’06						✔										✔						✔				◕ 🟥									◕	n/a
BBC [39]	’09						✔										✔							✔			◑ 🟥									◕	●
PAriCheck [41]	’10						✔									✔							✔				◑ 🟥									◕	n/a
Low-fat Pointer [42], [43]	’17	✔					✔										✔							✔			◑ 🟥									◕*	●
Undangle [57]	’12								✔									✔							✔			◕								○	○
FreeSentry [59]	’15								✔							✔						✔	✔					◕ 🟥								◕	n/a
DangNull [58]	’15								✔								✔						✔					◑ 🟥								◕	●
DangSan [60]	’17								✔								✔					✔	✔					◑ 🟥								◕*	●
MSan [66]	’15	✔									✔						✔				✔								◕ 🟥							◕*	●
CaVer [69]	’15											✔				✔	✔						✔			✔				◑ 🟥						◕	●
TypeSan [70]	’16											✔				✔	✔					✔				✔				◑ 🟥						◕*	●
HexType [71]	’17											✔				✔	✔						✔			✔				◕						●*	●
Loginov et al. [72]	’01												✔			✔					✔										◑ 🟥					○	n/a
LLVM TySan [73]	’17	✔											✔			✔	✔				✔										◕					n/a	◑
EffectiveSan [74]	’18												✔			✔	✔			✔				✔		✔	● 🟥	◑			◕ 🟥					◑	●
Clang CFI [68]	’15	✔										✔	✔			✔	✔			✔						✔				◔		●				●*	●
HexVASAN [79]	’17													✔			✔						✔			✔							● 🟥			●*	●
UBSan [67]	’12	✔										✔	✔		✔	✔				✔						✔				◔		●		●	◑	◕*	●

TABLE II
OVERVIEW OF SANITIZERS

SPATIAL SAFETY VIOLATION
◔ No stack/global var. overflow detection
◑ No overflow to padding detection
◕ No intra-object overflow detection

TEMPORAL SAFETY VIOLATION
◔ Dangling pointer identified at compile time
◑ No protection for pointers to reused memory and/or register-stored pointers
◕ No protection for pointers to stack variables and/or integer-typed pointers

USE OF UNINITIALIZED VARIABLES
◑ Fair coverage
◕ Good coverage

BAD-CASTING
◔ Polymorphic class support only
◑ Non-polymorphic class support
◕ Better but incomplete run-time type tracing

LOAD TYPE MISMATCH
◑ Scalar type granularity
◕ Incomplete run-time type tracing

OTHER UNDEFINED BEHAVIOR
◑ Partial coverage

🟥 Known false positives

PERFORMANCE OVERHEAD
○ Over 10x
◑ Up to 10x
◕ Up to 3x
● Up to 10%
●* Verified (see Appendix A)

MEMORY OVERHEAD
○ Over 10x
◑ 3x to 10x
● 1x to 3x

n/a Data not available

do not recognize the intended referent of an OOB pointer and therefore fail to register the pointer to the correct object. If the intended referent gets deallocated, the dangling pointer checker will not invalidate the OOB pointer. Worse, if the temporarily

OOB pointer happened to point to a valid object (different from its intended referent) when it was registered, then the pointer checker will erroneously invalidate the pointer when the program deallocates that different object.

Pointers that temporarily go OOB can also cause problems in low-fat pointer-based bounds checkers [42], [43], which perform bounds checks whenever a pointer escapes from a function. If an OOB pointer is passed to a function that converts that pointer back to its original in-bounds value, the checker will have raised a false positive warning.

Uninitialized memory read detectors raise warnings when the program reads uninitialized memory [27]. The language standard allows this, as long as the uninitialized values are not used in the program.

Some pointer type error checkers fail to capture the effective type of an object under certain circumstances [69], [70]. For example, if a memory region is repurposed using placement new in C++, these checkers may fail to update or invalidate the type metadata associated with that region. This can lead to false positive detections when the old type is used for type checking.

Several tools do not respect the language standard's aliasing rule. Loginov et al.'s pointer use monitor requires that the stored value of an object is accessed using the type identical to the run-time type of the object [72]. HexVASAN requires that the given argument types are identical to the ones used in `va_arg` [79]. These tools will generate false alarms when, for example, the program legitimately uses a character type pointer to alias different types of objects.

B. False Negatives

False negatives (i.e., failing to report bugs that are in scope) happen due to discrepancies between the bug finding policy and the mechanism that implements the policy. We identified several bug finding mechanisms that do not fully cover all of the bugs that are supposed to be covered by the policy.

Spatial safety violation detection mechanisms based on red-zone insertion and guard pages only detect illegal accesses to the red-zone or guard page directly adjacent to the intended referent of that access. Memory accesses that target a valid object beyond the red-zone or guard page are not detected. These same mechanisms also fail to detect intra-object overflows because they do not insert red-zones or guard pages between subobjects in the same parent object.

Spatial safety violation detectors that use tagged pointers may round up the allocation sizes for newly allocated objects to the nearest power of two [39], or to the nearest supported allocation size [42], [43]. The bounds checks performed by these detectors enforce allocation bounds, rather than object bounds. Thus, if an illegal memory access targets the padding added to an object, it will not be reported.

Per-object bounds tracking tools do not detect intra-object overflows because they cannot (always) distinguish object pointers from subobject pointers. This happens, e.g., when a parent object has a subobject as its first member. This subobject is located at the same memory address as the parent object.

Temporal safety violation detection mechanisms based on location-based checking or guard pages cannot detect dereferencing of dangling pointers whose target has been reused for a new memory allocation. This problem can be mitigated by delaying memory reuse for a limited time, or eliminated if a pool allocation analysis can determine when deallocated is no longer accessible [35]. Pool allocation analysis is only available at compile time, when sufficient type information is available, however.

Guard page-based temporal safety violation detectors cannot invalidate local variables that have gone out of scope. These local variables are stored in stack frames. These frames cannot be replaced by guard pages because they usually share memory pages with other frames that are still in use. Consequently, guard page-based techniques cannot detect use-after-scope and use-after-return vulnerabilities.

Temporal safety violation detectors based on dangling pointer tagging only invalidate pointers that are stored in memory. Dangling pointers stored in registers are not invalidated, even if the program eventually copies them into memory.

Most uninitialized memory use detectors approximate the language standard by considering a value as "used" only in limited circumstances, such as when it is passed to a system call, or when it is used in a branch condition.

Pointer type error detectors such as TySan [73] also conservatively approximate the effective type rules of the language standard, thus failing to detect bugs involving objects of a type unknown to their system.

Some sanitizers fail to recognize pointers that are cast to integers or copied via `memcpy`. Identity-based access checkers that use disjoint per-pointer metadata, for example, regularly fail to propagate pointer bounds across these constructs. This problem also affects sanitizers that tag dangling pointers by instrumenting stores of pointer-typed variables, but miss pointers temporarily cast to integers or copied in a type-unsafe manner.

C. Incomplete Instrumentation

Sanitizers that instrument programs statically cannot fully support programs that generate code at run time (e.g., just-in-time compilers) or programs that interact with external libraries that are not or cannot be instrumented (e.g., because their source code is not available). Some sanitizers that instrument programs at the compiler IR level also do not support programs containing inline assembly code because the compiler front-end does not translate such code into compiler IR code. In all of these cases, the sanitizer might fail to insert checks, potentially leading to false negatives. For example, if a program accesses memory from within a block of dynamically generated code, a spatial safety violation sanitizer will generally not be able to verify whether the memory access is legal.

Moreover, the sanitizer might also fail to emit the necessary instructions to update metadata. This is particularly problematic for sanitizers that need to propagate metadata explicitly (e.g., disjoint per-pointer metadata or memory status bits) [46]–[49], [52], [56], [66]. For example, if a program copies a pointer with disjoint metadata to a new memory location from within an external uninstrumented library, then the sanitizer will not copy the metadata for the source pointer. Without proper metadata propagation, the sanitizer might generate false negatives (because metadata might be missing from the store) or false positives (because the metadata might be outdated).

Incomplete instrumentation is also a problem for some tools that change the pointer representation [39], [45], [46], [54], [55], as passing fat/tagged pointers to uninstrumented code can lead to crashes, while interpreting pointers received from uninstrumented code as fat/tagged pointers can lead to false positive and false negative detections.

These problems can be overcome by embedding the sanitizer at run time instead, using a dynamic binary instrumentation framework. These frameworks cannot provide accurate type information, however, and consequently do not support certain types of sanitizers (e.g., pointer casting monitors).

D. Thread Safety

Sanitizers that maintain metadata for pointers and objects can incur both false positives and false negatives in multi-threaded programs. This can happen because they might access metadata structures in a thread-unsafe way, or because the sanitizer does not guarantee that it updates the metadata in the same transaction as program's atomic updates to its associated pointers or objects. The former problem affects FreeSentry [59] and makes the sanitizer unable to support multi-threaded programs. The latter problem affects Intel Pointer Checker [49], Intel MPX [88], and MSan [66] among others. These sanitizers allow pointers or objects to go out of sync with their metadata, which may lead to false positives and/or false negatives. Some sanitizers such as Memcheck [28] sidestep this issue by serializing the execution of multi-threaded programs, thereby always atomically updating metadata along with pointers and objects associated with it.

E. Performance Overhead

The run-time performance requirements for sanitizers are not as stringent as those for exploit mitigations. While the latter typically only see real-world deployment if their run-time overhead stays below 5% [15], we observed that sanitizers incurring less than 3x overhead are widely used in practice. In some cases, such as when the source code for a program is not (fully) available, or if the program generates code on-the-fly, even larger overheads of up to 20x are acceptable. Yet, there are good reasons to try to minimize a sanitizer's overhead. One reason in particular is that the faster a sanitizer becomes, the faster a sanitization-enabled program can be fuzzed. This in turn allows the fuzzer to explore more code paths before it stops making meaningful progress (cf. Section VII).

The primary contributors to a sanitizer's run-time overhead are its checking, metadata storage and propagation, and run-time instrumentation cost. The run-time instrumentation cost is zero for most sanitizers, because they instrument programs statically (at compile time). For sanitizers that use dynamic binary instrumentation, however, the run-time instrumentation cost can be very high. Valgrind's Memcheck [28], for example, incurs 25.7x overhead on the SPEC2000 benchmarks. 4.9x of this run-time overhead can be attributed to Valgrind itself [84], the DBT framework Memcheck is based on.

The metadata storage and propagation cost primarily depends on the metadata storage scheme. In general, embedded metadata and tagged or fat pointers are the most efficient storage schemes because they cause less cache pressure than disjoint/shadow metadata storage schemes. Embedded metadata and tagged/fat pointers have the additional advantage that their metadata automatically propagates when an object or pointer is copied. Using tagged or fat pointers is problematic in programs that cannot be fully instrumented, however (cf. Section VI-B). The one exception is low-fat pointer-based bounds tracking [42], [43], where the metadata is stored implicitly in the tagged pointer so that the tagged pointer can still be dereferenced in uninstrumented libraries. In practice, we observe that disjoint/shadow metadata storage schemes are preferred over tagged and fat pointers, despite the fact that they cause more cache pressure and require explicit metadata propagation when objects or pointers are copied.

The checking cost is strongly correlated with the sanitizer's checking frequency, which, in turn, strongly depends on the type of sanitizer. Since memory error detectors generally require coverage of all memory accesses or pointer arithmetic operations performed by a program, they introduce more overhead than other tools such as type casting checkers that monitor a smaller set of operations. Some memory error detection tools provide selective instrumentation, e.g., to monitor memory writes only, to achieve better performance at the cost of reduced coverage.

F. Memory Overhead

Sanitizers that increase the allocation sizes for memory objects, or that use disjoint or shadow metadata storage schemes have sizable memory footprints. This can be problematic on 32-bit platforms, where the amount of the addressable memory space is limited. ASan [31], for example, inserts red zones into every memory object and maintains a direct-mapped shadow bitmap to store addressability information. Consequently, ASan increases the memory usage of the SPEC2006 benchmarks by 3.37x on average. Guard page-based memory safety sanitizers, such as Electric Fence [32] and PageHeap [33], insert entire memory pages at the end of dynamically allocated objects, and therefore have even bigger memory footprints. In general, however, most sanitizers increase the program's memory footprint by less than 3x on average, even if the sanitizer stores metadata for every object or pointer in the program.

IX. Deployment

We studied the current use of sanitizers. Our goals were to determine (i) what sanitizers are favored by developers and (ii) how they differ from those that are not.

A. Methodology

Popular GitHub repositories: We compiled a list of the top 100 C and top 100 C++ projects on GitHub and examined their build and test scripts, GitHub issues, and commit histories. Most of the sanitizers we reviewed need to be integrated into the tested program at compile time. A program's build configuration would therefore reveal whether it is regularly sanitized. Our examination of the test suites and testing scripts further showed which sanitizers can be enabled during testing. Contrary to the build system/configuration files, references to sanitizers that instrument programs at run time (e.g., Memcheck) would show up here.

Sanitizer web pages: We examined the web sites for sanitizer tools and looked for explicit references to projects using the sanitizer and acknowledgments of bug discoveries.

Search trends: We examined and compared search trends for different sanitizers. We used ASan as the baseline in the search trends as our study indicates that it is currently the most widely deployed sanitizer.

B. Findings

AddressSanitizer is the most widely adopted sanitizer: We found that ASan is used in 24 and 19 of the most popular C and C++ projects on GitHub respectively. We believe that this popularity can be attributed to several of ASan's strengths: (i) ASan detects the class of bugs with the highest chance of exploitation (memory safety violations), (ii) ASan is highly compatible because it does not incur additional false positives when the program is not fully instrumented (e.g., because the program loads uninstrumented shared libraries), (iii) ASan has a low false positive rate in general and false positives that do occur can be suppressed by adding annotations to the program, or by adding the location where the false positive detection occurs to a blacklist, (iv) ASan is integrated into mainstream compilers, requiring only trivial changes to the tested program's build system, and (v) ASan scales to large programs such as the Chromium and Firefox web browsers. A weakness of ASan, and of other sanitizers that combine location-based checking with red-zone insertion, is that it produces false negatives.

One interesting observation is that DBT-based memory safety sanitizers such as Memcheck and Dr. Memory have nearly identical strengths. Additionally, these sanitizers can always instrument the full program even if part of the program's source code is not available. Yet, our study shows that while Memcheck was popular before ASan was introduced into LLVM and GCC, and its real-world use now trails that of ASan. Dr. Memory, being a much more recent tool, never achieved the same level of adoption than either of the competitors.

The adoption rate for other LLVM-based sanitizers is lower: MSan and UBSan have also seen adoption, mainly due to increased attention towards vulnerabilities such as uninitialized memory use and integer overflows. However, users frequently report high false positive rates and avoiding them requires significant effort. In fact, developers have to go to great lengths to apply those sanitizers to large projects like the Chromium web browser. To avoid false positives for MSan, the entire program must be instrumented. In Chromium's case, this means that MSan must be inserted into the web browser itself, as well as in all of its dependencies. For UBSan, the developers maintain a long list of suppressions that most notably suppresses all detections in the entire V8 JavaScript engine.

C. Deployment Directions

The deployment landscape hints at the desirable properties of a sanitizer. First, all the deployed sanitizers are easy to use. Specifically, they can be enabled via a compiler flag (Clang sanitizers) or can be applied to any binary (Memcheck). Second, the false positive rate and adoption are inversely related, i.e., fewer false positives means higher adoption (ASan and Memcheck vs. MSan and UBSan). Third, performance overhead is not a primary concern (Memcheck is used), but is avoided when a faster alternative is available (Memcheck vs. ASan).

Our own experience of applying sanitizers to the SPEC benchmarks shows that research prototypes suffer even more from false positives than widely deployed sanitizers (cf. Table III). ASan successfully ran all the benchmarks, correctly reporting known bugs in SPEC. Memcheck ran all benchmarks except for `447.dealII`, which takes more than 48 hours to finish. In contrast, SoftBound+CETS fails to run many of the benchmarks raising false alarms, due to strictness (e.g., not supporting integer-pointer casts) and compatibility (e.g., failure to update bounds for pointers created in uninstrumented libraries) issues. LFP failed to run several benchmarks, because the assumed invariant that OOB pointers do not escape the creating function is too strict. DangSan provides their own patches to circumvent incorrect invalidation of pointers.

Developers who want to sanitize memory safety issues in their projects can pick up ASan or Memcheck without much effort. However, they should be aware that these tools do not detect all classes of memory safety violations. Developers who want to adopt MSan and UBSan should expect continued efforts for recompilation of all the dependencies and/or for blacklisting and annotation to weed out false positives. For the vulnerabilities not covered by these popular sanitizers (e.g., intra-object overflow and type errors caused by type punning), developers currently have no viable option. Further research in this area is required, because existing research prototypes do not scale to real-world code bases.

X. Future Research and Development Directions

A. Type Error Detection

Due to type-punning constructs such as C-style casts and C++'s `reinterpret_cast`, it is still possible for pointers to have an illegal type when the program dereferences them. Pointer use monitoring can solve this problem because it tracks the effective types of every storage location, and detects illegal dereferences of pointers that were derived through type punning. Unfortunately, there are only a few tools that monitor pointer uses [73], [74], and they suffer from false positives and/or high performance and memory overheads. This is in large part due to the complexity of the effective type and aliasing rules in the language standard. Therefore, designing tools that detect pointer misuses with better precision and performance remains an interesting area of research.

B. Improving Compatibility

Although there exist other memory vulnerability sanitizers with better precision, ASan is by far the most deployed sanitizer. We believe that the primary reason for ASan's wider deployment is its excellent compatibility with the de facto language standards, and with partially instrumented programs. We encourage future research and development efforts to put more emphasis on compatibility. One promising direction towards better compatibility with partially instrumented programs is to combine compiler instrumentation with static or dynamic binary translation [66], where binary translation is used to instrument parts of the program that are not instrumented at compile time.

C. Composing Sanitizers

Sanitizers typically detect one particular class of bugs. Embedding multiple sanitizers is unfortunately not possible at present because existing sanitizers use a variety of incompatible metadata storage schemes. Current practice is therefore to test the program multiple times, once with each sanitizer. This is less than ideal because the resource usage may not be optimal, and multiple sanitizers may generate duplicate reports by detecting different side-effects of the same underlying bug. We encourage further research into composing sanitizers for comprehensive bug detection. One promising direction is to use multi-variant execution systems, which run multiple variants of the same program in parallel on the same input. Different sanitizers can be embedded in each variant, allowing incompatible sanitizers to run in parallel [100], [101]. Developing new sanitizers with comprehensive detection by design is also a choice (e.g., EffectiveSan [74]), which can be facilitated by generic metadata storage schemes [87], [102].

D. Hardware Support

Hardware features can lower the run-time performance impact of sanitization, improve bug detection precision, and alleviate certain compatibility issues. The idea of using special hardware instructions to accelerate memory safety violation detection has already been extensively explored [61], [103], [104]. Recent Intel CPUs even include an ISA extension called memory protection extension (MPX) built for memory error detection [88]. Intel MPX improves the speed of the software implementation of the same mechanism, though there is still room for improvement [105]. In addition, hardware features could improve compatibility and precision. For example, ARMv8's virtual address tagging allows top eight bits of the virtual address be ignored by the processor [106]. This can be used to implement the tagged pointer scheme which does not incur binary compatibility issues, because dereferencing a tagged pointer in an uninstrumented library no longer leads to processor faults. This tag also propagates back to the instrumented code, potentially increasing the bug detection precision. Hardware-assisted AddressSanitizer [107], being developed as of writing, uses this feature to detect both spatial and temporal memory safety violations at lower performance and memory costs than ASan.

E. Kernel and Bare-Metal Support

Sanitizers have traditionally only been available for user-space applications. Lower-level software such as kernels, device drivers and hypervisors is therefore missing out on the security benefits of sanitization. Unfortunately, security bugs may have the most disastrous consequences in such low-level software. Efforts are ongoing to remedy this problem. Projects led by Google, for example, are bringing ASan and MSan to the Linux kernel [108], [109]. We encourage these efforts and hope to see other classes of sanitizers adopted to lower level software. One challenge for this is to reduce the sanitizer's memory footprint. While memory overheads of 3x or more are acceptable in user-space applications for 64-bit platforms, such overheads could be a problem for lower level software on 32-bit architectures. The Linux kernel, in particular, is often compiled and run on 32-bit platforms (e.g., on IoT devices).

XI. Conclusion

Sanitization of C/C++ programs has been an active area of research for decades. Researchers have particularly focused on detecting memory and type safety issues. Many sanitizers are now available and some have seen widespread adoption. They play a critical role in finding security issues in programs during pre-release testing. There is still room for improvement, however.

We presented an in-depth analysis of state-of-the-art sanitization techniques for C/C++ programs, highlighting their precision, performance, and compatibility properties. We also presented how these properties impact real-world adoption by surveying the current deployment landscape. Our analysis identified several promising research directions. For example, there is a currently lack of precise and efficient solutions for pointer type misuse. There is similarly a lack of a systematic way to compose sanitizers to look for different classes of bugs during a single run. Finally, sanitization for kernels, hypervisors, and other privileged software has yet to be fully explored despite the high risks posed by vulnerabilities at this level.

Acknowledgment

The authors thank the anonymous reviewers for their constructive feedback. We also thank Gregory J. Duck, Mathias Payer, Nathan Burow, Bart Coppens, and Manuel Rigger for their helpful feedback. This material is based upon work partially supported by the Defense Advanced Research Projects Agency under contracts FA8750-15-C-0124 and FA8750-15-C-0085, by the United States Office of Naval Research under contract N00014-17-1-2782, and by the National Science Foundation under awards CNS-1619211 and CNS-1513837. 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 Defense Advanced Research Projects Agency or its Contracting Agents, the Office of Naval Research or its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government. The authors also gratefully acknowledge a gift from Oracle Corporation.

References

[1] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[2] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security Symposium*, 2015.

[3] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[4] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[5] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.

[6] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[7] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *European Software Engineering Conference Held Jointly with ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005.

[8] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *ACM Conference on Computer and Communications Security (CCS)*, 2006.

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[10] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[11] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):16, 2017.

[12] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[13] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *IEEE Symposium on Security and Privacy (S&P)*, 2008.

[14] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *USENIX Security Symposium*, 2014.

[15] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[16] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: The past, the present, and the future. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2012.

[17] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7, 1996.

[18] Solar Designer. Getting around non-executable stack (and fix). Email to the Bugtraq mailing list, August 1997.

[19] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine*, 11, 2001.

[20] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, 2005.

[21] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *European Workshop on System Security (EuroSec)*, 2009.

[22] ISO/IEC JTC1/SC22/WG14. ISO/IEC 9899:2011, Programming Languages — C, 2011.

[23] ISO/IEC JTC1/SC22/WG14. ISO/IEC 14882:2014, Programming Language C++, 2014.

[24] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. Taming undefined behavior in LLVM. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[25] National Vulnerability Database. NVD - CVE-2009-1897. https://nvd.nist.gov/vuln/detail/CVE-2009-1897, 2009.

[26] Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding integer overflow in C/C++. In *International Conference on Software Engineering (ICSE)*, 2012.

[27] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *USENIX Winter Conference*, 1992.

[28] Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference (ATC)*, 2005.

[29] Derek Bruening and Qin Zhao. Practical memory checking with Dr. Memory. In *International Symposium on Code Generation and Optimization (CGO)*, 2011.

[30] Niranjan Hasabnis, Ashish Misra, and R. Sekar. Light-weight bounds checking. In *International Symposium on Code Generation and Optimization (CGO)*, 2012.

[31] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference (ATC)*, 2012.

[32] Bruce Perens. Electric fence malloc debugger, 1993.

[33] Microsoft Corporation. How to use Pageheap.exe in Windows XP, Windows 2000, and Windows Server 2003, 2000.

[34] Richard WM Jones and Paul HJ Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *International Workshop on Automatic Debugging*, pages 13–26, 1997.

[35] Dinakar Dhurjati and Vikram Adve. Efficiently detecting all dangling pointer uses in production servers. In *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2006.

[36] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In *USENIX Security Symposium*, 2017.

[37] Olatunji Ruwase and Monica S Lam. A practical dynamic buffer overflow detector. In *Symposium on Network and Distributed System Security (NDSS)*, 2004.

[38] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *International Conference on Software Engineering (ICSE)*, 2006.

[39] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, 2009.

[40] Frank Ch Eigler. Mudflap: Pointer use checking for C/C++. In *Annual GCC Developers' Summit*, 2003.

[41] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R Sekar, Frank Piessens, and Wouter Joosen. PAriCheck: An efficient pointer arithmetic checker for C programs. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2010.

[42] Gregory J. Duck and Roland H. C. Yap. Heap bounds protection with low fat pointers. In *International Conference on Compiler Construction (CC)*, pages 132–142, 2016.

[43] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. Stack bounds protection with low fat pointers. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[44] Samuel C Kendall. Bcc: Runtime checking for C programs. In *USENIX Summer Conference*, 1983.

[45] Joseph L Steffen. Adding run-time checking to the portable C compiler. *Software: Practice and Experience*, 22(4):305–316, 1992.

[46] Todd M Austin, Scott E Breach, and Gurindar S Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1994.

[47] Harish Patil and Charles Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software — Practice and Experience*, 27(1):87–110, 1997.

[48] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[49] Intel. Pointer checker. https://software.intel.com/en-us/node/522702, 2015.

[50] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2002.

[51] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference (ATC)*, 2002.

[52] Wei Xu, Daniel 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.

[53] Nicholas Nethercote and Jeremy Fitzhardinge. Bounds-checking entire programs without recompiling. In *Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE)*, 2004.

[54] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. SGXBounds: Memory safety for shielded execution. In *European Conference on Computer Systems (EuroSys)*, 2017.

[55] Nathan Burow, Derrick McKee, Scott A Carr, and Mathias Payer. CUP: Comprehensive user-space protection for C/C++. In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2018.

[56] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler enforced temporal safety for C. In *International Symposium on Memory Management (ISMM)*, 2010.

[57] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 133–143, 2012.

[58] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[59] Yves Younan. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[60] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. DangSan: Scalable use-after-free detection. In *European Conference on Computer Systems (EuroSys)*, 2017.

[61] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. HardBound: Architectural support for spatial safety of the C programming language. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[62] Chris Lattner and Vikram Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[63] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Symposium on Network and Distributed System Security (NDSS)*, 2005.

[64] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[65] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX Security Symposium*, 2004.

[66] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: Fast detector of uninitialized memory use in C++. In *International Symposium on Code Generation and Optimization (CGO)*, 2015.

[67] LLVM Developers. Undefined behavior sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2017.

[68] LLVM Developers. Control flow integrity. https://clang.llvm.org/docs/ControlFlowIntegrity.html, 2017.

[69] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. Type casting verification: Stopping an emerging attack vector. In *USENIX Security Symposium*, 2015.

[70] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. TypeSan: Practical type confusion detection. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[71] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. HexType: Efficient detection of type confusion errors for C++. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[72] Alexey Loginov, Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Debugging via run-time type checking. In *International Conference on Fundamental Approaches to Software Engineering*, pages 217–232, 2001.

[73] Hal Finkel. The Type Sanitizer: Free yourself from -fno-strict-aliasing. In *LLVM Developers' Meeting*, 2017.

[74] Gregory J. Duck and Roland H. C. Yap. EffectiveSan: Type and memory error detection using dynamically typed C/C++. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[75] Parasoft. Insure++. https://www.parasoft.com/product/insure, 2017.

[76] Timothy Tsai and Navjot Singh. Libsafe 2.0: Detection of format string vulnerability exploits. *White paper, Avaya Labs*, 2001.

[77] Michael F Ringenburg and Dan Grossman. Preventing format-string attacks via automatic and efficient dynamic checking. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[78] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Michael Frantzen, and Jamie Lokier. FormatGuard: Automatic protection from printf format string vulnerabilities. In *USENIX Security Symposium*, 2001.

[79] Priyam Biswas, Alessandro Di Federico, Scott A Carr, Prabhu Rajasekaran, Stijn Volckaert, Yeoul Na, Michael Franz, and Mathias Payer. Venerable variadic vulnerabilities vanquished. In *USENIX Security Symposium*, 2017.

[80] Manuel Rigger, Stefan Marr, Stephen Kell, David Leopoldseder, and Hanspeter Mössenböck. An analysis of x86-64 inline assembly in C programs. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2018.

[81] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*, 2004.

[82] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization (CGO)*, 2003.

[83] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[84] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[85] Timothy W Curry et al. Profiling and tracing dynamic library usage via interposition. In *USENIX Summer Conference*, pages 267–278, 1994.

[86] Linux Programmer's Manual. ld.so(8), 2017.

[87] Istvan Haller, Erik van der Kouwe, Cristiano Giuffrida, and Herbert Bos. METAlloc: Efficient and comprehensive metadata management for software security hardening. In *European Workshop on System Security (EuroSec)*, 2016.

[88] Intel. Introduction to Intel memory protection extensions. https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions, 2013.

[89] Michał Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl, 2017.

[90] LLVM Developers. libFuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2017.

[91] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[92] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[93] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware evolutionary fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[94] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[95] Julian Lettner, Dokyung Song, Taemin Park, Stijn Volckaert, Per Larsen, and Michael Franz. PartiSan: Fast and flexible sanitization via run-time partitioning. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2018.
[96] Intel. Intel Inspector. https://software.intel.com/en-us/intel-inspector-xe, 2017.
[97] Micro Focus. DevPartner. https://www.microfocus.com/products/devpartner, 2017.
[98] UNICOM Global. PurifyPlus. https://teamblue.unicomsi.com/products/purifyplus, 2017.
[99] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. Into the depths of C: Elaborating the de facto standards. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.
[100] Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee. Bunshin: Compositing security mechanisms through diversification. In *USENIX Annual Technical Conference (ATC)*, 2017.
[101] Lus Pina, Anastasios Andronidis, and Cristian Cadar. FreeDA: Deploying incompatible stock dynamic analyses in production via multi-version execution. In *ACM International Conference on Computing Frontiers (CF)*, 2018.
[102] Taddeus Kroes, Koen Koning, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. Fast and generic metadata management with mid-fat pointers. In *European Workshop on System Security (EuroSec)*, 2017.
[103] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *International Symposium on Computer Architecture (ISCA)*, 2012.
[104] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. WatchdogLite: Hardware-accelerated compiler-based pointer checking. In *International Symposium on Code Generation and Optimization (CGO)*, 2014.
[105] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX explained: A cross-layer analysis of the Intel MPX system stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(2):28, 2018.
[106] ARM. ARM Cortex-A series programmer's guide for ARMv8-A. http://infocenter.arm.com/help/topic/com.arm.doc.den0024a/DEN0024A_v8_architecture_PG.pdf, 2015.
[107] LLVM Developers. Hardware-assisted AddressSanitizer design documentation. https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html, 2018.
[108] Google. Kernel address sanitizer. https://github.com/google/kasan, 2017.
[109] Google. Kernel memory sanitizer. https://github.com/google/kmsan, 2017.

Appendix A

We measured the run-time performance overhead of 10 tools by running all 19 SPEC CPU2006 C/C++ benchmarks (or all 7 C++ benchmarks for type casting sanitizers) on a single experimental platform. To assist future sanitizer developers in measuring the relative overhead of their tool, we open-sourced our fully automated build and benchmarking scripts[1].

A. Scope

We included sanitizers that are either actively maintained and/or were published between 2008 and 2017. For sanitizers that are not publicly available, we sent the authors a request for source code access. However, the authors either did not respond [41], [57], [58], or refused our request because of licensing restrictions [39], [49] or code quality concerns [36], [59]. We excluded several sanitizers that either failed to compile or run more than half of the benchmarks [48], [56], [73], or do not support our experimental platform [33]. CaVer caused several instrumented binaries to run significantly faster than the baseline binaries [69]. Since these speedups were not reported in the original paper, and since we did not have time to properly investigate the cause of these speedups, we decided to exclude CaVer from our evaluation. In the end, we evaluated 10 tools.

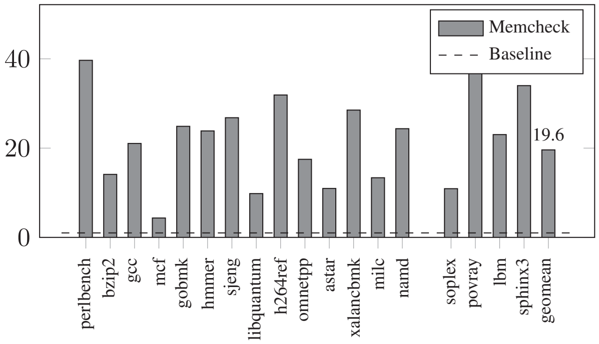
B. Experimental Setup

We conducted all experiments on a host equipped with an Intel Xeon E5-2660 CPU with 20MB cache and 64GB RAM running 64-bit Ubuntu 14.04.5 LTS. Unless stated otherwise, we used the system default libraries installed in the OS distribution.

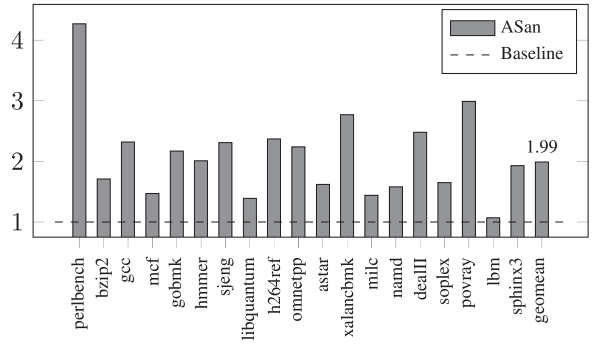
C. Results

We ran each benchmark three times with the sanitizer and report the median for each run normalized to the baseline. We used the median of three runs without the sanitizer as the baseline result. We also report and give details about false positives encountered while running benchmarks in this section. Table III summarizes the overheads and false positives.

1) Memcheck: We used the official version of LLVM/-Clang 6.0.0 to compile the baseline binaries. We measured Memcheck's overhead by running Valgrind 3.13.0 with the `--tool=memcheck` option. We excluded `447.dealII` as it does not finish within 48 hours.



2) AddressSanitizer: We used the official version of LLVM/-Clang 6.0.0 to compile the baseline binaries, and compiled the AddressSanitizer binaries using the `-fsanitize=address` flag for that same compiler. We patched several known bugs in `400.perlbench` and `464.h264ref` to avoid crashing those benchmarks early. We disabled detection of memory leaks and alloc/dealloc mismatches.

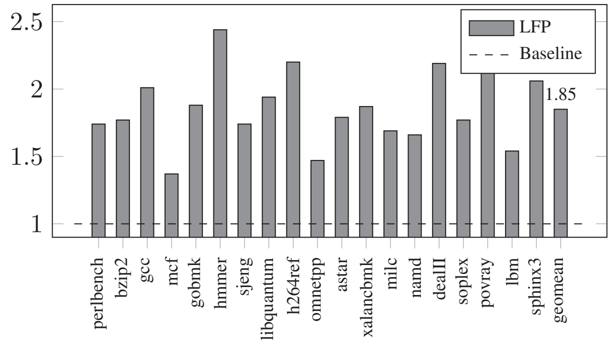


[1] https://github.com/securesystemslab/sanitizing-for-security-benchmarks

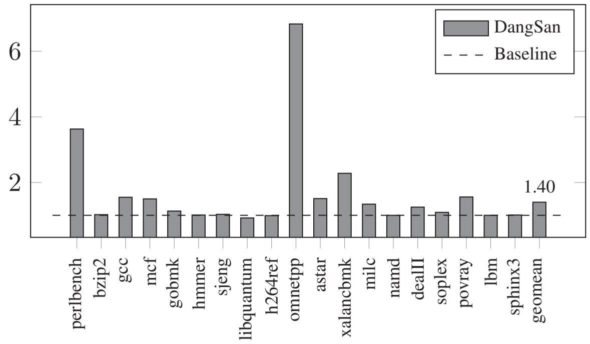
TABLE III
Normalized Performance Overheads and False Positives

	SPEC CPU2006 INT												SPEC CPU2006 FP							Geometric Mean
	400.perlbench	401.bzip2	403.gcc	429.mcf	445.gobmk	456.hmmer	458.sjeng	462.libquantum	464.h264ref	471.omnetpp	473.astar	483.xalancbmk	433.milc	444.namd	447.dealII	450.soplex	453.povray	470.lbm	482.sphinx3	
	C	C	C	C	C	C	C	C	C	C++	C++	C++	C	C++	C++	C++	C++	C	C	
Memcheck [28]	39.7	14.1	21.0	4.34	24.9	23.9	26.8	9.81	31.9	17.5	11.0	28.5	13.4	24.4		10.9	47.8	23.0	34.0	19.6
ASan [31]	4.27	1.71	2.32	1.47	2.17	2.01	2.31	1.39	2.37	2.24	1.62	2.77	1.44	1.58	2.48	1.65	2.99	1.07	1.93	1.99
Low-fat Pointer [42], [43]	1.74	1.77	2.01	1.37	1.88	2.44	1.74	1.94	2.20	1.47	1.79	1.87	1.69	1.66	2.19	1.77	2.47	1.54	2.06	1.85
DangSan [60]	3.63	1.02	1.55	1.50	1.13	1.01	1.03	0.92	0.99	6.83	1.51	2.28	1.34	1.00	1.25	1.09	1.56	1.00	1.01	1.40
MSan [66]	3.50	2.04	3.51	2.16	2.51	3.68	3.49	1.93	3.40	2.24	1.95	2.21	1.99	1.96	2.62	1.95	3.29	2.20	2.85	2.53
TypeSan [70]										1.64	0.99	1.41		0.99	1.81	1.00	1.24			1.26
HexType [71]										1.13	1.01	1.08		1.00	1.18	1.02	1.01			1.06
Clang CFI [68]										1.46	1.00	1.16		1.00	1.09	0.99	1.03			1.09
HexVASAN [79]	1.03	1.01	1.01	1.00	1.01	1.00	0.99	1.03	1.00	1.04	1.00	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.01
UBSan [67]	2.20	2.33	1.98	1.90	3.04	6.68	2.94	2.87	4.31	4.26	2.22	5.37	2.38	2.08	9.16	2.67	3.01	1.24	3.02	2.97

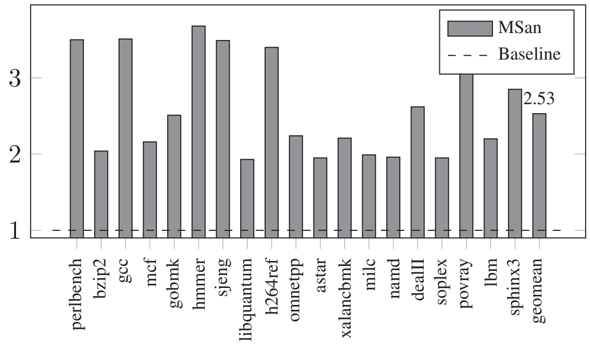
3) Low-fat Pointer: We used LLVM/Clang 4.0.0 to compile the baseline binaries, and applied LFP's changes to that compiler to generate the LFP binaries with `-fsanitize=lowfat`. The tested LFP version includes global variable support, which was not included in the original implementation [43]. We used the default size encoding in the low-fat pointer representation. As CPUs in our experimental platform do not have bit manipulation extensions, we removed optimization flags using those extensions, though they are enabled in author's build script. We applied author's blacklist to measure the overhead for benchmarks with a bug (`464.h264ref`) or false positives.



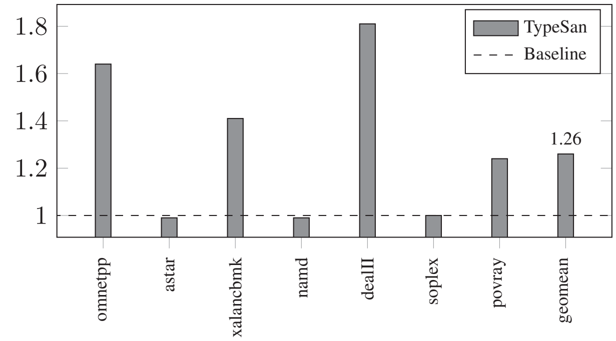
4) DangSan: We used LLVM/Clang 3.8.0 to compile the baseline binaries, and the DangSan plugin for that same compiler to generate the DangSan binaries. DangSan requires linking with the GNU gold linker and link time optimization (`-flto`) enabled, so we used gold and `-flto` to link the baseline binaries as well. Similarly, since DangSan uses `tcmalloc` as the default memory allocator, we enabled `tcmalloc` for the baseline binaries too. We did not enable `-fsanitize=safe-stack` for the baseline binaries, since it incurs overhead. To avoid false positives in `450.soplex`, we applied the pointer unmasking patch provided by the authors. We marked `400.perlbench` as having false positives based on their metadata invalidation workaround present in `tcmalloc`.



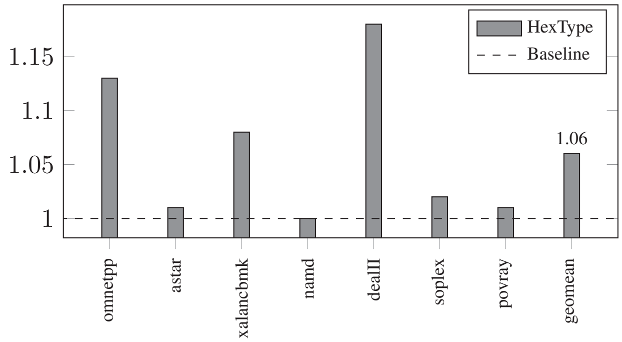
5) MemorySanitizer: We used the official version of LLVM/Clang 6.0.0 to compile the baseline binaries, and used the `-fsanitize=memory` flag for that same compiler to generate the MemorySanitizer binaries. We used instrumented versions of `libcxx` and `libcxxabi` when running C++ benchmarks. This addresses a false positive detection in `450.soplex`. We disabled early program termination after check failures to measure the run-time overhead for `403.gcc`.



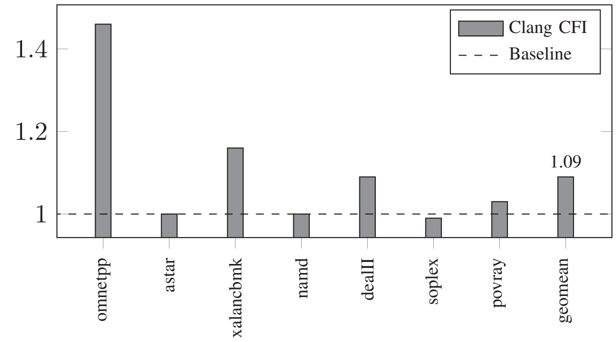
6) TypeSan: We used LLVM/Clang 3.9.0 to compile the baseline binaries, and applied TypeSan's changes to that compiler to generate the TypeSan binaries with `-fsanitize=typesan`. TypeSan uses `tcmalloc` as its default memory allocator, so we used `tcmalloc` in the baseline binaries too.



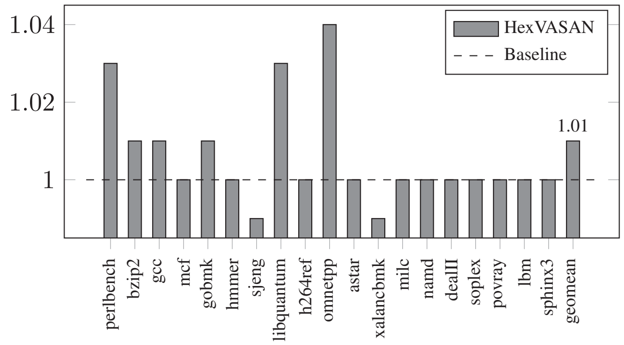
7) HexType: We used LLVM/Clang 3.9.0 to compile the baseline binaries, and applied HexType's changes to that compiler to generate the HexType binaries with `-fsanitize=hextype`. We enabled all type tracing coverage and optimization features supported by HexType. This is consistent with the experiments performed by the authors.



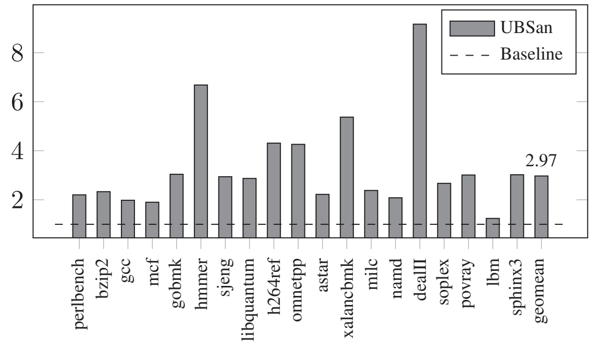
8) Clang CFI: We used the official version of LLVM/Clang 6.0.0 to compile the baseline binaries, and used the `-fsanitize=cfi` flag for that same compiler to generate the Clang CFI binaries. Clang CFI inserts checks (i) for casts between C++ class types and (ii) for indirect calls and C++ member function calls (virtual and non-virtual). We enabled `-fno-sanitize-trap` to print diagnostic information.



9) HexVASAN: We used LLVM/Clang 3.9.1 to compile the baseline binaries, and applied HexVASAN's changes to that compiler to generate the HexVASAN binaries with `-fsanitize=hexvasan`. We disabled early program termination after check failures to measure the run-time overhead for `471.omnetpp`, which has a known false positive reported in the HexVASAN paper.



10) UndefinedBehaviorSanitizer: We used the official version of LLVM/Clang 6.0.0 to compile the baseline binaries, and used the `-fsanitize=undefined` flag for that same compiler to generate the UBSan binaries. `-fsanitize=undefined` enables a carefully chosen set of sanitizers. We refer the reader to the UBSan web page for a description of the sanitizers in this set [67].


UBSan
Baseline
perlbench
bzip2
gcc
mcf
gobmk
hmmer
sjeng
libquantum
h264ref
omnetpp
astar
xalancbmk
milc
namd
dealII
soplex
povray
lbm
sphinx3
geomean