EffectiveSan: Type and Memory Error Detection using Dynamically Typed C/C++*

Gregory J. Duck
Department of Computer Science
National University of Singapore
Singapore
gregory@comp.nus.edu.sg

Roland H. C. Yap
Department of Computer Science
National University of Singapore
Singapore
ryap@comp.nus.edu.sg

Abstract

Low-level programming languages with weak/static type systems, such as C and C++, are vulnerable to errors relating to the misuse of memory at runtime, such as (sub-)object bounds overflows, (re)use-after-free, and type confusion. Such errors account for many security and other undefined behavior bugs for programs written in these languages. In this paper, we introduce the notion of *dynamically typed C/C++*, which aims to detect such errors by dynamically checking the "effective type" of each object before use at runtime. We also present an implementation of dynamically typed C/C++ in the form of the *Effective Type Sanitizer* (EffectiveSan). EffectiveSan enforces type and memory safety using a combination of low-fat pointers, type meta data and type/bounds check instrumentation. We evaluate EffectiveSan against the SPEC2006 benchmark suite and the Firefox web browser, and detect several new type and memory errors. We also show that EffectiveSan achieves high compatibility and reasonable overheads for the given error coverage. Finally, we highlight that EffectiveSan is one of only a few tools that can detect *sub*-object bounds errors, and uses a novel approach (dynamic type checking) to do so.

CCS Concepts • **Software and its engineering → Dynamic analysis**; **Data types and structures**; **Software testing and debugging**; • **Security and privacy → Systems security**; **Software and application security**;

*This research was partially supported by a grant from the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (TSUNAMi project, No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192388

Keywords Type errors, memory errors, (sub-)object bounds errors, use-after-free errors, type confusion, dynamic types, type checking, bounds checking, sanitizers, low-fat pointers, C, C++

ACM Reference Format:
Gregory J. Duck and Roland H. C. Yap. 2018. EffectiveSan: Type and Memory Error Detection using Dynamically Typed C/C++. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3192366.3192388

1 Introduction

Modern programming languages employ type systems to control object usage and detect bugs. Type systems may be *static* (compile time), *dynamic* (run time), *strong* (strict), or *weak* (loose). The type system of C/C++ is static and weak, meaning that it is up to the programmer to prevent type errors from occurring at runtime, including: bounds overflows (e.g., accessing the 101st element of an `int[100]`), (re)use-after-free (type mutation) and type confusion (e.g., bad cast) errors. Detecting such errors is desirable for a number of reasons, including: security, debugging, conformance to the compiler's *Type Based Alias Analysis* (TBAA) [5] assumptions, C/C++ standards [16, 17] compliance, code quality (e.g., readability, portability, maintainability, etc.), and revealing type-related undefined behavior. For example, type and memory errors are well known to be a major source of security bugs, e.g., accounting for over 75% of remote code execution vulnerabilities in Microsoft software alone [27]. Type errors can also be problematic for reasons other than security. For example, errors that violate the compiler's TBAA assumptions may lead to program mis-compilation—a known problem for some SPEC2006 benchmarks [13].

One solution is to deploy a *sanitizer* that instruments the program with additional code aiming to detect errors at runtime. Sanitizers are typically used for testing and debugging during the development process—helping to uncover problems before the software is deployed—and sometimes also for hardening production code (with a performance penalty). However, existing sanitizers tend to be specialized for specific classes of errors rather than enforcing comprehensive dynamic type safety. For example, TypeSan [11], BaggyBounds [1] and CETS [28] are specialized tools designed

to detect type confusion, bounds overflows and use-after-free errors respectively. Neither tool offers any protection against other kinds of errors. Furthermore, many sanitizers only offer incomplete protection against the class of errors they target. For example, a known limitation of AddressSanitizer [32], LowFat [6, 8] and BaggyBounds [1] is that they do not protect against sub-object bounds overflows, e.g.:

```
struct account {int number[8]; float balance;}
```

Modification of (`balance`) from an overflow in the account (`number`) will not be detected. Another example is CaVer [23], TypeSan [11], and HexType [18] which are specialized to detect bad casts between C++ `class` types only.

In this paper, we propose dynamic type checking for C/C++ as a unified method for detecting a wide range of memory misuse errors, including type confusion, (sub-)object bounds overflows and (re)use-after-free. We also propose an implementation of dynamic type checking in the form of the *Effective Type Sanitizer* (a.k.a. EffectiveSan). EffectiveSan dynamically verifies the *effective* type (see [16] §6.5.0 ¶6) of each object before use, allowing for the direct detection of the following classes of errors:

- **Type-errors**: By dynamically checking types, EffectiveSan directly detects type errors that are a common source of security vulnerabilities (e.g., type confusion [23]) and other undefined behavior. EffectiveSan's type checking is comprehensive, covering all standard C/C++ types (`int`, `float`, pointers, `structs`, `classes`, `unions`, etc.). Furthermore, coverage is not limited to explicit cast operations.
- **(Sub-)Object-bounds-overflows**: C/C++ types intrinsically encode bounds information, e.g., (`int[100]`), meaning that type and bounds checking go hand-in-hand. EffectiveSan uses dynamic types to detect bounds errors, as well as sub-object bounds overflows within the same object.

Furthermore, EffectiveSan can detect some classes of (re)use-after-free errors:

- **(Re)Use-after-free** and **Double-free**: By binding unallocated objects to a special type, EffectiveSan can also detect some use-after-free and double-free errors. Likewise, reuse-after-free (when the object is reallocated before the erroneous access) is protected if the reallocated object is bound to a different type.

In essence, EffectiveSan is a "generalist" sanitizer that finds multiple classes of errors using a single underlying methodology, namely, dynamic type checking. Such errors account for the majority of attacks [27] as well as other undefined behavior. Furthermore, unlike existing C/C++ type error sanitizers [11, 18, 20, 23], EffectiveSan checks pointer *use* (i.e., dereference) rather than explicit casts.

Our EffectiveSan implementation works by extending low-fat pointers [6, 8] to dynamically bind type meta data to allocated objects. Low-fat pointers have several advantages, including: speed, low memory overheads and compatibility with uninstrumented code. The key insight is to store meta data at the base of allocated objects, analogous to a hidden `malloc` header, which can be retrieved using standard low-fat pointer operations. This differs from most existing sanitizers that store meta data in a shadow space or some other adjunct memory. EffectiveSan's type meta data is detailed, storing the type and bounds of every possible sub-object, allowing for *interior pointers* (pointers to sub-objects inside allocated objects) to be checked at runtime. We experimentally evaluate EffectiveSan against the SPEC2006 benchmarks and the Firefox web browser [10]. EffectiveSan finds multiple type, (sub-)object bounds, and reuse-after-free errors in SPEC2006, with some errors previously unreported.

EffectiveSan offers more comprehensive error detection compared to more specialized tools. However, more comprehensive error detection necessitates more instrumented checks, so the trade-off is higher performance overheads. EffectiveSan is intended for deployment in the software development and testing life-cycle where error coverage is the priority. While EffectiveSan's design philosophy is to "check everything" by default, it is also possible to trade coverage for performance. To demonstrate this, we also evaluate two reduced-instrumentation variants of EffectiveSan, namely:

- EffectiveSan-type: for type-cast-checking-only; and
- EffectiveSan-bounds: for bounds-checking-only.

Both variants have similar coverage compared to existing state-of-the-art specialized sanitizers. In summary, the main contributions of this paper are:

- *Dynamic Type Checking*: We introduce dynamically typed C/C++ as a general methodology against a wide range of errors relating to the misuse of memory.
- *EffectiveSan*: We present a practical implementation of dynamically typed C/C++ in the form of the *Effective Type Sanitizer* (EffectiveSan). EffectiveSan offers comprehensive type error detection (for both C and C++), comprehensive (sub-)object bounds overflow error detection, as well as partial detection for some (re-)use-after-free errors, all using the same underlying methodology.
- *Sub-object Bounds Checking*: Dynamic type checking offers a novel approach to sub-object bounds checking. Most existing bounds-checking tools either check object bounds only (e.g., AddressSanitizer [32]), or require explicit tracking of sub-object bounds information, e.g., by changing the *Application Binary Interface* (ABI) (e.g., SoftBound [28]), however this can be a source of incompatibility. In contrast, EffectiveSan uses dynamic type information to derive sub-object bounds "on demand", does not change the ABI, and is thread-safe.
- *Evaluation*: We experimentally evaluate EffectiveSan against the SPEC2006 benchmark suite [13] and the Firefox web browser [10]. SPEC2006 is a heavily analyzed codebase, yet EffectiveSan is able to detect several new errors.

2 Background

Dynamically typed languages, such as JavaScript, Python, Lua, etc., check the types of objects at runtime. In contrast, statically typed languages, such as C, check types at compile time. Similarly, C++ is a statically typed language with the limited exception of *Run-Time Type Information* (RTTI) and the (`dynamic_cast`) operator for downcasting (casting from a base to a derived class). The C/C++ type system is intentionally weak, i.e., allowing for arbitrary pointer casting and pointer arithmetic, meaning that type and memory errors will not be prevented at compile time. By using dynamic typing, we can detect such errors at runtime at the cost of additional overheads. Note that dynamic typing concerns pointer or reference access only, e.g., (`f = *(float *)p`) is a type error if `p` does not point to a (`float`) object. Casts that create copies of objects, such as (`f = (float)i`), are valid conversions and not type errors.

Aside from RTTI, there is limited existing work on dynamic type checking for C/C++. A simple dynamic checking system for C that tags each data word with a basic type, e.g., *integral*, *real*, *pointer*, etc., was proposed in [25]. Unlike our approach, there is no distinction between different types of pointers (i.e., all pointers are treated as (`void *`)). Overheads are also very high at 35×-133× for SPEC95 [25]. Type confusion sanitizers also provide a limited form of dynamic typing discussed below. Bounds-checking is also a weak form of dynamic typing (where only the type's size is checked). CCured [30] extends the C type system with memory-safety guarantees. However, CCured has limited compatibility, no C++ support, and does not track types over arbitrary casts.

2.1 Sanitizers

Type and memory errors have long been recognized as a major source of bugs in programs written in low-level languages such as C/C++. As such, many different bug detection tools (sanitizers) have been proposed which we survey and compare.

Type Confusion. C++ provides a limited form of dynamic typing in the form of RTTI and (`dynamic_cast`) for downcasting. However, programmers will sometimes opt for the faster yet unsafe (`static_cast`) version of the same operation—a known source of security vulnerabilities. CaVer [23] and TypeSan [11] are specialized for detecting such type confusion errors caused by unsafe downcasts. Another approach is the *Undefined Behavior Sanitizer* (UBSan) [33] that transforms `static_casts` into `dynamic_casts` to enable standard RTTI protections. HexType [18] is a more general tool that extends protection to other kinds of C++ casts such as (`reinterpret_cast`), (`const_cast`), etc. Finally, libcrunch [20] can detect bad pointer casts for C programs.

Existing type confusion sanitizers have several limitations. Firstly, existing sanitizers only verify *incomplete* types that lack bounds information (e.g., T`[]` is *incomplete* whereas T`[100]` is *complete*). For example, if (`p`) points to an object of type (T`[100]`), then (`p+101`) may point to an object of any type or unused memory. Existing sanitizers will not detect such bounds errors since they assume both (`p`) and (`p+101`) have the same type (T`[]`). EffectiveSan detects (sub-)object bounds errors based on complete type information. The second limitation is that existing sanitizers instrument explicit cast operations only. *Implicit casts* (e.g., via memory, unions, function arguments, etc.) are unprotected. For example, the following is an implicit cast from (`ptrA`) to (`ptrB`):

```
memcpy(buf, &ptrA, 8); memcpy(&ptrB, buf, 8);
```

EffectiveSan instruments *pointer use* (i.e., dereference) meaning that type errors arising from (`ptrB`)'s usage will be detected (regardless of how the cast occurred). The final limitation is that existing sanitizers focus only on a subset of explicit C/C++ casts, e.g., C++ `class` casts for CaVer/TypeSan/HexType. In contrast, EffectiveSan can detect type errors for any C/C++ type (`int`, `float`, `structs`, pointers, etc.). EffectiveSan generally performs more type checks than existing tools, mainly because of increased type coverage and pointer use instrumentation.

(Sub-)Object Bounds Overflows. Object bounds overflows are well known to be a major source of security vulnerabilities. As such, many existing solutions have been proposed, including [1, 2, 6, 8, 9, 14, 19, 22, 28, 30, 32, 34] amongst others. Many solutions, such as BaggyBounds [1], LowFat [6, 8], Intel MPX [14] and SoftBound [28], work by binding *bounds meta data* (object size and base) to each pointer. The binding is typically implemented using some form of shadow memory (e.g., SoftBound, MPX) or encoding the meta data within the pointer itself (e.g., LowFat with low-fat pointers). Solutions that use shadow memory may also have compatibility issues interfacing with uninstrumented code that allocates its own memory (the corresponding entries in the shadow memory will not be initialized). This can be partially mitigated by intercepting standard memory allocation functions, or by hardware-based solutions (e.g., with MPX). Low-fat pointers avoid the problem by encoding bounds meta data within the pointer itself. Another approach to memory safety is AddressSanitizer [32] which uses *poisoned red-zones* and shadow memory to track the state of each word of memory, e.g. *unallocated*, *allocated* or *red-zone*. Out-of-bounds memory access that maps to a red-zone will be detected, however, memory errors that "skip" red-zones may be missed.

Most existing bounds overflow sanitizers protect *allocation* or *object bounds only*. This means the overflows contained within an allocated object will not be detected, e.g., the overflow into (`balance`) from Section 1. A few bounds checking systems, e.g., SoftBound [28] and Intel MPX [14], can also detect sub-object bounds overflows by using static type information for *bounds narrowing*, i.e., an operation that further constrains bounds meta information to a specific sub-object for more accurate protection. This also requires sub-object

bounds to be associated with pointers when they are passed between contexts, e.g., a pointer parameter in a function call. For example, MPX solves this problem by passing bounds information through special registers (`bnd0`-`bnd3`), or failing that, by resorting to the *bounds directory* stored in shadow memory. Similarly, SoftBound also explicitly tracks bounds information, e.g., by inserting additional function parameters [28]. Both MPX and SoftBound use shadow memory schemes that have been shown to be unsuitable for multi-threaded environments [31]. In contrast, EffectiveSan detects (sub-)object bounds errors using dynamic type information. For example, a pointer (`p`) of static type (`int *`) can be matched against an object of dynamic type (`account`), since (`p`) points to the sub-object (`number`) of compatible type. EffectiveSan will enforce the sub-bounds for (`number`), thereby preventing overflows into (`balance`) or outside the (`account`). Unlike other sub-object bounds checkers, EffectiveSan does not change the *Application Binary Interface* nor rely on thread-unsafe shared state.

(Re)Use-After-Free. Use-after-free (UAF) sanitizers include tools such as AddressSanitizer [32] and Compiler Enforced Temporal Safety (CETS) [29]. AddressSanitizer stores the allocation state in shadow memory, allowing for the detection of use-after-free errors. AddressSanitizer also mitigates reuse-after-free by putting freed objects into a "quarantine" that delays reallocation (a technique also applicable to EffectiveSan). CETS uses a more sophisticated identifier-based approach, that binds a unique tag to each allocated object, allowing for general (re)use-after-free detection.

EffectiveSan's use-after-free protection is related to the AddressSanitizer approach—but with type meta data replacing AddressSanitizer's shadow memory scheme. EffectiveSan can also detect reuse-after-free provided the object is reallocated with a different type. Although EffectiveSan's protection is not as comprehensive as specialized tools such as CETS, it is nevertheless worthwhile to target such errors anyway, since this incurs no additional costs. Tools based on instrumentation (including EffectiveSan, CETS, AddressSanitizer) may also miss some use-after-free errors because of the inherent race between the check and a call to (`free`), e.g., by another thread. (Re)use-after-free can also be mitigated using other means, such as garbage collection [3].

2.2 Our Approach

Figure 1 summarizes existing sanitizers and their capabilities. Most sanitizers are specialized to one particular class of error and/or offer partial protection against the classes of errors they do support. This means that, if more comprehensive error detection is desired, multiple different tools must be deployed at once. However, this is problematic, since most sanitizers are compiler specific (e.g., `clang` versus `gcc`) and use competing instrumentation/shadow-memory schemes that are not generally designed to be interoperable. Even if it were possible to seamlessly combine sanitizers, EffectiveSan still offers a more comprehensive level of error detection, such as type errors caused by implicit casts.

Sanitizer	Types	Bounds	UAF
CaVer [23]	Partial*	✗	✗
TypeSan [11]	Partial*	✗	✗
UBSan [33]	Partial*	✗	✗
HexType [18]	Partial*	✗	✗
libcrunch [20]	Partial^	✗	✗
BaggyBounds [1]	✗	Partial†	✗
LowFat [6, 8]	✗	Partial†	✗
Intel MPX [14]	✗	✓	✗
SoftBound [28]	✗	✓	✗
CETS [29]	✗	✗	✓
AddressSanitizer [32]	✗	Partial†	Partial‡
SoftBound+CETS [28, 29]	✗	✓	✓
EffectiveSan	✓	✓	Partial§

Figure 1. Summary of different sanitizers and capabilities against type and memory errors. Here (✓) means comprehensive protection, (✗) means no or incidental protection, and (Partial) means partial protection with caveats. The caveats are: (∗) only protects a subset of explicit C++ casts, (^) only protects explicit C casts, (†) only protects allocation bounds, (‡) only protects use-after-free (not reuse-after-free), and (§) only protects reuse-after-free for different types.

EffectiveSan's underlying approach is to convert C/C++ into a dynamically typed programming language. The basic idea is to bind a *dynamic type* to each allocated object, which can be retrieved at runtime and compared against the static type declared by the programmer. The dynamic type information is complete and supports standard C/C++ types, thus allowing for the detection of type errors beyond CaVer, TypeSan, HexType and libcrunch. Furthermore, C/C++ types encode (sub-)object size information, and thus dynamic types can be used to enforce (sub-)object bounds. EffectiveSan's bounds enforcement is precise and offers more comprehensive error detection than BaggyBounds, LowFat and AddressSanitizer. Finally, by binding deallocated objects to a special type, dynamic typing can also detect some (re)use-after-free errors. Although use-after-free detection is partial, it incurs no additional costs while still detecting many common cases.

3 Dynamic Types for C/C++

In this section, we present a dynamic type system for C/C++. This is essentially equivalent to the standard (static) type system, but also includes extensions for handling unallocated memory, and methods for calculating sub-object types and bounds at runtime.

The dynamic type of an object is a qualifier-free[1] version of the *effective type* ([16] §6.5.0 ¶6) or *object type* ([17] §3.9.0 ¶8) as defined by the C/C++ standards. The dynamic type can

[1] Qualifiers do not affect memory layout or access ([16] §6.5.0 ¶7).

	$\mathcal{L}$	: Type $\times \mathbb{Z} \mapsto P(\text{Type} \times \mathbb{Z})$
(a)	$\mathcal{L}(T, 0)$	$\ni \langle T, 0\rangle$
(b)	$\mathcal{L}(T, \text{sizeof}(T))$	$\ni \langle T, \text{sizeof}(T)\rangle$
(c)	$\mathcal{L}(T\texttt{[}N\texttt{]}, k)$	$\supseteq \mathcal{L}(\text{T}, k \bmod \text{sizeof}(T))$
(d)	$\mathcal{L}(T\texttt{[}N\texttt{]}, k)$	$\ni \langle T\texttt{[}N\texttt{]}, k\rangle$ if $k \bmod \text{sizeof}(T) = 0$
(e)	$\mathcal{L}(\texttt{struct S}, k)$	$\supseteq \mathcal{L}(T_{memb}, k - \text{offsetof}(\text{S}, memb)))$
(f)	$\mathcal{L}(\texttt{class C}, k)$	$\supseteq \mathcal{L}(T_{memb}, k - \text{offsetof}(\text{C}, memb)))$
(g)	$\mathcal{L}(\texttt{union U}, k)$	$\supseteq \mathcal{L}(T_{memb}, k)$
(h)	$\mathcal{L}(\texttt{FREE}, k)$	$= \{\langle \texttt{FREE}, 0\rangle\}$

Figure 2. The layout function ($\mathcal{L}$). Rules (c)-(h) implicitly assume that k is within the bounds of the object, that is, $0 \le k < \text{sizeof}(T)$ for rule matching $\mathcal{L}(T, k)$. Rules (e)-(g) apply to all members (*memb*) of the corresponding structure/-class/union. Here (sizeof) and (offsetof) are the standard ANSI C operators.

be any C/C++ type, including fundamental types (e.g., `int`, `float`, etc.), pointers, function pointers, arrays, structures, classes and unions. Dynamic types are always *complete*, i.e., the type's size is known. We assume w.l.o.g. that type aliases (e.g., `typedef`) are fully expanded and C++ templates and namespaces are fully instantiated. Structures, classes and unions are considered equivalent based on tag (i.e., the name), or in the case of anonymous types, based on layout. We denote the set of all types as (Type). For brevity, we use the C++ convention of referring to types by their tag, e.g., (S) is short for (`struct S`).

During a new allocation (e.g., stack allocation or heap allocation via `malloc`, `new`, `new[]`) the dynamic type will be bound to the object. For stack allocations and C++'s `new`/`new[]` operators, the dynamic type is the same as the declared type of the object as defined by the program. For `malloc` the dynamic type is deemed equivalent to the first lvalue usage type. The latter is determined by a simple program analysis.

Example 1 (Dynamic Types)**.** Consider the type definitions and allocations:

```
struct S {int a[3]; char *s;};
struct T {float f; struct S t;};
S x[8]; q = new T; r = (T *)malloc(sizeof(T));
s = (T *)malloc(100*sizeof(T));
```

Pointer `x` will be bound to type (`S[8]`), `q` and `r` bound to type (`T[1]`), and `s` bound to type (`T[100]`). Notice that all dynamic types are *complete*, where the type's size is determined by the allocation size. □

Deriving Sub-object Types. The dynamic type represents the type of the top-level allocated object. In C/C++, it is common for pointers to point to sub-objects contained within larger objects—so called *interior* pointers. Interior pointers can point to array elements, or to a member contained within a structure, class or union. Another example is C++ classes with inheritance, where base class(es) are typically implemented as sub-objects of the derived class.

EffectiveSan explicitly tracks the dynamic type of top-level allocated objects. Sub-object types are derived dynamically from the containing allocated object's type (or containing type for short) and an *offset*, i.e., the pointer difference (in bytes) between the interior pointer and the base pointer of the containing object. For the ease of presentation, we shall assume all pointer arithmetic uses byte offsets regardless of the underlying pointer type. To derive sub-object types, we assume a runtime system that can map interior pointers to containing types and offsets (see Section 5). Next the containing type and offset is mapped to the set of possible sub-object types using a *memory layout function*, denoted ($\mathcal{L}$), that is formalized as the relation defined inductively over rules (a)-(h) from Figure 2. Essentially, given a pointer p to the base of an allocated object of dynamic type T and an offset k (in bytes), the function $\mathcal{L}(T, k)$ returns the set of type/integer pairs $\langle U, \delta\rangle$ that represent *all valid sub-objects* pointed to by pointer (p+k). Here, the type (U) represents the sub-object's type, and integer δ represents the distance from the pointer (p+k) to the sub-object's base. The integer δ is used later for sub-object bounds calculation. For example, the layout for `int` assuming sizeof(`int`)=4 is

$$\mathcal{L}(\texttt{int}, 0) = \{\langle \texttt{int}, 0\rangle\} \qquad \mathcal{L}(\texttt{int}, 4) = \{\langle \texttt{int}, 4\rangle\}$$
$$\mathcal{L}(\texttt{int}, k) = \emptyset \quad \text{(otherwise)}$$

Thus, if p points to `int`, then both (p+0) and (p+4) also point to `int` by rules Figure 2(a)-(b) respectively. Rule (b) accounts for the one-past-the-last-element required by the C standard ([16] §6.5.6 ¶7,8). The layouts for other fundamental types, pointers, functions and enumerations are defined similarly. For compound types (arrays, structures and unions) we build more complicated layouts. Rules (e)-(g) state that the layout of a struct/class/union member (*memb*) of type (T_{memb}) includes the layout of (T_{memb}) offset within the containing type (the offset is zero for unions). Similarly rule (c) for arrays. For classes with inheritance, we consider any base class to be an implicit embedded member. Finally, special rule (d) states that interior pointers to array elements can also be considered pointers to the containing array itself. This is necessary because a common idiom is to scan arrays using pointers rather than element indices.

Example 2 (Structure Layout)**.** Consider a pointer p to type (`T`) defined in Example 1. Then all (sub-)objects for p are described by the following table:

Sub-obj.	Offset	Type
p	p+0	`T`
p`->f`	p+0	`float`
p`->t`	p+4	`S`
p`->t.a`	p+4	`int[3]`

Sub-obj.	Offset	Type
p`->t.a[0]`	p+4	`int`
p`->t.a[1]`	p+8	`int`
p`->t.a[2]`	p+12	`int`
p`->t.s`	p+16	`char *`

Consider (p+4), which points to the base of sub-objects (p`->t`), (p`->t.a`) and (p`->t.a[0]`) respectively, as well as

pointing to the end of sub-object (`p->f`). Using the rules from Figure 2, we derive:

$$\mathcal{L}(\texttt{T}, 4) = \{\langle \texttt{S}, 0\rangle, \langle \texttt{int[3]}, 0\rangle, \langle \texttt{int}, 0\rangle, \langle \texttt{float}, 4\rangle\}$$

Pointers to array elements can also be treated as pointers to the array itself (rule Figure 2(c)). Thus for (p+12):

$$\mathcal{L}(\texttt{T}, 12) = \{\langle \texttt{int[3]}, 8\rangle, \langle \texttt{int}, 0\rangle, \langle \texttt{int}, 4\rangle\}$$

corresponding to the array sub-object (`p->t.a`) via rule (c), the array element (`p->t.a[2]`) and the end of the previous array element (`p->t.a[1]`), respectively. □

The layout for compound objects is a *flattened* representation, meaning that ($\mathcal{L}$) returns types for even nested objects, e.g., (`p->t.a[2]`) is three levels deep.

Finally, we remark that `unions` (`U`) are treated no differently to `structs` or `classes`, except that the member offset is defined to be zero, i.e., (offsetof(`U`, *memb*) = 0). This means that members always overlap. However, even `structs` may have overlapping sub-objects, as demonstrated by Example 2.

A Special Type for Deallocated Memory. Deallocated objects are bound to a special type (`FREE`) that is defined to be distinct from all other C/C++ types. This reduces use-after-free and double-free errors to type errors without any other special treatment. The (`FREE`) type has a special layout defined by rule Figure 2(h). Essentially, if p points to deallocated memory, then so does (p+k) for all k. Reuse-after-free is already handled for the case where the reallocated object has a different type to that of the dangling pointer.

Calculating Sub-object Bounds. C/C++ types also encode bounds information. For example, the type (`int[100]`) is an array object of length 100, and accessing an element outside the range 0..99 is an object bounds error. Hence, full dynamic type checking necessitates the enforcement of object bounds at runtime. To support this, we calculate (sub-)object bounds using dynamic type information. The basic idea is as follows: let p point to an object of type T and q=(p+k), then each pair $\langle U, \delta\rangle \in \mathcal{L}(T, k)$ corresponds to a sub-object of type U pointed to by q. The integer δ represents the distance from q to the start of the sub-object, and is not necessarily zero (e.g., interior pointers to arrays). The sub-object bounds, represented as an address range, can be calculated using the following helper function:

$$\texttt{type\_bounds}(q, \langle U, \delta\rangle) = q{-}\delta \mathinner{..} q{-}\delta{+}\texttt{sizeof}(U)$$

For example, let us consider the pointer q=(p+12) into the sub-object (`p->t.a`) corresponding to the pair $\langle$`int[3]`, 8$\rangle \in \mathcal{L}$(`T`, 12) from Example 2. The sub-object bounds for (`p->t.a`) is (p+4)..(p+16), i.e., spanning offsets 4..16 bytes.

4 Dynamic Type Check Instrumentation

The aim of *dynamic type checking* is to verify that pointer *use* (a.k.a., pointer dereference) is consistent with the dynamic type of the underlying object. The basic idea is as follows: suppose pointer p with static type (T `*`) is dereferenced, then dynamic type checking verifies the following properties:

- *Type Correctness*: pointer p must point to the i^{th} element of an object with dynamic type (T`[`N`]`) for some i, N; and
- *Bounds Correctness*: index i must be within the bounds of the object, i.e., $i \in 0..N{-}1$.

These properties ensure that the dereference is consistent with the complete dynamic type (T`[`N`]`)—including both the incomplete type (T`[]`) and bound (N)—effectively transforming C/C++ into a dynamically typed programming language.

EffectiveSan implements dynamic type checking in the form of *dynamic type check instrumentation* which ensures that all pointer use is guarded by an explicit type/bounds check. For performance reasons, EffectiveSan's instrumentation also aims to minimize the number of type checks. One key observation is that the dynamic type is invariant w.r.t. pointer arithmetic, e.g., for q=p+k, then pointers p and q reference the same underlying object, and hence the same type. Thus, only p need be type checked provided the derived pointer q remains within the bounds of the object. Similarly, we can avoid type checking field access q=`&`p`->`m. EffectiveSan's dynamic type check instrumentation schema is shown in Figure 3, and is summarized as follows:

- Figure 3(a)-(d): *Type checking input pointers.* All *input pointers* (i.e., function parameters 3(a), call returns 3(b), pointers read from memory 3(c) and pointers created by casts 3(d)[2]) are type checked against the incomplete[3] static type declared by the programmer. The check also calculates the (sub-)object bounds based on the dynamic type, representing the address range for which the static type is correct.
- Figure 3(e)-(f): *Propagating/narrowing bounds to derived pointers.* Rule 3(e) covers pointer arithmetic and 3(f) field access; and
- Figure 3(g): *Bounds checking* all pointer use/escapes.

Rule (d) also extends to other kinds of casts such as integer-to-pointer, C++'s `static_casts`, etc. The type check is implemented as a call to a special (`type_check`) function supplied by the EffectiveSan runtime system (to be defined later in Section 5). The (`type_check`) function will log an error message if the pointer does not point to an object, or sub-object of a larger object, of the complete type (`type[`N`]`) for some N. For example:

```
int *p = new int[100];
BOUNDS b1 = type_check(p, int[]);
BOUNDS b2 = type_check(p, float[]);
```

The first type check passes but the second fails since (`int`) and (`float`) are distinct types. Assuming there is no error, the (`type_check`) function will also return the bounds of

[2] For our purposes, we consider pointers created by casts to be inputs.
[3] As a simplification, we assume that all static types are incomplete. A complete static type check can be decomposed into an incomplete type check followed by a bounds narrowing operation.

(a)
```
f(type *p) {
BOUNDS b = type_check(p, type[]);
... }
```

(b)
```
type *p = f(···);
BOUNDS b = type_check(p, type[]);
```

(c)
```
type *p = *q;
BOUNDS b = type_check(p, type[]);
```

(d)
```
type *p = (type *)q;
BOUNDS b = type_check(p, type[]);
```

(e)
```
type *p = &q->field;
BOUNDS b = bounds_narrow(bq, q->field);
```

(f)
```
type *p = q + k;
BOUNDS b = bq;
```

(g)
```
bounds_check(p, b);
```
`val = *p;` *or* `*p = val;` *or* `p` escapes [6]

Figure 3. The dynamic type check instrumentation schema. Here `b` represents the bounds for `p`, and `bq` for `q`.

the matching (sub-)object. Bounds are represented by a pair of pointers, e.g., `b1`=$\{p..p + 100 * \text{sizeof}(\texttt{int})\}$.

The next step is to ensure that all (derived) pointer use is within the calculated bounds. For this, rule Figure 3(g) inserts a bounds check, as represented by a call to a special (`bounds_check`) function, before each pointer use. A call `bounds_check(p,b)` will report an error if:

$$\{\texttt{p..p} + \text{sizeof}(\texttt{*p})\} \cap b \neq \{\texttt{p..p} + \text{sizeof}(\texttt{*p})\}$$

Rule Figure 3(e) represents bounds *narrowing* to any sub-object selected by field access. The (`bounds_narrow`) operation between bounds (`b`) and field (`p->field`) is defined as interval intersection:

$$b \cap \{(\texttt{\&p->field})..(\texttt{\&p->field} + \text{sizeof}(\texttt{p->field}))\}$$

Narrowing is similar to that of MPX [14]. Note that ordinary pointer arithmetic (e.g., array access) is not narrowed, see rule 3(f), since the resulting pointer may still refer to the containing array. Finally, we note that EffectiveSan will limit instrumentation to *used* pointers only (either directly or indirectly via a derived pointer). For example, a function that merely casts and returns a pointer will not attract instrumentation, unlike CaVer, TypeSan, HexType and libcrunch. For EffectiveSan, it is the responsibility of the eventual user of the pointer to check the type.

EffectiveSan's instrumentation schema does not change the *Application Binary Interface* (ABI) nor does it rely on disjoint mutable meta data to pass information between functions. Instead, type/bounds information is always (re)calculated "on demand". This helps maximize compatibility/thread-safety, which is essential when instrumenting large code bases such as Firefox [10]. However, this also assumes that input pointers are within the bounds of the underlying object. To help enforce this, rule 3(g) also checks the bounds of *pointer escapes* (e.g., passing a pointer as a parameter, writing a pointer to memory, etc.). This is the same rationale used by *low-fat pointers*, see [6] for more information.

Example 3 (Dynamic Type Check Instrumentation)**.** Consider two functions: (`length`) calculates the length of a linked-list and (`sum`) calculates the sum of an array. The instrumented versions of these functions (using the Figure 3 schema) is shown in Figure 4. The instrumentation in lines $\{2, 7, 8, 10, 16, 20\}$ is highlighted, and the original functions can be obtained by deleting these lines and eliminating temporary variables. For the (`length`) function, the input pointer(s) (`xs`) on lines $\{2, 10\}$ are type checked against the static type (`node[]`) declared by the programmer. This means that (`xs`) must point to an object (or sub-object of a larger object) compatible with type (`node`). The (`type_check`) function does not guarantee (`xs`) points to the base of a complete (`node`) object (e.g., rule Figure 2(b) allows (`xs`) to point to the end of an object), so derived pointer (`tmp`) may be an overflow. To prevent this, the derived pointer (`tmp`) is bounds checked on line 8. Similarly, for the (`sum`) function, the input pointer (`a`) is type checked against the static type (`int[]`), and the derived pointer (`tmp`) is bounds checked before access. Note how all pointer use (lines $\{9, 21\}$) is preceded by a type/bounds check. Figure 4 also illustrates how the number of type checks depends on the program itself. For example, (`length`) requires $O(N)$ type checks (one for each node in the list) whereas (`sum`) only requires a single type check on function entry. □

The (`sum`) function also highlights how the instrumentation schema minimizes the number of type checks. Here the input pointer (`a`) is type checked exactly once outside of the loop, whereas the subsequent derived pointers (`a+i`) are merely bounds checked.

Finally, we remark that the Figure 3 schema is not designed to be complete with respect to use-after-free errors. For completeness, the combined type/bounds check and memory operation must be atomic, else a call to (`free`), e.g., by another thread, may mutate the type. In practice, this means that some use-after-free errors may not be detected. That said, complete use-after-free detection is not a design goal of EffectiveSan, and even partial detection can be useful. For example, EffectiveSan detects known SPEC2006 use-after-free bugs (see Section 6).

5 Dynamic Type Check Runtime

EffectiveSan's runtime system is based on *low-fat pointers*.

Low-fat Pointers. Low-fat pointers [6, 8] are a method for encoding bounds meta data (i.e., size and base of an allocation) within the native machine pointer representation itself. Low-fat pointers require sufficient pointer bit-width, and are

```
1  int length(node *xs) {
2     BOUNDS b = type_check(xs, node[]);
3     int len = 0;
4     while (xs != NULL) {
5        len++;
6        node **tmp = &xs->next;
7        b = bounds_narrow(b, xs->next);
8        bounds_check(tmp, b);
9        xs = *tmp;
10       b = type_check(xs, node[]);
11    }
12    return len;
13 }
14
15 int sum(int *a, int len) {
16    BOUNDS b = type_check(a, int[]);
17    int sum = 0;
18    for (int i = 0; i < len; i++) {
19       int *tmp = a + i;
20       bounds_check(tmp, b);
21       sum += *tmp;
22    }
23    return sum;
24 }
```

Figure 4. Instrumented `length` and `sum` functions.

feasible for 64-bit systems (e.g., the `x86_64`). To use low-fat pointers, objects must be allocated using a special *low-fat memory allocator* that ensures the returned pointers are suitably encoded. The low-fat heap allocator [6] provides replacement functions (`lowfat_malloc`, `lowfat_free`, etc.) to all the `stdlib` memory allocation functions. The replacement functions have the same interface (i.e., function prototype) as the originals. We also implement low-fat pointers for both stack [8] and global [7] objects.

Several low-fat pointer encodings have been proposed. For this paper, we use the low-fat pointer encoding from [6, 8]. This encoding provides the following abstract operations: given a low-fat pointer p to (possibly the interior of) a low-fat allocated object O, then:

$$size(p) = \text{sizeof}(O) \qquad base(p) = \&O$$

That is, given a low-fat pointer p, we can use the *size* and *base* operations to quickly determine the bounds meta data of the allocated object. E.g., if

```
str = lowfat_malloc(sizeof(char[32]))
```

then *size*(`str+10`)=32 and *base*(`str+10`)=`str`, etc. Not all pointers will be low-fat pointers, and such pointers are referred to as *legacy*. For legacy pointer q, $size(q)$ = `SIZE_MAX` and $base(q)$ = `NULL`. Support for legacy pointers is essential to handle non-instrumented code, e.g., libraries, and also some pointers from *Custom Memory Allocators* (CMAs).

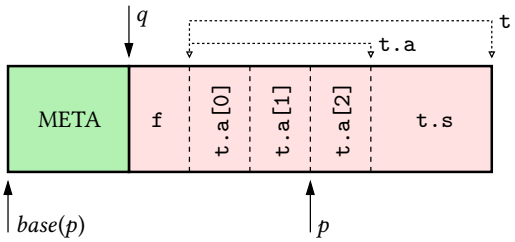

Figure 5. Object and object meta data layout.

The low-fat pointer encoding of [6, 8] works by (1) arranging objects into different memory regions based on allocation size, and (2) ensuring that all objects are allocation size-aligned. Thus, given a pointer p, we can quickly derive the allocation size (i.e., $size(p)$) based on which memory region p points into. Next the $base(p)$ operation is implemented by rounding p down to the nearest $size(p)$-aligned address. Both the $size(p)$ and $base(p)$ operations are fast and constant time $O(1)$. For more on low-fat pointers, see [6, 8].

Using Low-fat Pointers for Meta Data. Low-fat pointers were originally designed for *allocation bounds checking* using the meta data encoded in the pointer. That is, given pointer p, any access outside the range $base(p)..base(p)+size(p)$ is a bounds error that will abort the program. For EffectiveSan, we repurpose low-fat pointers as a general method for binding meta data (in our case, type information) to objects. The basic idea is to store the meta data at the base of the object, and this meta data can be retrieved from any interior pointer by using the $base(p)$ operation. We refer to this as *object meta data*, since it is associated with every allocated object. In the case of EffectiveSan, the object meta data contains a representation of the dynamic type of the allocated object.

Example 4 (Object Meta Data)**.** An example of the combined object and meta data layout is shown in Figure 5. Here we assume the object is of type (`T`) from Example 1, and the layout of each sub-object (e.g., `f`, `t`, `t.a[0]`, etc.) from Example 2 is also illustrated. The memory is divided into two main parts: space for the object meta data (META) and space for the allocated object itself. Given a pointer p to the object or sub-object (e.g., p=`&t.a[2]` in Figure 5), the pointer to the object meta data can be retrieved by $base(p)$. □

It is important to note that the meta data (META) is bound to the outermost object only, not each sub-object, and occupies memory immediately before the start of the object (q). Thus, (META) is analogous to a "malloc header" that is invisible to the program, and the layout of C/C++ objects is otherwise unchanged. Under our scheme, the (META) header is a type-integer pair storing (1) the (top-level) allocation type of the object, and (2) the object's allocation size (e.g., the parameter to (`malloc`)). For sub-objects, the type can be retrieved using the layout function ($\mathcal{L}$) discussed below.

```
1  void *type_malloc(size_t size,TYPE t){
2    META *meta = lowfat_malloc(
3      sizeof(META)+size);
4    meta→type = t;
5    meta→size = size;
6    return (void *)(meta + 1);
7  }
8
9  BOUNDS type_check(void *ptr, TYPE s) {
10   META *meta = base(ptr);
11   if (meta == NULL) /*legacy pointer*/
12     return (0..UINTPTR_MAX);
13   TYPE t = meta→type;
14   void *bptr = (void *)(meta + 1);
15   BOUNDS b = (bptr..bptr+meta→size);
16   ssize_t k = ptr-bptr;
17   for (auto o : ℒ(t,k))
18     if (o.type == s) {
19       BOUNDS c = type_bounds(ptr, o);
20       return bounds_narrow(b, c);
21     }
22   type_error();      /*report error*/
23   return (0..UINTPTR_MAX);
24 }
```

Figure 6. Simplified definitions for the `type_malloc` and `type_check` functions.

To implement the object layout of Figure 5, we replace standard (`malloc`) with the version shown in Figure 6 lines 1-7. Here, (`type_malloc`) is a thin wrapper around the underlying low-fat memory allocator (`lowfat_malloc`). The wrapper function takes a type (`t`) as an argument (similar to C++'s `new` operator). Here we treat types as first class objects of type (`TYPE`). In lines 2-3, the wrapper allocates space for both the object (of sizeof(`t`)) and the object meta data (of sizeof(`META`)) using the underlying low-fat allocator. Lines 4-5 store the allocated object's meta data at the base address. Line 6 returns the pointer to the start of the allocated object excluding the meta data. The (`type_malloc`) function essentially binds the allocation type (`t`) to the memory returned for the allocated object. We similarly wrap low-fat stack [8] and global [7] objects with meta data.

Memory deallocation is handled by a (`type_free`) replacement for stdlib (`free`). The replacement function overwrites the object meta data with the special type (`FREE`) defined in Section 3 before returning the memory to the underlying low-fat allocator. The low-fat allocator has also been modified to ensure that the meta data will be preserved until the memory is reallocated. The (`type_free`) function also detects double free errors.

Type Checking with Meta Data. By replacing the standard allocators, all objects are bound to the allocation type which can be retrieved using the *base* operation. Combined with the layout function ($\mathcal{L}$), the (`type_check`) function can be implemented as shown in Figure 6 lines 9-24. Here the (`type_check`) function has three basic steps:

1. Get the allocation type (`t`), bounds (`b`) and object base pointer (`bptr`) (lines 10-15);
2. Calculate the sub-object offset (`k`) (line 16);
3. Scan all sub-objects at offset (`k`) as returned by the layout function ($\mathcal{L}$) (lines 17-21). Return the bounds of the sub-object that matches the declared static type (`s`) narrowed to the allocation bounds, else raise a type error (line 22) if no match exists.

For legacy pointers, the (`type_check`) function always succeeds and returns "wide bounds" (lines 11-12) for compatibility reasons. Likewise, wide bounds are returned after a type error has been logged.

Example 5 (Type Check)**.** Let p point to an allocated object of type (`T`) from Example 2. Assuming that sizeof(`META`)=16, the type will be stored as object meta data at address $base(p) = (p-\text{sizeof}(\texttt{META})) = (p-16)$. Consider the interior pointer (q=p+12). Then `type_check`(q, `int[]`) computes:

1. t = ((`META` *)$base(q)$)->`type` = `T`
2. $k = q - base(q) + \text{sizeof}(\texttt{META}) = 12$
3. $\mathcal{L}(\texttt{T}, 12) = \{\langle \texttt{int[3]}, 8\rangle, \langle \texttt{int}, 0\rangle, \langle \texttt{int}, 4\rangle\}$

Type (`int[]`) matches the first sub-object $\langle$`int[3]`, 8$\rangle$, and the bounds p+4..p+16 are returned. On the other hand, the `type_check`(q, `double[]`) will fail since there is no matching sub-object for type (`double[]`). □

As illustrated in Example 5, it is sometimes possible to have multiple matching sub-objects, i.e., $\langle$`int[3]`, 8$\rangle$, $\langle$`int`, 0$\rangle$, and $\langle$`int`, 4$\rangle$ all match type (`int[]`). In such cases, the following tie-breaking rules are used:

1. sub-objects with wider bounds are preferred; and
2. pointers-to-the-end-of-sub-objects (see Figure 2(b)) are matched last.

Thus, the sub-object bounds for (`int[3]`) is returned. Note that our approach for deriving (sub-)object bounds differs from that of other systems such as SoftBound [28] and MPX [14]. These systems track (sub-)object bounds by passing meta data whereas EffectiveSan always (re)calculates bounds using the dynamic type. Explicit tracking may allow for narrower bounds for some cases of type ambiguity, e.g., when the bounds for (`int`) is intended. In order to pass narrowed pointer arguments, SoftBound necessitates changing the *Application Binary Interface* (ABI). EffectiveSan's approach achieves very good binary compatibility since the underlying ABI is not changed. Furthermore, SoftBound/MPX require meta data updates when a pointer is written to memory, which creates a data race for multi-threaded applications [31]. Our approach requires no such updates, allowing for better multi-threading support.

Layout and Type Meta Data Implementation. The object meta data is a representation of the dynamic type of the

corresponding object. EffectiveSan represents incomplete types (i.e., T[]) as pointers to a *type meta data* structure containing useful information, such as the type's size (i.e., sizeof(T)), name (for reflection) and layout information. The type meta data structure is defined once per type.

To reduce overheads, EffectiveSan uses a *layout hash table* representation. The basic idea is as follows: for all possible type (T), sub-object type (S) and sub-object offset (k) combinations:

$$\langle \mathtt{S}, \delta \rangle \in \mathcal{L}(\mathtt{T}, k) \qquad 0 \leq k \leq \mathrm{sizeof}(\mathtt{T})$$

the layout hash table will contain a corresponding entry:

$$\mathtt{T} \times \mathtt{S} \times k \mapsto -\delta \,..\, \mathrm{sizeof}(\mathtt{S}) - \delta$$

The entry maps a (T×S×k) triple to the corresponding sub-object bounds relative to offset k. In order to keep the hash table finite, only entries corresponding to offsets $0 \leq k \leq$ sizeof(T) are stored. Otherwise, for entries outside this range, the offset is first normalized ($k := k$ mod sizeof(T)). If multiple matching sub-objects exist for the same (S) the above tie-breaking rules apply. Using this implementation, the sub-object matching of Figure 6 lines 17-21 can be efficiently implemented as an $O(1)$ hash table lookup.

Example 6 (Layout Hash Table)**.** The layout hash table for (T[]) from Example 2 includes the following entries:

$$(\mathtt{T}, \mathtt{T}, 0) \mapsto -\infty..\infty \quad (\mathtt{T}, \mathtt{float}, 0) \mapsto 0..4 \quad (\mathtt{T}, \mathtt{S}, 4) \mapsto 0..20$$
$$(\mathtt{T}, \mathtt{int}, 4) \mapsto 0..12 \quad (\mathtt{T}, \mathtt{int}, 8) \mapsto -4..8$$
$$(\mathtt{T}, \mathtt{int}, 12) \mapsto -8..4 \quad (\mathtt{T}, \mathtt{char}\ *, 16) \mapsto 0..8$$

Note that, since type (T[]) is incomplete, the corresponding top-level entry is unbounded.[4] Consider the type check of pointer (q=p+12) against (int[]) from Example 5. The corresponding hash table entry (T, int, 12) maps to the bounds (−8..4). Thus, the type check succeeds with the final bounds p+12−8..p+12+4 = p+4..p+16. Furthermore, the type check of q against (double[]) fails since there is no corresponding layout hash table entry for (T, double, 12). □

Our basic approach has also been extended to handle other standard C/C++ language features, including:

1. Structure types with *flexible array members*; and
2. Automatic coercions between types allowable under the C, "sloppy" [20] or "de facto" [26] standards.

Structures with *Flexible Array Members* (FAMs) have definitions of the form (struct T {⋯; U member[];}), where (member) of type (U[]) is the FAM. Other forms are also possible. The size of the FAM is determined by the object's allocation size. Structures with a FAM are treated as equivalent to (struct T {⋯; U member[1];}), and EffectiveSan uses an alternative offset normalization for k>sizeof(T):

$$k := ((k - \mathrm{sizeof}(\mathtt{T})) \bmod \mathrm{sizeof}(\mathtt{U})) + \mathrm{sizeof}(\mathtt{T})$$

[4] The final bounds returned by (type_check) is narrowed to the actual allocation size (Figure 6 line 20) stored in the object meta data.

The final feature is automatic coercion between different types, such as automatically coercing (char[]) to other types. To implement this, EffectiveSan uses two layout hash table lookups instead of one: if the first lookup (T, S, k) fails, next (T, char, k) is tried, representing the coercion from (char[]) to (S[]). This idea can be generalized to other kinds of useful coercions, such as (void *) to (S *).

Type meta data, including the layout hash table, is automatically generated using a compiler pass, once per compiled module. Each type meta data object is declared as a *weak symbol*, meaning that only one copy will be included in the final executable. The type meta data is constant (read-only) and thus cannot be modified at runtime.

6 Experiments

We have implemented a prototype version of EffectiveSan using the LLVM compiler infrastructure [24] version 4.0.0 for the x86_64 architecture. EffectiveSan's instrumentation is a two step process. In the first step, a modified clang front-end generates a *type annotated* LLVM [24] *Intermediate Representation* (IR) of the C/C++ program. Here, type annotations are associated with each LLVM IR instruction/-global/function using the standard DWARF [4] debug format (similar to that generated by the (-g) command-line option). In the second step, the type annotated IR is instrumented using the schema from (Figure 3). This step also replaces all heap/stack/global allocations with the typed variants and generates the runtime type meta data described in Section 5. Our implementation supports all types described in Section 3, including fundamental types, pointers, structures, classes, unions, etc., as well as standard C/C++ features such as inheritance, virtual inheritance, templates, multi-threading, basic coercions between (T) to/from (char[]) and (T *) to/from (void *), and objects with flexible array members. In addition to the Figure 3 schema, our EffectiveSan prototype supports basic optimizations such as: removing dynamic type checks that can never fail (e.g., C++ upcasts), removing subsumed bounds checks, and removing redundant bounds narrowing operations. For speed, all instrumentation except (type_check) is inlined.

By default, EffectiveSan logs all errors without stopping the program. EffectiveSan may also be configured to merely count errors (without detailed log messages), and/or to abort after N errors for some $N \geq 1$. For our experiments, logging mode is used to find errors, and counting mode is used for measuring performance.

Limitations. The EffectiveSan prototype may not detect all possible type and memory errors. For example, EffectiveSan can only partially protect legacy pointers in the form of bounds narrowing. For non-legacy pointers, EffectiveSan must correctly bind the allocation type with each allocated object. For global/stack objects as well as objects allocated using C++'s new, the allocation type is simply the declared

type and is unambiguous. However, for heap objects allocated by (`malloc`) we use a simple program analysis (see Example 1). Some *Custom Memory Allocators* (CMAs) use internal data structures to track memory, resulting in type errors when cast. For our experiments, we use a version of SPEC2006 with some CMAs removed, see Appendix A. EffectiveSan will also not detect errors that are optimized away by LLVM before the instrumentation pass.

For practical reasons, the current prototype implements some simplifications, including: treating `enum`s as (`int`), C++ references as pointers, and virtual function tables as arrays of generic functions. Some simplifications are inherited from the `clang` frontend. The current prototype also does not aim to implement a strict interpretation of the C/C++ standards. For example, there is no tracking of pointer *provenance* [12]. Furthermore, the prototype implements some common "sloppy" [20] and "de facto" [26] extensions, such as (`T *`) to/from (`void *`) coercions. The final limitation relates to sub-object matching. By default, EffectiveSan heuristically chooses the sub-object with the widest bounds (see the tie-breaking rules), which may differ from the intended bounds. For example, given:

```
union { float a[10]; float b[20]; };
```

A type check against (`float[]`) will always return `b`'s bounds.

6.1 Effectiveness

To test the effectiveness of EffectiveSan, we use the SPEC2006 [13] benchmarks and the Firefox web browser version 52 (ESR). The SPEC2006 benchmarks (~1.1million sLOC) comprise several integer and floating point C/C++ programs. For SPEC2006 we use the standard workloads. For Firefox (~7.9million sLOC) we use standard web benchmarks (see Figure 10).

The results for SPEC2006 are summarized in Figure 7. Here (kilo-sLOC) represents the source lines of code (in thousands), (#Type) the number of type checks, (#Bounds) the number of bounds checks, (#Issues-found) the number of issues logged by EffectiveSan. We bucket issues by type and offset to prevent the same issue from being reported at multiple different program points. Of the ~2.2 trillion type checks in Figure 7, only ~1.1% were performed on legacy pointers, meaning that EffectiveSan achieves high coverage.

For SPEC2006 our EffectiveSan prototype detects several issues (see Figure 7), including:

- A use-after-free bug in `perlbench` (reported in [32]).[5]
- A bounds overflow error in `h264ref` (reported in [32]).
- Three sub-object bounds overflow errors in `gcc`, `h264ref` and `soplex`.[6]
- Multiple type errors (discussed below).

[5] Only applicable to the SPEC2006 `test` workload.
[6] Some are also found by MPX, see [31].

SPEC2006				
Bench.	kilo-sLOC	checks (billions) #Type	#Bound	#Issues-found
`perlbench`	126.4	177.9	297.7	35
`bzip2`	5.7	70.1	644.3	1
`gcc`	235.8	105.2	204.1	41
`mcf`	1.5	34.9	98.7	0
`gobmk`	157.6	90.9	421.3	0
`hmmer`	20.7	22.0	1393.4	0
`sjeng`	10.5	27.3	478.0	0
`libquantum`	2.6	276.4	561.1	0
`h264ref`	36.1	392.5	891.5	3
`omnetpp`++	20.0	86.5	194.7	0
`astar`++	4.3	72.5	216.8	0
`xalancbmk`++	267.4	267.8	390.6	15
`milc`	9.6	29.4	347.1	1
`namd`++	3.9	16.1	362.6	1
`dealII`++	94.4	266.1	701.3	13
`soplex`++	28.3	80.8	219.8	1
`povray`++	78.7	83.2	176.0	10
`lbm`	0.9	4.0	333.3	1
`sphinx3`	13.1	89.4	903.9	2
Totals (all)	1117.5	2193.0	8836.3	124
Totals (C++)	497.0	873.1	2261.7	40

Figure 7. Summary of the SPEC2006 benchmarks. C++ benchmarks are marked with a (++), and the rest are C. We bucket issues by type and offset. Benchmarks with issues are highlighted.

As far as we are aware, all previously known bounds, type confusion, and use-after-free errors are detected. EffectiveSan also detects new type errors that have not been previously reported (see below).

Interestingly, EffectiveSan reports zero issues (on executed paths) for the `mcf`, `gobmk`, `hmmer`, `sjeng`, `libquantum`, `omnetpp`, and `astar` benchmarks. Similarly, the benchmarks `bzip2`, `h264ref`, `milc`, `namd`, `soplex`, `lbm` and `sphinx3` report one or two minor issues. This shows that it is feasible for well disciplined C/C++ code to have zero type and memory errors. Of the remaining benchmarks, `perlbench`, `gcc`, and `povray` had the most issues, as is discussed below.

Type Errors. EffectiveSan detects multiple type errors in the SPEC2006 benchmarks, including:

- Bad C++ downcasts (type confusion) in `xalancbmk`;
- Multiple instances of casting to container types, i.e.: (`T`) cast to (`struct S {T t; ... }`) for some `T`, `S`. Several instances relating to `stdlib++` are similar to those previously reported by CaVer [23].
- Multiple instances of casting between classes/structures with the same layout (e.g., phantom classes [23]).
- `gcc`/`sphinx3` casts objects to (`int[]`) to calculate hash values or checksums.
- `gcc` with incompatible definitions for the same type (e.g., different `struct` definitions using the same tag).

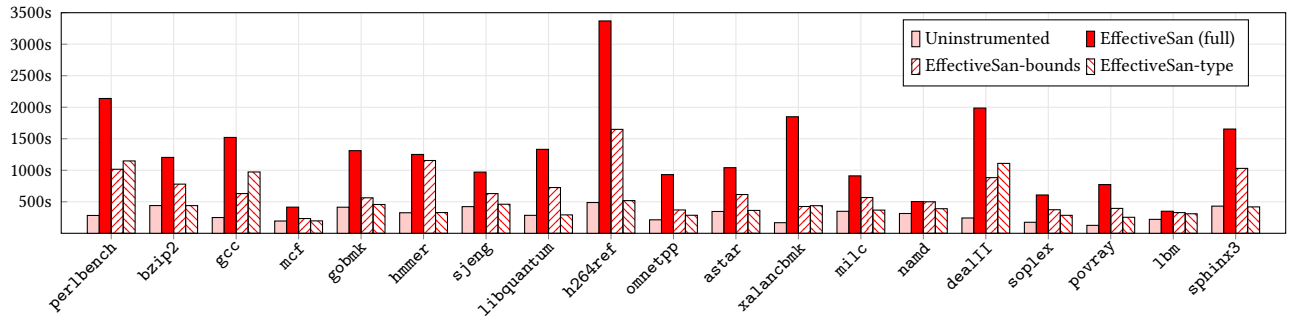

Figure 8. EffectiveSan SPEC2006 timings in seconds. We test three EffectiveSan variants: EffectiveSan (full instrumentation), EffectiveSan-bounds (object bounds checking only), and EffectiveSan-type (type cast checking only).

- `bzip2`/`lbm` confuses fundamental types (`lbm` case also reported in [15]).
- `perlbench` reusing memory (as a different type) rather than explicitly freeing it.
- `perlbench` frequently confuses (T `*`) with (T `**`).
- `perlbench`/`povray`'s ad hoc implementation of C++-style inheritance by defining structures with a common shared prefix, and casting to and fro.

Two type errors from the `xalancbmk` benchmark relate to bad C++ downcasting, similar to the class of errors detectable by CaVer, TypeSan and HexType. The first arises from:

```
SchemaGrammar& sGrammar =
  (SchemaGrammar&) grammarEnum.nextElement();
```

This operation represents a downcast from the base class (`Grammar`) (returned by `nextElement`) to the derived class (`SchemaGrammar`). However, at runtime, `nextElement` may also return a (`DTDGrammar`), which is neither a base nor derived class of (`SchemaGrammar`), and thus the downcast is invalid. A second case arises from an invalid downcast from a (`DOMDocumentImpl`) to a (`DOMElementImpl`). In both cases, the result of the bad cast is only used to access virtual methods from a shared base class. The code relies on undefined behavior, namely, that the virtual function tables of the derived classes are compatible.

Some type errors relate to *type abuse*, i.e., likely deliberate type errors introduced by the programmer. For example, `perlbench` and `povray` use an idiom that confuses `structs` with shared common prefixes as an ad hoc implementation of C++-style inheritance, e.g.

```
struct Base { int x; float y; };
struct Derived { int x; float y; char z; };
```

The (`Base`) and (`Derived`) structure types are incompatible ([16] §6.2.7), thus accessing an object of one type through the other is undefined behavior ([16] §6.5.0 ¶7). Such idioms may break the compiler's *Type-Based Alias Analysis* (TBAA) [5] assumptions and cause programs to be mis-compiled—a known problem for `perlbench` [13]. The code can be re-factored as follows to avoid type errors:

```
struct Derived { struct Base base; char z; };
```

Alternatively `unions` or standard C++ `classes` with inheritance can be used.

Memory Errors. In addition to previously reported [32] memory errors in `perlbench` and `h264ref`, EffectiveSan detects the following sub-object bounds overflows:

- `gcc` overflows the (`mode`) field of type (`rtx_const`) to access structure padding inserted by the compiler.
- `h264ref` overflows the (`blc_size`) field of an object of type (`InputParameters`).
- `soplex` underflows the (`themem1`) field of an object of type (`UnitVector`).

The `soplex` underflow appears to be intentional (it is documented in the source code comments), and relies on the compiler not inserting padding between fields.

Interestingly, the `gcc` error is not reported by MPX [31]. This is possibly because MPX assumes the static type (`int[]`) is correct and does not narrow. In contrast, EffectiveSan matches the static type against the first field of dynamic type (`rtx_const`), implying much narrower bounds. Furthermore, EffectiveSan does not report false positives that affect other tools. For example, `xalancbmk` performs container-style subtraction from the base of a structure, which is reported as a sub-object bounds overflow by MPX [31]. However, this is not considered a sub-object overflow by EffectiveSan, since the operation involves a cast to (`char *`), resetting the bounds to the containing object.

Discussion. As noted above, some issues found by EffectiveSan correspond to intentional type/memory abuse introduced by the programmer, and not unintentional bugs. Even memory errors, such as the `soplex` sub-object bounds underflow (detailed above), may be intentional. EffectiveSan does not currently distinguish between intentional abuse and unintentional bugs, as such a distinction relies on application rather than language semantics, and is therefore best left to the programmer.

That said, even exposing intentional type/memory abuse can be useful, such as for code quality or standards compliance reasons. Removing abuse may also help isolate more

serious issues. For example, `perlbench` is rife with type error abuse—resulting in a large error log—which makes finding "real" bugs more difficult. Type errors may also identify opportunities for code refactoring. For example, EffectiveSan detects multiple type errors in the `povray` benchmark relating to an idiosyncratic implementation of C++-style inheritance using C-style `structs` with overlapping layouts. This was surprising, given that `povray` itself is implemented in C++, and is possibly an artifact of a previous C to C++ transition. Such idioms also affect code quality, since the object hierarchy is represented in a non-standard way, affecting code readability. The `povray` type errors can be resolved by switching to standard C++ `classes` and inheritance. Indeed, modern versions of `povray`[7] have done so. Finally, some type errors may help identify *Custom Memory Allocators* (CMAs) used by the program. Such CMAs can be replaced with standard allocators to help improve the accuracy of EffectiveSan and related tools, as was done with the SPEC2006 benchmarks.

6.2 Performance

Timings. To evaluate performance, we test the EffectiveSan prototype against the SPEC2006 benchmark suite [13]. All experiments are run on a Xeon E5-2630v4 processor (clocked at 2.20GHz) with 32GB of RAM. The results are shown in Figure 8. Here we evaluate three different variants:

- EffectiveSan (full): full EffectiveSan instrumentation;
- EffectiveSan-bounds: protects object bounds only;
- EffectiveSan-type: protects bad C/C++ casts only.

EffectiveSan-bounds protects object bounds only by replacing type check instrumentation (rules Figure 3(a)-(d)) with a simpler (`bounds_get`) function that returns the allocated object bounds without checking whether the type is correct. The object bounds are calculated from the object's dynamic type T, i.e., by sizeof(T). EffectiveSan-type restricts type checking to C/C++-style cast operations only (rule Figure 3(d)) and all other instrumentation is removed. Unlike EffectiveSan (full), rule 3(d) is applied regardless of whether the resulting pointer is used. The main motivation for the variants is to enable a meaningful comparison with related tools such as AddressSanitizer and HexType. We also compare against the uninstrumented baseline at (`-O2`).

The additional performance overheads of EffectiveSan, EffectiveSan-bounds and EffectiveSan-type are 288%, 115% and 49% respectively (Figure 8).[8] Unsurprisingly, EffectiveSan with full comprehensive instrumentation enabled has the highest overhead, at 288% overall. However, this mode makes no assumptions about the type and bounds of objects, so is the most likely to find errors in the program. Reducing the instrumentation trades error coverage for performance, as demonstrated by the (EffectiveSan-bounds) and (EffectiveSan-type) variants, with runtime overheads of 115% and 49% respectively.

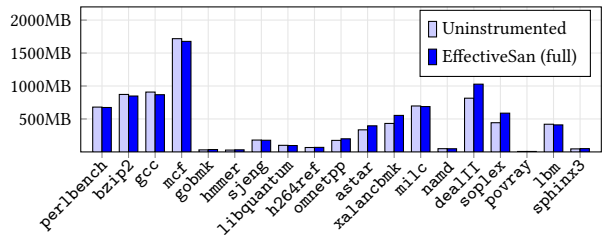

Figure 9. EffectiveSan (full) memory usage in MB.

Tool Comparison. To understand the cost of bounds checks, EffectiveSan-bounds can be compared to more specialized sanitizers that protect object or allocation bounds only, such as AddressSanitizer (73% [32], or 92% reported in [8]), SoftBound (67% [28] for partial SPEC2000/SPEC2006; between 60-249% for 4/19 SPEC2006 [21]; ~100% for 6/19 SPEC2006 [31]), MPX (~200% for SPEC2006 [31]), BaggyBounds (60% [1] for SPEC2000), and LowFat (54% [8] for SPEC2006). The overhead of EffectiveSan-bounds is higher than most tools, but is not meant to replace specialized solutions. EffectiveSan's meta data representation is primarily designed for type checking, meaning that object bounds must be calculated indirectly from dynamic type information.

EffectiveSan-type's instrumentation is comparable to existing type confusion sanitizers such as CaVer [23], TypeSan [11] and HexType [18]. CaVer reports a 20.0-29.6% overhead for 2/19 SPEC2006 benchmarks, TypeSan a 12.1% overhead for 7/19 benchmarks, and HexType a 3.3% overhead for 7/19 benchmarks. EffectiveSan-type has higher overhead, at 49% for all of SPEC2006. However, these existing sanitizers are specialized for casts between C++ `classes`, which results in far less checking. For example, TypeSan does a total of 5.9 billion type checks for all SPEC2006 C++ benchmarks [11], whereas EffectiveSan (full) does 873.1 billion (Figure 7, excluding bounds checks) and EffectiveSan-type does 361.1 billion, with 264.2 billion from `perlbench`, `gcc`, and `dealII` alone. These existing sanitizers do not handle C programs (`perlbench` and `gcc`) nor C-style casts from `dealII`, which account for most of the additional checks. Our results also show that the overhead-per-check ratio strongly favors EffectiveSan. HexType [18] also uses optimizations such as avoiding tracking for objects that are never cast. In principle, such optimizations could also be adapted to EffectiveSan-type, however EffectiveSan-type is not meant to replace specialized tools. Finally, we note that EffectiveSan is also significantly faster than previous runtime type checking systems for C, such as [25] with a 35×-133× slowdown for SPEC95.

[7] http://www.povray.org

[8] The protection and overhead of each EffectiveSan variant shown in Figure 8 is not meant to be additive. For example, only full EffectiveSan can detect non-cast type errors and sub-object bounds overflow errors.

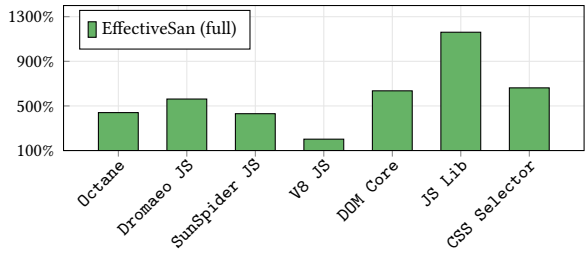

Figure 10. EffectiveSan relative performance for Firefox and various standard browser benchmarks.

Memory. The memory overheads (peak resident set size) for EffectiveSan are shown in Figure 9. Overall we see that EffectiveSan introduces a ~12% memory overhead, which is comparable to the ~3% overhead introduced by the underlying low-fat pointer implementation [8]. This also suggests that the memory overheads introduced by object and type meta data are modest. Many existing sanitizers that use shadow memory report higher memory overheads, e.g., AddressSanitizer [32] at 237% overhead.

6.3 Web Browser Evaluation

We evaluate EffectiveSan against Firefox [10] in order to test complex and multi-threaded software. Firefox is built using EffectiveSan after: (1) disabling `jemalloc`[9] (2) replacing components written directly in assembly (EffectiveSan assumes C/C++ source code), and (3) applying a one-line patch that removed stack object ordering assumptions that are incompatible with the low-fat stack allocator [8]. Aside from `jemalloc`, we instrument Firefox "as is" without replacing other *Custom Memory Allocators* (CMAs), the same approach used by [11]. Finally, we note that, as far as we are aware, EffectiveSan is the first full type and sub-object bounds checker used to build a web browser, demonstrating the compatibility of our approach. Other sub-object bounds checkers, such as MPX and SoftBound, do not support multi-threaded code [31] required for browsers.

The results for standard browser benchmarks are shown in Figure 10. Overall we see that EffectiveSan (full) introduces a 422% overhead compared to the uninstrumented baseline, which is (1.5×) the additional overhead compared to the SPEC2006 results. Although the overhead for Firefox is higher, our result is consistent with CaVer (2.6× for 2/19 SPEC2006) [23], TypeSan (2.8× for 7/19 SPEC2006) [11], and HexType (55× for 7/19 SPEC2006) [18] which similarly report higher overheads for Firefox relative to the SPEC2006 benchmarks. In [11] it is noted that Firefox creates large numbers of temporary objects which leads to increased overheads for tools implementing type checking.

[9] Disabling `jemalloc` is also standard practice for compiling Firefox with AddressSanitizer.

EffectiveSan detects multiple issues for Firefox summarized below. Most issues relate to *type abuse* (similar to our SPEC2006 results) or CMAs, including:

- Multiple instances of casts between types that are equivalent modulo template parameters. For example, an object of type (`T<U*>`) being cast to (`T<void*>`) and vice versa, such as (`nsTArray_Impl<void*>`) being confused with (`nsTArray_Impl<PVRLayerParent*>`), etc.
- Multiple instances of type abuse similar to our SPEC2006 results, including: casting to container types and casting structures to fundamental types (e.g., `int[]`).
- Multiple errors relating to the use of CMAs. For example, function (`XPT_ArenaCalloc`) is one such CMA that returns objects typed with an internal allocator structure (`BLK_HDR`). This results in type errors, e.g., (`BLK_HDR`) versus (`XPTMethodDescriptor`), etc.

The latter demonstrates how type errors can sometimes identify CMAs. Such CMAs can be replaced with standard allocators to better assist dynamic analysis tools such as EffectiveSan. However, due to the size and complexity of the Firefox code-base, such an exercise is left as future work.

7 Conclusion

In this paper, we have proposed dynamic typing as a general method for comprehensive type and memory error detection in C/C++ programs. We also presented EffectiveSan, a practical implementation of dynamic typing using a combination of low-fat pointers, meta data, and type/bounds check intrumentation. We have evaluated EffectiveSan against the SPEC2006 benchmark suite and Firefox, finding several new errors. We also show that EffectiveSan is effective at detecting sub-object bounds errors, one of only a few tools that can do so, while being compatible with multi-threaded environments and preserving the Application Binary Interface.

The scope for future work is broad. EffectiveSan's method for tracking dynamic type information can likely be generalized to other useful properties, enabling new classes of C/C++ sanitizers. The performance of our prototype can also likely be improved as new optimizations are implemented.

A SPEC2006 Modifications

For our SPEC2006 experiments, the following CMAs/wrappers were replaced with the standard (`malloc`) equivalent:

`Perl_malloc`, `safemalloc`, `Perl_safesysmalloc`, `BZALLOC`, `xmalloc`, `pov_malloc`, `MallocOrDie`, `MemoryManager::allocate`, `XMemory::operator new`, `__ckd_malloc__`, `__mymalloc__`

The analogous CMAs/wrappers for (`realloc`), (`calloc`) and (`free`) were also replaced.

References

[1] P. Akritidis, M. Costa, M. Castro, and S. Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense Against Out-of-Bounds Errors. In *USENIX Security Symposium*. USENIX.

[2] T. Austin, S. Breach, and G. Sohi. 1994. Efficient Detection of All Pointer and Array Access Errors. In *Programming Language Design and Implementation*. ACM.

[3] H. Boehm and M. Weiser. 1988. Garbage Collection in an Uncooperative Environment. *Software Practical Experience* 18, 9 (1988).

[4] Debugging Information Format Committee. 2010. *DWARF Debugging Information Format V4*.

[5] A. Diwan, K. McKinley, and J. Moss. 1998. Type-based Alias Analysis. In *Programming Language Design and Implementation*. ACM.

[6] G. Duck and R. Yap. 2016. Heap Bounds Protection with Low Fat Pointers. In *Compiler Construction*. ACM.

[7] G. Duck and R. Yap. 2018. An Extended Low Fat Allocator API and Applications. *CoRR* abs/1804.04812 (2018).

[8] G. Duck, R. Yap, and L. Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers. In *Network and Distributed System Security Symposium*. The Internet Society.

[9] F. Eigler. 2003. Mudflap: Pointer Use Checking for C/C++. In *GCC Developer's Summit*.

[10] Firefox 2018. Firefox Web Browser. https://www.mozilla.org/.

[11] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe. 2016. TypeSan: Practical Type Confusion Detection. In *Computer and Communications Security*. ACM.

[12] C. Hathhorn, C. Ellison, and G. Roşu. 2015. Defining the Undefinedness of C. In *Programming Language Design and Implementation*. ACM.

[13] J. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *Computer Architecture News* 34, 4 (2006).

[14] Intel Corporation. 2018. Intel 64 and IA-32 Architectures Software Developer's Manual.

[15] I. Ireland. 2013. *SafeType: Detecting Type Violations for Type-Based Alias Analysis of C*. Master's thesis. University of Alberta.

[16] ISO. 2011. *Programming Languages – C*. ISO/IEC 9899:2011.

[17] ISO. 2017. *Programming Languages – C++*. ISO/IEC 14882:2017.

[18] Y. Jeon, P. Biswas, S. Carr, B. Lee, and M. Payer. 2017. HexType: Efficient Detection of Type Confusion Errors for C++. In *Computer and Communications Security*. ACM.

[19] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. 2002. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference*. USENIX.

[20] S. Kell. 2016. Dynamically Diagnosing Type Errors in Unsafe Code. In *Object-Oriented Programming, Systems, Languages, and Applications*. ACM.

[21] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. 2014. Code-pointer Integrity. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX.

[22] A. Kwon, U. Dhawan, J. Smith, T. Knight, and A. DeHon. 2013. Low-fat Pointers: Compact Encoding and Efficient Gate-level Implementation of Fat Pointers for Spatial Safety and Capability-based Security. In *Computer and Communications Security*. ACM.

[23] B. Lee, C. Song, T. Kim, and W. Lee. 2015. Type Casting Verification: Stopping an Emerging Attack Vector. In *USENIX Security Symposium*. USENIX.

[24] LLVM. 2018. `http://llvm.org`.

[25] A. Loginov, S. Yong, S. Horwitz, and T. Reps. 2001. Debugging via Run-Time Type Checking. In *Fundamental Approaches to Software Engineering*. Springer.

[26] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. Watson, and P. Sewell. 2016. Into the Depths of C: Elaborating the De Facto Standards. In *Programming Language Design and Implementation*. ACM.

[27] Microsoft. 2013. *Microsoft Security Intelligence Report: Featured Intelligence*. Volume 16.

[28] S. Nagarakatte, Z. Santosh, M. Jianzhou, M. Martin, and S. Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Programming Language Design and Implementation*. ACM.

[29] S. Nagarakatte, J. Zhao, M. Martin, and S. Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *International Symposium on Memory Management*. ACM.

[30] G. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. 2005. CCured: Type-safe Retrofitting of Legacy Software. *Transactions on Programming Languages and Systems* (2005).

[31] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer. 2017. Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches. *CoRR* abs/1702.00719 (2017).

[32] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference*. USENIX.

[33] UBSan 2018. Undefined Behavior Sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[34] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen. 2010. PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs. In *Information, Computer and Communications Security*. ACM.