TypeSan: Practical Type Confusion Detection

Istvan Haller*† istvan.haller@gmail.com
Yuseok Jeon‡ jeon41@purdue.edu
Hui Peng‡ peng124@purdue.edu
Mathias Payer‡ mathias.payer@nebelwelt.net
Cristiano Giuffrida*† giuffrida@cs.vu.nl
Herbert Bos*† herbertb@few.vu.nl
Erik van der Kouwe*† vdkouwe@cs.vu.nl

ABSTRACT

The low-level C++ programming language is ubiquitously used for its modularity and performance. Typecasting is a fundamental concept in C++ (and object-oriented programming in general) to convert a pointer from one object type into another. However, downcasting (converting a base class pointer to a derived class pointer) has critical security implications due to potentially different object memory layouts. Due to missing type safety in C++, a downcasted pointer can violate a programmer's intended pointer semantics, allowing an attacker to corrupt the underlying memory in a type-unsafe fashion. This vulnerability class is receiving increasing attention and is known as *type confusion* (or *bad-casting*). Several existing approaches detect different forms of type confusion, but these solutions are severely limited due to both high run-time performance overhead and low detection coverage.

This paper presents TypeSan, a practical type-confusion detector which provides both low run-time overhead and high detection coverage. Despite improving the coverage of state-of-the-art techniques, TypeSan significantly reduces the type-confusion detection overhead compared to other solutions. TypeSan relies on an efficient per-object metadata storage service based on a compact memory shadowing scheme. Our scheme treats all the memory objects (i.e., globals, stack, heap) uniformly to eliminate extra checks on the fast path and relies on a variable compression ratio to minimize run-time performance and memory overhead. Our experimental results confirm that TypeSan is practical, even when explicitly checking almost all the relevant typecasts in a given C++ program. Compared to the state of the art, TypeSan yields orders of magnitude higher coverage at 4–10 times lower performance overhead on SPEC and 2 times on Firefox. As a result, our solution offers superior protection and is suitable for deployment in production software. Moreover, our highly efficient metadata storage back-end is potentially useful for other defenses that require memory object tracking.

*Vrije Universiteit Amsterdam
†Amsterdam Department of Informatics
‡Purdue University

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS'16, October 24 - 28, 2016, Vienna, Austria

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4139-4/16/10...$15.00
DOI: http://dx.doi.org/10.1145/2976749.2978405

CCS Concepts

•Security and privacy → Systems security; Software and application security;

Keywords

Type safety; Typecasting; Type confusion; Downcasting

1. INTRODUCTION

Type confusion bugs are emerging as one of the most important attack vectors to compromise C++ applications. C++ is popular in large software projects that require both the modularity of object-oriented programming and the high efficiency offered by low-level access to memory and system intrinsics. Examples of large C++ programs are Google Chrome, large parts of Microsoft Windows and Firefox, and the Oracle Java Virtual Machine. Unfortunately, C++ enforces neither type nor memory safety. This lack of safety leads to type confusion vulnerabilities that can be abused to attack certain programs. Type confusion bugs are an interesting mix between lack of type safety and lack of memory safety. Generally, type confusion arises when the program interprets an object of one type as an object of a different type due to unsafe typecasting—leading to reinterpretation of memory areas in different contexts. For instance, it is not uncommon for a program to cast an instance of a parent class to a descendant class, even though this is not safe if the parent class lacks some of the fields or virtual functions of the descendant class. When the program subsequently uses the fields or functions of the descendant class that do not exist for the given object, it may use data, say, as a regular field in one context and as a virtual function table (vtable) pointer in another. Exploitable type confusion bugs have been found in a wide range of software products, such as Adobe Flash (CVE-2015-3077), Microsoft Internet Explorer (CVE-2015-6184), PHP (CVE-2016-3185), and Google Chrome (CVE-2013-0912). This paper shows how to detect type confusion with higher detection coverage and better performance than existing solutions.

TypeSan: always-on type checking Current defenses against type confusion [20, 17] are impractical for production systems, because they are too slow, suffer from low coverage,

Checker	xalancbmk	soplex	omnetpp	dealII
CaVer	24 thousand	0	0	0
TypeSan	254 mln	209 thousand	2.0 bln	3.6 bln

Table 1: Coverage achieved by the type checkers on SPEC. The numbers represent the number of downcasts verified by each of these systems while executing the reference workloads. The CaVer numbers are from the original paper to ensure a fair comparison.

Checker	xalancbmk	soplex
CaVer	29.6%	20.0%
TypeSan	7.1%	1.8%

Table 2: Performance overhead achieved by the type checkers on SPEC. The CaVer numbers are taken from the original paper to ensure a fair comparison. The comparison only uses the applications CaVer has been evaluated on by its authors.

and/or only support non-polymorphic classes. The greatest challenge in building an always-on type checker is the need for per-object metadata tracking which quickly becomes a bottleneck if the program allocates, frees, casts, and uses objects at high frequency (e.g., on the stack).

To address the high overhead and the low coverage of existing solutions, we present TypeSan, an explicit type checking mechanism that uses LLVM-based instrumentation to enforce explicit type checks. Compared to previous work, TypeSan provides extended coverage and massively reduced performance overhead. Our back-end uses a highly efficient metadata storage service (based on a shadowing scheme with a variable compression ratio) to look up types from pointers. This limits the amount of data written for large allocations (such as arrays of objects) while at the same time supporting efficient and scalable lookups, requiring only 3 memory reads to look up a type. We envision this new type of metadata storage to be also useful for other sanitizers, e.g., to verify memory safety, and we plan to explore further applications in future work.

We primarily envision TypeSan as an always-on solution, making explicit type checks practical for commodity software. Used in attack prevention mode, TypeSan-hardened binaries are shipped to end users and terminate the program on bad casts, thereby preventing zero-day type confusion exploits. Combined with liveness reports for modern software (like the Google Chrome and Mozilla Firefox crash reporters), such a deployment signals the developers about potentially missing type checks. In addition, TypeSan can be used in software testing where TypeSan identifies potential bad casting in the source code. In relaxed mode, TypeSan simply logs all bad casts to scan for underlying vulnerabilities, e.g., when running a test suite.

We have implemented a prototype of TypeSan for Linux on top of LLVM 3.9. Our prototype implementation is compatible with large source code bases. We have evaluated TypeSan with the SPEC CPU2006 C++ programs and the Firefox browser. Compared to CaVer [17], the current state-of-the-art type confusion detector, we decrease overhead by a factor 3–6 for the SPEC benchmarks they reported on while simultaneously increasing the number of typecasts covered by checks by several orders of magnitude (see Table 1 and Table 2 for more details).

Contributions We make the following contributions:

- A design for high-performance, high-coverage typecast verification on legacy C++ code that is 3–6 times faster than the state-of-the-art detector with lower memory overhead and orders of magnitude more typecasts.
- A thorough evaluation that shows how our design delivers both nearly complete coverage and performance that is suitable for production usage.
- An automatically generated test suite for typecasting verification to ensure that all different combinations of C++ types are properly handled.
- An open-source implementation of our TypeSan design, available at https://github.com/vusec/typesan.

2. BACKGROUND

In this section, we first explain typecasting in C++ and how doing so incorrectly can lead to vulnerabilities. Afterwards, we discuss existing defenses against type confusion.

2.1 Type confusion

Object-oriented programming languages such as C++ allow object pointers to be converted from one type into another type, for example by treating an instance of a derived class as if it were an instance of one of its ancestor classes. Doing so allows code to be reused more easily and is valid because the data layout is such that an object from a derived class contains the fields of its parent classes at the same relative offsets from each other.

In our discussion on the safety of type conversions (or typecasts), we will use the following terminology: the *run-time type* refers to the type of the constructor used to create the object, the *source type* is the type of the pointer that is converted, and the *target type* is the type of the pointer after the type conversion. Since the program may treat objects as if they are instances of their ancestor types, an object pointer should always refer to an object with a run-time type that is either equal to or a descendant of the pointer type. Therefore, a type conversion is always permissible when the target type is an ancestor of the source type. A compiler can verify such casts statically, because if the source type is a descendant of the target type it implies that the run-time type is also a descendant. We refer to this type of conversion as an *upcast*.

If, on the other hand, the target type is a descendant of the source type, the conversion may or may not be permissible depending on whether the run-time type is either equal to or a descendant of the target type. This is impossible to verify in the general case at compile time because the run-time type is not known to the compiler, due to inter-procedural/inter-component data flows. We refer to this type of conversion as a *downcast*. Downcasts require run-time verification to ensure type safety. Incorrect downcasts may allow attackers to exploit differences in the memory layout or semantics of the fields between the target type and run-time type.

The C++ programming language permits both upcasts and downcasts and allows the programmer to specify whether downcasts should be checked at run time. Specifically, the language provides three fundamental types of casts: `reinterpret_cast`, `dynamic_cast`, and `static_cast`. Dynamic casts are enforced at run time with an explicit type check

and are therefore a safe but expensive way to ensure type safety. Static casts on the other hand only verify whether the conversion could be a valid upcast or downcast based on the source and target types. This lack of an online check can easily lead to type confusion when the underlying type observed at run time differs from the expected type in the source code.

As an example, in V8Clipboard in Chrome 26.0.1410.64, we find the following static cast:

```
static_cast<HTMLImageElement*>(node)->cachedImage()
```

Here, the program explicitly casts an image `node` to an `HTMLImageElement` without properly checking that it is of the right type. Unfortunately, `node` could be an SVG image, which is of a sibling class and has a much smaller vtable than `HTMLImageElement`. Note that the program immediately calls `cachedImage()` on the invalid object which leads to a virtual function call that erroneously interprets the memory adjacent to the SVG image's vtable as code pointers.

If the program would check all static casts dynamically, we would not run into the type confusion problem (except for explicitly forced "problems" through reinterpreted casts). However, casting is such a common operation that the overhead of checking all static casts dynamically is significant and therefore, C++ allows the programmer to choose an explicit run-time cast only where "needed" (according to the programmer).

For completeness, we mention that the last form of casting, `reinterpret_cast`, forces a reinterpretation of a memory area into a different type. It allows a programmer to explicitly break underlying type assumptions.

2.2 Defenses against type confusion

In recent years, several projects have tried to address the type confusion problem. There are two main types of approaches: those based on vtable pointers embedded in the objects and those based on disjoint metadata. Solutions based on vtable pointers have the advantage that they do not need to track active objects but they have the fundamental limitation that they cannot support non-polymorphic classes, which do not have vtables, without breaking binary compatibility.

Examples of vtable-based solutions are UBSan [20] and Clang Control-Flow Integrity [4] (CFI). UBSan instruments static casts to execute an explicit run-time check, effectively turning them into dynamic casts. UBSan requires manual blacklisting to prevent failure on non-polymorphic classes. Unfortunately, the existing type check infrastructure that is available in C++ compilers is inherently slow (as it was designed under the assumption that only few dynamic checks would be executed with the majority of checks being static). This is another reason why type-safety solutions did not see wide adoption. Therefore, UBSan is intended as a testing tool and not as an always-on solution due to its prohibitive overhead. Clang CFI is designed to be faster but has not published performance numbers. Moreover, like all solutions in this group, it cannot support non-polymorphic classes.

CaVer [17], on the other hand, uses disjoint metadata to support non-polymorphic classes without blacklisting. Unfortunately, the overhead is still prohibitively high due to inefficient metadata tracking (especially on the stack) and slower checks, reaching up to 100% for some browser benchmarks. Because CaVer cannot rely on vtables (present only in polymorphic objects), it must track all live objects. In particular, CaVer uses red-black trees to track stack-allocated objects and a direct mapping scheme based on a custom memory allocator for heap-based objects. As a consequence, it has to determine the correct memory region for each pointer to be checked and it cannot handle stack objects shared between threads even if proper synchronization is used in the application. In addition, as shown in Section 9.2, CaVer has poor object allocation coverage in practice, ultimately leading to reduced type-confusion detection coverage. For example, CaVer reports only 24k verified casts on xalancbmk and none on all other SPEC CPU2006 C++ benchmarks, while we show that four benchmarks actually have a large amount of relevant casts with their total numbers is in the billions. As such, TypeSan is the first solution that provides efficient and comprehensive protection against type confusion attacks in the field, protecting users from vulnerabilities not found during testing.

In this paper we introduce TypeSan, a generic solution for typecast verification based on object tracking, that supports all types of classes with no need for blacklisting. Moreover, we cover a very large percentage of all relevant casts at an acceptable overhead. Table 3 gives a high-level comparison of the typecast verification solutions presented here. UBSan and Clang CFI are restricted to polymorphic types, with UBSan requiring further blacklisting to handle certain code bases. CaVer and TypeSan also support non-polymorphic types, but this comes at the cost of needing to track the type for each object. CaVer further comes with the limitation that threads cannot share stack objects safely even with proper synchronization. Table 3 and Table 4 show which allocation types are tracked in practice by tracking solutions. See Section 5.1 for a more in-depth discussion of the various allocation types. CaVer officially supports stack and global data, but missed such bad casts in our coverage tests (see Section 9.2).

Checker	Poly	Non-poly	No blacklist	Tracking	Threads
UBSan	✓				✓
Clang CFI	✓		✓		✓
CaVer	✓	✓	✓	✓	limited
TypeSan	✓	✓	✓	✓	✓

Table 3: High-level feature overview of checkers.

Checker	stack	global	new/new[]	malloc family
CaVer	NO	PARTIAL	✓	NO
TypeSan	✓	✓	✓	✓

Table 4: Allocation types tracked by checkers, see Section 5.1 for a detailed discussion.

3. THREAT MODEL

We assume that the attacker can exploit any type confusion vulnerability but is unable to perform arbitrary memory writes otherwise. Our type safety defense mechanism exclusively focuses on *type confusion*. Other types of vulnerabilities such as integer overflows or memory safety vulnerabilities are out of scope and we assume that orthogonal defense mechanisms protect against such vulnerabilities. Our defense mechanism tolerates arbitrary reads as we do not rely on secret information that is hidden from the attacker.

4. OVERVIEW

TypeSan is an extension to the Clang/LLVM compiler [15] that detects invalid `static_casts` (i.e., all instances in the program where an object is cast into a different type without using an explicit run-time check through `dynamic_cast` or an explicit override through `reinterpret_cast`) in legacy C++ programs. Upon detection of an invalid cast, the program is terminated, optionally reporting the cause of the bad typecast. TypeSan is a compiler-based solution and any C/C++ source code can be hardened, without modification, by recompiling it with our modified `clang++` compiler with the `-fsanitize=type` option and linking against the tcmalloc memory allocator [7] using the `-ltcmalloc` linker flag. As we show in Section 9, TypeSan has reasonable performance for usage in production software.

Figure 1 presents an overview of TypeSan. The *instrumentation layer* consists of hooks inserted by the compiler to monitor object allocations and potentially unsafe casts, as well as a static library containing the implementations of those hooks. To perform its task, this layer makes use of two services. The *type management service* encodes type information and performs the actual typecast verification. It includes type information for all the types that may be used at run time. The instrumentation layer uses it to look up type information for new allocations and informs it whenever a cast needs to be checked. Finally, the *metadata storage service* stores a pointer-to-type mapping and can look up the type information of an object about to be typecast. This service allocates a memory area to store the mapping at run time. It provides an operation to bind type information to a pointer and to lookup a previous binding for a pointer.

All mechanisms that explicitly protect from type confusion will incur two forms of run-time overhead: overhead for maintaining metadata (allocating and deallocating objects) and overhead for explicit type checks at cast locations. We designed TypeSan to minimize the allocation/deallocation time overhead. C++ programs are heavily affected by allocator performance, as implicitly shown by how large projects tend to replace the standard memory allocator with high-performance allocators (tcmalloc and other allocators in Chrome, jealloc in Firefox). Many of the objects being allocated may also never be subject to downcasts, making it even more important to minimize the impact of type tracking on such objects.

In order to meet these requirements, TypeSan relies on a clean, uniform front-end design that flags allocations and casts to the back-end system but introduces a completely different mechanism to track type information, called the metadata storage service in our design. Our metadata storage service builds on the memory layout and structure inherently provided by the allocator and uses this structure

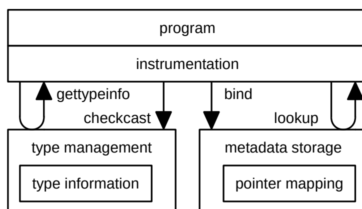

Figure 1: Overview of TypeSan components.

to reduce the access overhead ("aligning" objects with their metadata). Compared to existing disjoint metadata storage layers that use different forms of lookup functions from red-black trees to hashing for pointer range queries, our approach offers fast constant-time updates and lookups.

For the type checking instrumentation and related data structures, we use a design focused on simplicity and cache-efficient traversal. This design is effective even for workloads with an extremely high number of casting operations. Furthermore, for performance-sensitive applications, safe operations can be blacklisted or an approach like ASAP [24] can trade off security against acceptable overhead. This is another key motivation to minimize allocation-time overhead, as it does not scale with the number of instrumented casts in a program, acting as residual overhead instead.

Lastly, the instrumentation layer and the metadata storage service are connected by the instrumentation layer, which uses the Clang/LLVM compiler framework [15] to track allocations, instrument cast operations and extract type information. The instrumentation was designed with completeness in mind, following code patterns discovered in real-world programs as well as basic C/C++ constructs expected to be supported. Our instrumentation also allows full C-C++ inter-operability, a novelty compared to state-of-the-art solutions. This is important as some SPEC programs mix C-style allocation with C++ classes and browsers also use a mixture of C and C++ code.

5. INSTRUMENTATION LAYER

In this section we discuss the design of TypeSan's instrumentation layer. The instrumentation layer interacts with the TypeSan-hardened program by inserting hooks at relevant locations. We first consider the instrumentation of allocations, including the types of allocations we support to be able to track run-time object types with high coverage. Then, we discuss the instrumentation of typecasts to be able to introduce type checks.

5.1 Instrumenting allocations

TypeSan adds a compiler pass to LLVM [15] to detect object allocations and insert instrumentation to store the corresponding pointer-to-type mapping. For each allocated object we store a pointer corresponding to the type layout of the object as a whole. However, keep in mind that downcasts in C++ might be applied not just to the allocated object pointer, but also to pointers internal to a given allocation range, specifically in the case of composite types (arrays, multiple inheritance, nested structs/classes). For example, an object of class `A` containing objects of classes `B` and `C` can be cast to `B*` using the original pointer while a cast to `C*` requires an offset to reference the correct member of class `A`. In this scenario, the type of the internal pointer differs from the type associated at the allocation time, which we need to account for in the design. For performance reasons, we chose to keep the instrumentation of the allocations as simple as possible and to defer handling composite types to the check operation itself (discussed in Section 6). This approach still introduces two additional requirements to our design. First, the metadata storage service must be able to retrieve the mapping for internal pointers (discussed in Section 7). Second, the checker needs access to the offset within the class definition corresponding to the internal pointer. To support the latter, we also track the base pointer of the al-

location besides the type mapping. This design results in simple and low-impact instrumentation, using the metadata storage service to only store two pointers for each allocation in the program. In the following, we describe the allocation types we support as well as their motivation.

In C++, objects may reside in global memory, stack memory, or on the heap. These three kinds of objects have different lifetimes and we therefore instrument them differently. However, other than the location where we insert the hooks to instrument the allocation, the single uniform metadata storage service allows us to to treat objects in different memory areas equally. This simplifies our solution and improves performance because TypeSan must not determine where a pointer lives before looking it up.

The initialization and reuse of the mappings in object metadata depend on the object's memory area. For instance, we can initialize the mappings for *global objects* once using global constructors. However, for stack and heap objects, we need a dynamic approach.

In the case of *stack objects*, we need to notify the metadata storage service to take control over the object. These objects are not put on the regular stack but are instead moved to a specific memory area where metadata tracking is enabled (see Section 7 for details on this operation). With this change, we can use the metadata storage service to create a mapping from the new object pointer to its metadata at allocation time. This design decision of moving the objects of interest to a separate location brings additional benefits from a tracking perspective, since the memory location occupied by a previously tracked stack object will only be reused for another tracked object. During allocation, the new object will overwrite the old metadata, removing it from the system permanently. This allows us to persist the metadata mapping after the lifetime of a stack object, removing the need for explicit cleanup.

A special class of stack objects (often ignored in existing solutions) arises when the program passes classes or structs by value as function arguments. To address this special case, TypeSan uses the same approach applied by the current SafeStack implementation in LLVM, moving such objects to a separate stack[1].

Not all stack objects need tracking and we optimize our solution by omitting allocation instrumentation wherever we can prove that the program will never cast the allocated stack objects. To be conservative, we verify whether the function itself or the functions it may call perform any relevant casts. We assume that any indirect (including virtual) or cross-module function calls may perform downcasts, because for these cases we cannot statically determine that they do not. Using this approach, we reduce overhead without missing any checks. It is worth noting that our approach is more conservative than CaVer's, which optimistically considers that such callees never attempt casts within their respective call-graphs[2].

For *heap objects*, we add instrumentation after calls to the `new` and `new[]` operators as well as the traditional `malloc` family of allocation functions. Although C++ structs and classes are expected to be allocated using `new` (to ensure calls to the appropriate constructors), we observed that one of the four SPEC benchmarks with downcasts, uses `malloc/realloc` for allocating its C++ objects. Specifically, the soplex benchmark uses `malloc/realloc` to handle allocations within its core `DataSet` class, which acts like an object pool. Other classes, such as `SVSet`, maintain pointers to objects managed by a particular `DataSet`. As these pointers are also subject to downcasting, it is critical to track `malloc/realloc` in order to have type information available for checking. Tracking heap deallocation is not necessary as we built the metadata storage service to be robust and to clean stale metadata. This ensures that such metadata cannot influence type checks in case of an accidentally missed deallocation. More details can be found in Section 7.

While inferring the allocation type and array size is trivial for `new` and `new[]` (as it is part of the syntax), this is more complicated for the `malloc` family of functions. We traverse the intermediate representation code to look for cast operations applied to the resulting pointer to find the allocation type. This method might fail for type-agnostic allocation wrappers, but such wrappers can easily be added to the set of allocation functions which we track. Array allocations can be tricky when trying to infer the element count from a `malloc`-like call-site, but our tracking scheme was designed to be agnostic to array sizes, thus mitigating potential issues. In practice we found no coverage issues when evaluating TypeSan against SPEC, showcasing our ability to track relevant heap objects with our solution.

An interesting point in heap allocations is support for allocations within the standard template library (STL), which countless applications use for their core data structures. Luckily, STL's template-based design means that all the code related to data structures is located within headers included into every source file. This includes all their allocation wrappers, which are also templated and instantiated on a per-type basis. We confirmed that our instrumentation correctly picks up the allocations within the STL data structures and we successfully check the downcasting operations applied to the individual elements.

5.2 Instrumenting typecasts

Whenever TypeSan encounters a downcast operation (from a base class to a derived class), it inserts a call to our typecasting verification function. Such a cast is present in the code either when performing a `static_cast` operation between appropriate type or when using an equivalent construct such as static C-style casts. In practice, downcasting can exhibit two types of behavior and we optimize our checker to support each one specifically. In the general case, the result of the static cast is the source pointer itself (with no added offset), but with a new source-level type. This happens when the source base type is the primary base class of the derived type, which is always the case when casting without multiple inheritance. In this case, the TypeSan instrumentation calls the checker with the value of the pointer and an identifier corresponding to the destination type. If classes use multiple inheritance it can happen that a cast operation occurs from a secondary base class to a derived type. In this scenario, a negative offset is added to the source pointer as transformation from an internal pointer (to the secondary base) to the base pointer of the object. The checker needs information about the resulting pointer to infer the appropriate offset within the structure layout, but in case of type confusion the negative offsets might make a valid pointer go out of bounds, making it impossible to in-

[1] http://reviews.llvm.org/D14972

[2] We reported this issue to the authors of CaVer.

fer the appropriate type information for the object pointed to. For this reason, TypeSan calls a second version of the checker in this instance, which takes both the source and destination pointers as well as the type identifier for the resulting type.

6. TYPE MANAGEMENT SERVICE

TypeSan manages metadata on a per-allocation basis. Every block allocated by a single `malloc` call, `new` operator, global variable, or stack variable is associated with at most a single pointer to a type information data structure. This data structure therefore encodes all permissible casts using pointers pointing into the allocated block. Any object can be cast back to itself using the original pointer, but composition and inheritance create additional opportunities for casts. For example, a pointer to an object can be cast to any of its base classes and a pointer to a field of an object can be cast to the type of the field (and transitively to any type on the inheritance chain). In this section, we discuss the data structures used to encode this information.

The type management service is responsible for associating type layouts with allocation sites and using these layouts to validate downcast operations. Type checking at its core can be divided into two steps. The first one is the ability to infer the allocation-time type associated with the pointer resulting from the typecast. This is the most derived type associated with the particular offset (from the pointer to the allocation base) within the original allocation. Once this information is known, the second part of the type check involves the comparison of the allocation-time type with the type specified in the cast. The layout of the latter must be compatible with the former for the cast to be valid. The data structures employed by this service share the same purpose as the `THTable` structure in CaVer, but we further optimize the type checks by dividing it in in two phases.

In the following sections, we describe the data structures TypeSan uses to perform these operations.

6.1 Type layout tables

Type layout tables describe each relevant offset within an allocation to enable fast lookups of the offset-specific type information during a check. Specifically, a type layout table is a list of mappings of unique offsets to data fields corresponding to nested types. The list (array) starts with an entry for offset 0 containing the unique hash corresponding to the type as a data field. The layout tables incorporate nested types in one of two ways. As a first option, the nested types can be flattened into a type layout for good cache efficiency during traversal. Alternatively, they can be separated via an indirection for better space efficiency. Flattening a nested type involves injecting an offset-adjusted copy of its type layout into the containing type, where its type layout is copied and adjusted into the layout of the containing type.

An avid reader may notice that flattening can invalidate the property mentioned earlier that offsets in the type layout table are unique. After all, a nested class might occur at offset 0 (for example with primary base classes). This is where the second part of our type check, the layout compatibility check, comes into play. A class which includes a nested class at offset 0 is practically layout compatible with the latter. Intuitively, if the next type class has another class at offset 0 of object, then we can always use the object as representative for both types. TypeSan tracks this relationship in *type relationship tables* (see Section 6.2). Thus, the type layout table only needs to track the (unique) "derived-most" type for matching offsets.

As mentioned earlier, flattening is not compulsory, since the type layout table also supports indirection. In this mode, the data element of a particular entry includes a pointer to the type layout table corresponding to the nested type. In addition, TypeSan adds a sentinel element to the type table to mark the end of the nested type. This allows the traversal code to infer quickly whether or not it should follow the indirection. It skips the sentinel element if its offset overlaps with an existing entry in the table to maintain uniqueness.

Flattening generally improves performance at the cost of space. While TypeSan mostly uses the flattened mode, it uses the non-flattened mode to generate an efficient array abstraction as the array length may be dynamic (in the lack of a way to optimize for performance we optimize for space). In particular, it replaces each nested array with a single indirect entry to the type layout table of the array element type, allowing TypeSan to support nested arrays of any size, without degrading checking speed.

The property of enforcing unique offsets in the type layout table allows us to implement efficient traversal by ordering the entries by offset. During indirection, the type management service updates the offset that is being searched to match the follow-up type layout table (which is just a subset of the original type). In the case of a nested array, it updates the offset to represent the offset into the corresponding array element, instead of the array itself—using the array-stride as input. This is supported, by including the overall size of the type (including requested alignment) as part of the type layout table.

When the instrumentation adds metadata at an allocation site, it simply requests a type layout table that corresponds to the type that is allocated. Type layout tables are generated once for each type, by recursively traversing the type information in LLVM to find all nested elements. For potential array allocations, it marks the pointer to the type layout table to signal additional processing of the offset. This processing is identical to how we deal with nested arrays. Accidentally marking an allocation as being of type array does not affect correctness, it just involves a couple of extra instructions being executed during type lookup. As such our static analysis does not have to be complete as long as it is conservative in identifying allocations of single elements.

6.2 Type relationship tables

In the second stage of the type check we check for raw pointer compatibility between the type identified for the pointer and the type defined in the cast. Such compatibility typically happens between derived classes and their primary base class. As mentioned earlier, another case of compatibility happens between unrelated types if one of the types is nested in the other at offset 0.

Furthermore, CaVer defined the concept of phantom classes: derived classes with no additional members compared to their base class. Sometimes the program downcasts base classes to such phantom classes, resulting in benign bad casts. Thus we also include phantom classes in our compatibility model. Using the compatibility model, we generate a set of type hashes corresponding to the compatible types for each class in the program and refer to it as a *type relationship table*. Once TypeSan has extracted the type of a

pointer from the type layout table, it checks the corresponding type relationship table for the membership of the type defined in the cast. The operation needs to find an exact match to verify a cast. If it finds no match, it reports an error of a mismatched type.

Currently we implement sets as simple vectors, with hashes ordered according to the type hierarchy. We found this solution to be adequate in terms of speed, but we can easily replace it with alternative set implementations, such as the fast LLVM bitset variant. Doing so is easy as the type relationships table is conceptually nothing more than a set as a result of the split of the type information into separate layout and relationship tables.

By having the phantom classes be first-class members of the type relationship tables, we ensure uniform support for them without performance degradation. In contrast, the publicly released CaVer code requires a type check restart for every phantom class if normal inheritance fails to find a match.

6.3 Merging type information across source files

Generating large amounts of type information may necessitate merging across different source files—an expensive and potentially difficult operation. We rely on the One Definition Rule (ODR) in C++ to minimize the need to merge information across the project. ODR states that every class definition across all source files linked together needs to be identical. C++ also requires nested and base types to be fully defined in the source files that use them. As a result, the type layout information for the same type within different source files is always identical. The same is true for the type relationship tables—except their phantom class entries. Since the phantom classes represent derived classes, the set of phantom classes can easily change from one source file to another, and merging may be necessary. TypeSan uses a strategy where it only needs to merge these entries in the type relationship tables to minimizing the merging cost. Any program violating the ODR would trigger error reports in TypeSan. This would be correct, since violating the ODR is type confusion in itself.

7. METADATA STORAGE SERVICE

In this section, we discuss the metadata storage service, which handles storage of metadata at run time. This service allows us to map from object base addresses to type layout tables at run time. Key requirements for our metadata storage service are (i) fast update operations and (ii) range-based lookups (to support interior pointers due to composition). Related work [17, 2] has used alignment-based direct mapping schemes to track complex metadata, relying on dedicated custom memory allocators. Such systems often run into problems for stack-based memory allocations [17] where the allocator has no detailed knowledge of allocations. As a result we designed METAlloc [12], a novel object tracking scheme, based on variable compression ratio memory shadowing, which solves these issues and allows us to have an efficient and uniform metadata storage service for all allocation types.

Variable compression ratio memory shadowing relies on the assumption that all objects on a certain memory page share the same alignment. A uniform alignment guarantees that every alignment-sized slot within the page corresponds to a single object, enabling the tracking of metadata at the level of slots instead of objects, while preserving correctness. Each page can thus be associated with an alignment and a metadata range, including as many entries as the number of alignment-sized slots in the page. Such a mapping simplifies storage of metadata and allows us to assume a certain layout for objects on a per-page basis. Given a page table-like infrastructure, the mapping allows finding metadata corresponding to any particular pointer in constant time, by using the alignment to look up the slot index and to retrieve the metadata stored for the appropriate slot. This mapping also mirrors traditional page tables as the alignment and the base address of the metadata range can be compressed into a single 64-bit value (since pointers only require 48 bits). We call this data-structure the *metadata directory*. Figure 2 shows the mapping operation from any pointer to the corresponding object metadata.

An update operation with this metadata results in finding the metadata for the base address of the object and then updating all entries which correspond to the object range. The number of entries which need to be updated is the number of alignment-size slots that the object spans across, making it critical to select the largest possible alignment to improve update performance. This is where the variable compression ratio comes into play, with large objects having larger alignments, thus their metadata is compressed relative to their size. The system also works with the default 8-byte alignment of objects in existing systems, but the update operation would end up too costly for stack and heap objects. Using alignments which are too large can also generate increased memory fragmentation resulting in unnecessary performance overhead, making it critical to select the most appropriate alignments.

In the case of global memory, the overhead introduced by the update operations rarely affects the performance of a running program, thus we decided to leverage the existing 8-byte alignment applicable for global objects. We update the metadata directory entries to track all loaded sections whenever we detect that a new shared object has been loaded into the address space of the program.

For heap allocations, tcmalloc [7] (the allocator used by the Chrome browser and other Google products) already ensures that every memory page under its control contains only objects of the same size-class and alignment. It enforces this property to efficiently generate free lists, thus ensuring our assumptions for free, without needing to perform any changes to the allocation logic. We only extended tcmalloc to track the metadata directory entries whenever a memory page is associated with a certain size-class, which happens rarely in practice.

Stack allocations are challenging, as they can be subject to ABI restrictions. We mitigate this limitation by moving relevant objects to a secondary stack similar to the operating principles of SafeStack [14]. SafeStack is effective at moving dangerous stack allocations to a secondary stack with practically no overhead and minimal impact on application compatibility. We use the instrumentation layer, as mentioned earlier, to tell the metadata storage service about stack objects, which are then moved to a secondary stack tracked by the metadata directory, where we enforce a 64-byte alignment for each object. ABI restrictions are not applicable, since all tracked stack objects are local variables, whose location can be freely chosen by the compiler. The 64-byte alignment is reasonable as we only move a small

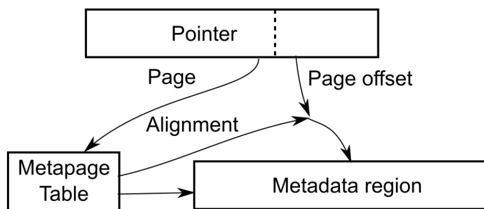

Figure 2: Mapping from a pointer to a metadata entry. The page component is used to look up the start address of the metadata region and the alignment. While the latter together with the page offset is used to compute the offset within the metadata region.

subset of stack objects, thus overall memory fragmentation of the program is limited.

As mentioned earlier in Section 5, deallocation of heap objects is handed internally by the metadata storage service and no extra instrumentation is needed. While the stack contains only tracked objects, thus metadata will always be up to date with allocations, `tcmalloc` does manage all heap objects, including untracked ones. As such, we extended `tcmalloc` to conservatively assume that every new allocation is untracked and to clear stale metadata associated with any new allocation if it detects such. This approach minimizes the overhead when metadata is used sparingly, while ensuring that stale metadata can never affect untracked allocations.

Moreover, our solution is not affected by thread concurrency. The metadata directory is only updated when memory is mapped in from the system. At this point all the entries read/written during this operation are the ones corresponding to the allocation range, which is still in sole control of the running thread. The metadata entries are also updated only during object allocation and the entries written are unique to the allocation range, thus they do not interfere with concurrent lookups. The update operation depends only on the metadata directory entry corresponding to the pointer itself, which cannot be subject to a concurrent write.

8. LIMITATIONS

Our approach is based on an LLVM-instrumentation pass that reasons on the clang and LLVM IR level, therefore source code in either C or C++ is required. Any allocations or casts in assembly are not supported.

As stated in our treat model (and similar to related work), we assume that the attacker has no unrestricted write primitive at their disposal. We therefore do not protect our metadata against malicious writes. Any metadata protection mechanism is orthogonal to this work and equally applies to other protection systems which complement TypeSan.

To support combined protections, TypeSan deliberately imposes as few restrictions and changes in the memory layout as possible. It already integrates with SafeStack, a fast stack protection solution offered by compilers today. Similarly, while TypeSan makes a design assumption about the heap allocator, it is compatible with arguably the two most commonly used custom memory allocators: tcmalloc (Chromium) and jemalloc (Firefox and Facebook). When combined with other memory safety solutions, TypeSan can preserve metadata integrity by construction. When deployed standalone, our design is amenable to existing low-overhead metadata protection techniques, such as APM [8] or write-side SFI (e.g., pointer masking). The overhead of such techniques is amortized across all defense solutions. For example, if we employ SafeStack and we expect SafeStack (already in clang) to be adequately protected moving forward due to recent attacks [8, 19], TypeSan can benefit from the same metadata protection guarantees at no additional cost. Note that TypeSan tolerates arbitrary reads as we do not rely on secret information hidden from the attacker.

Custom memory allocators (CMAs) can prohibit TypeSan from appropriately tracking heap allocations. Unfortunately, this is a fundamental limitation for instrumentation systems which rely on object information tracking. TypeSan uses `tcmalloc` as the back-end allocator. This is a suitable replacement for other general purpose allocators, but objects allocated within CMAs (such as pool or SLAB allocators) will not be tracked by our system.

9. EVALUATION

To show that TypeSan achieves higher coverage and lower overhead than previous solutions, we evaluated our prototype using a number of demanding CPU-intensive (and cast-intensive) workloads. We test Firefox because browsers are a common target for attackers of type confusion vulnerabilities, given the fact that they are usually written in C++ for performance reasons and have a large attack surface because of the fact that they provide a substantial API to foreign Javascript code. We benchmarked Firefox using the Octane [9], SunSpider [10], and Dromaeo [6] benchmarks. Octane and SunSpider focus on Javascript performance, while Dromaeo has subtests for both Javascript and the DOM. Moreover, we implemented our own microbenchmarks to isolate the overhead and report worst-case figures for TypeSan and existing solutions. In addition, we run the SPEC CPU2006 C++ benchmarks, which all heavily stress our allocator instrumentation and some (e.g., dealII and omnetpp) our typecast instrumentation.

In our evaluation, we consider a number of different system configurations. Our baseline configuration compiles with Clang 3.9 [15] at the default optimization levels. The baseline is not instrumented but does use the tcmalloc [7] allocator to report unbiased results, given that tcmalloc generally introduces a speedup that should not be attributed to TypeSan. In addition to the baseline, we have the TypeSan and TypeSan-res configurations. The former instruments every possible cast while the latter does not instrument any casts. The TypeSan-res configuration shows to what extent a system like ASAP [24] can reduce the performance impact of our instrumentation (trading off security) when only a small performance budget is available. We ran our benchmarks on an Intel Core i7-4790 CPU with 16 GB of RAM, using the Ubuntu 15.10 Linux distribution.

9.1 Performance

9.1.1 Microbenchmarks

To verify that TypeSan provides low allocation-time and typecast-time overhead, we created microbenchmarks that measure how long these operations take, both on the stack and on the heap. To compare our results against state-of-

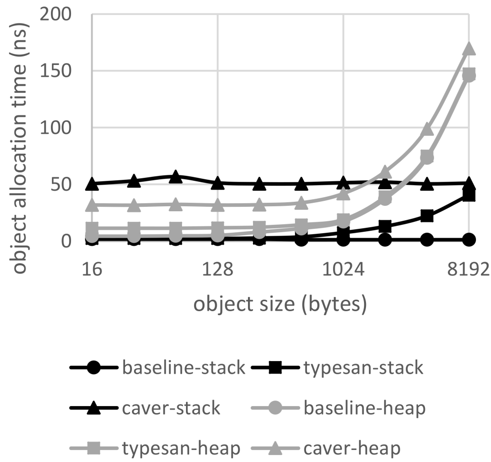

Figure 3: Allocation performance as a function of allocated object size.

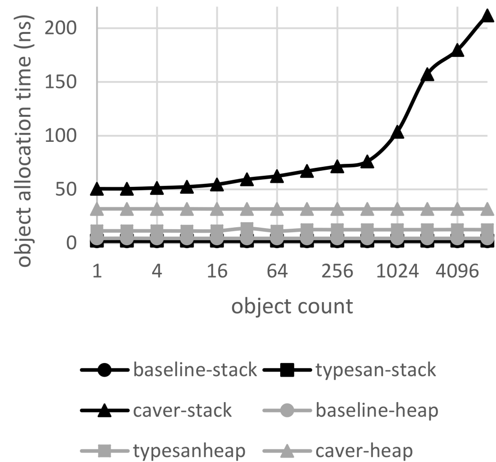

Figure 4: Allocation performance as a function of allocated object count.

the-art solutions, we compiled CaVer [17] from source [16] and configured in the same way as TypeSan—except that we do not use tcmalloc since CaVer ships with its own custom allocator. To prevent the target operations from being removed by optimizations, we switched to the `-O1` optimization level for this experiment. To isolate the overhead, we measured the impact of (i) the number of allocated stack objects and (ii) the object size. The former is important since CaVer tracks stack objects with red-black trees, whose performance degrades with the number of objects. The latter is important since TypeSan needs to initialize multiple metadata entries for large objects, incurring more overhead.

Figure 3 depicts the impact of the object size on allocation performance when no other stack objects are present. Allocating an object on the stack is almost instantaneous for the baseline and takes a fixed but long time for CaVer. For TypeSan, allocation time on the stack is proportional to the object size as multiple metadata entries need to be initialized for large objects. However, even for objects as large as 8KB, TypeSan is still faster than CaVer. Small objects up to 128 bytes take only 0.5ns extra to allocate with TypeSan, while CaVer adds at least 48.8ns even for these small (and common) allocations. On the heap, allocation time grows linearly with the allocation size in all cases. Overall, TypeSan is close in performance to the baseline while CaVer adds considerable overhead. For heap allocations up to 128 bytes, TypeSan adds at most 7.0ns overhead while CaVer adds at least 26.7ns.

We believe that it is unlikely for programs to frequently allocate large, bloated classes without using them and thus hiding the allocation overhead. As further mitigation to the scaling based on the stack object size, it is possible to extend TypeSan to use additional secondary stacks with increased alignments for such large classes. These results support our claims that TypeSan is particularly suitable for applications that allocate many objects, especially on the stack.

Figure 4 and Figure 5 show the impact of the number of allocated stack objects on allocation and typecast performance (respectively), using an object size of 16 bytes. The overhead patterns are in the same region for both scenarios. CaVer's overhead on the stack increases with the logarithm of the number of objects (due to the use of red-black trees)

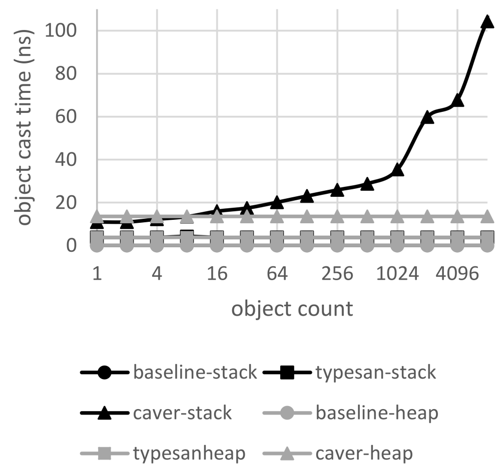

Figure 5: Typecast performance as a function of allocated object count.

while TypeSan's does not depend on the number of allocated objects. Even at relatively low allocation counts, CaVer's stack allocations are much more expensive than TypeSan's.

For every typecast, in turn, TypeSan adds an overhead of only 3.8ns, regardless of whether the object is allocated on the stack or on the heap and regardless of the number of allocated objects. CaVer's typecast overhead is higher in all cases. In particular, the heap overhead is a constant 13.6ns, while the stack overhead starts at 11.0ns and increases with the number of allocated objects.

9.1.2 Performance overhead

Table 5 reports our performance on the SPEC CPU2006 C++ benchmarks [13] and Firefox. The first four SPEC benchmarks perform static typecasts while the others do not. In the latter case, the overhead stems from TypeSan having to still track objects that cannot be statically and conservatively proven not be typecast during the execution. In the default configuration, overheads range from negligible to moderate and the overheads on the benchmarks reported by CaVer [17] are much lower. In particular, CaVer

	casts	TypeSan		TypeSan-res	
dealII	yes	30.8	(0.2)	3.6	(0.1)
soplex	yes	1.8	(0.7)	1.5	(0.8)
omnetpp	yes	27.2	(1.7)	2.4	(3.0)
xalancbmk	yes	7.1	(0.4)	4.4	(0.3)
namd	no	-0.6	(0.1)	-0.5	(0.1)
povray	no	23.9	(0.3)	22.6	(0.2)
astar	no	-0.2	(0.5)	0.1	(0.5)
geomean	yes	13.2		6.4	
geomean	both	12.1		4.6	
ff-sunspider		40.6	(1.4)	11.4	(1.1)
ff-octane		18.6	(5.1)	2.8	(0.9)
ff-drom-js		12.4	(1.5)	4.0	(1.7)
ff-drom-dom		71.2	(1.5)	43.5	(0.6)
geomean		33.9		14.3	

Table 5: Performance overhead on the SPEC CPU2006 C++ benchmarks and Firefox (%), stdev in parentheses.

reported four times our overhead (29.6%) on xalancbmk and 20.0% on soplex while ours is negligible. Povray stands out for having high overhead despite its lack of casts. This is mostly due to the many stack objects allocated in a recursion between the functions `Ray_In_Bounds` and `Intersection`. Other than this special case, overheads are lowered by reducing the number of checks (a la ASAP [24]). The negative overhead for namd may be explained by variations in memory layout [18]. For example, in all but one case, our TypeSan-res configuration can greatly bring down the overhead. The overhead for Firefox is unfortunately somewhat higher, especially for the Dromaeo DOM workload. The high overload is most likely due to the fact that Firefox performs many object allocations, especially on the stack. On average, however, our overhead is close to half of the overhead reported by CaVer (64.6%). The results for the TypeSan-res configuration show that this overhead can be reduced even further by selectively instrumenting casts. Note that our coverage on Firefox is considerably lower than on the other benchmarks mostly due to the use of CMAs (though still much higher than the competition, see Table 8), so extending our solution to cover the remaining casts could increase the overheads reported here.

9.1.3 Memory overhead

Table 6 reports our memory usage on the SPEC CPU2006 C++ benchmarks and Firefox, measured in terms of binary size (static) and the maximum resident set size (dynamic). While TypeSan generally introduces nontrivial memory overhead, we believe this is worthwhile (and acceptable in practice), given the security it offers with negligible performance overhead. Moreover, compared to CaVer [17], our relative run-time memory overhead is much lower for the two SPEC benchmarks they considered (9% vs. 56% on xalancbmk and 1% vs. 150% on soplex). It is unfortunately impossible to compare our memory overhead on Firefox as we measured a different version than the one reported by CaVer. The negative memory overhead for Dromaeo-DOM may be explained by the fact that this benchmark runs for a fixed amount of time, completing fewer runs when the browser is slowed down through our instrumentation. Our memory overhead is mostly due to the metadata storage service it-

	binary size			resident set		
	base	ts	inc%	base	ts	inc%
namd	0.3	0.6	81.6	50.9	56.9	11.7
dealII	3.1	5.2	69.3	818.6	1453.1	77.5
soplex	0.4	0.8	112.7	560.9	568.4	1.3
povray	1.0	1.4	45.2	8.7	18.8	117.4
omnetpp	0.6	1.2	90.4	157.8	224.5	42.3
astar	0.0	0.3	597.1	310.4	314.4	1.3
xalancbmk	4.1	7.6	85.8	451.3	492.7	9.2
geomean			118.0			31.7
ff-sunspider	159.1	318.6	100.2	491.1	928.2	89.0
ff-octane	159.1	318.6	100.2	844.4	1534.4	81.7
ff-drom-js	159.1	318.6	100.2	572.2	1005.8	75.8
ff-drom-dom	159.1	318.6	100.2	4232.0	4015.7	-5.1
geomean			100.2			19.9

Table 6: Memory usage for the SPEC CPU2006 C++ benchmarks and Firefox (MB), ts=TypeSan.

self. As future work, it is possible to share this infrastructure and its memory overhead with other defenses that maintain per-object metadata (e.g., bounds checking), reducing the memory usage of the combined system.

9.2 Coverage

9.2.1 Typecast coverage test suite

To test the correctness of TypeSan and future type-confusion detection systems, we developed a test suite for a wide range of different code constructs that might affect typecast sanitization. The test cases covered by the test suite are inspired by our extensive experience with real-world C++ programs. Our test suite is available as part of the open source repository of TypeSan to help future researchers in testing their systems. The test suite verifies correctness using three different dimensions: *allocation type*, *composition type*, and *cast type*. The test suite allows different configurations from each dimension to be combined and tested simultaneously.

Every test allocates an object of type `AllocationType` using the desired configuration. It then sends a pointer to the allocated object to a function, which derives a pointer to a member of type `BaseType` nested into `AllocationType` with the desired composition type configuration. Finally, we downcast the pointer to a type derived from `BaseType`, called `DerivedType`. We implement the functions in different source files to uncover any potential bugs in interprocedural cross-module static analysis in the checker. Other than false negatives, the test suite also checks for false positives, by replacing the `BaseType` with a derived type of `DerivedType` in `AllocationType`.

Our test suite considers the following allocation types: stack object, stack array, struct argument passed by value, global object, global array, new / new[] (including overloaded operator), and malloc/calloc/realloc (including arrays). For array types, we also consider multidimensional arrays where possible. We consider the following composition types (the relationship of `BaseType` with respect to `AllocationType`): equivalence, nested, nested array element, and inheritance (primary, secondary, and virtual). Finally, we support the following cast types: from primary base to derived type, from secondary base to derived type, and from primary base to a phantom class of the real type.

TypeSan successfully passes all combinations of these configurations, showcasing our coverage on a wide range of

	allocations			casts	types
	heap	stack	global		
dealII	322	5,231	1,125	716	1,238
soplex	40	447	161	2	311
omnetpp	441	85	623	449	568
xalancbmk	414	3,216	2,263	2,688	1,768
namd	10	18	4	0	16
povray	36	255	200	0	257
astar	13	11	7	0	18
firefox	2,458	185,908	42,710	68,369	38,764

Table 7: Instrumentations.

	allocations		casts				
	TypeSan	CaVer	non-null	TypeSan	%	CaVer	%
dealII	597m		3,596m	3,596m	100.00	0	0.00
soplex	21m	1,058	209k	209k	100.00	0	0.00
omnetpp	264m		2,014m	2,014m	100.00	0	0.00
xalancbmk	4,538m	278k	284m	254m	89.52	24k	0.01
ff-sunspider	463m		293m	92m	31.43		
ff-octane	967m		991m	122m	12.35		
ff-drom-js	15,824m		12,976m	3,032m	23.37		
ff-drom-dom	301,540m		46,961m	21,253m	45.26		
ff-total	318,793m	15,530k	61,222m	24,500m	40.02	1,077k	0.00

Table 8: Typecast coverage.

code constructs. For comparison, we also tested CaVer [16] and found that it only passes the following allocation types: global object and new / new[] (including overloaded operator). CaVer also reported false positives when nested arrays were used for the composition. We contacted the authors about all the tests that failed, but we have not received an explanation or a fix for these issues.

9.2.2 *Coverage on benchmarks*

Table 7 shows the number of code locations where we insert instrumentation. Note that the information about allocations is based on the source code because it cannot be easily recognized in the binary due to inlining. The number of cast checks and types are based on the final binary to make them comparable to CaVer [17]. The number of checks in the source code is considerably lower (for example 19,578 for Firefox), presumably due to optimizations that cause code to be duplicated. Compared to CaVer, we insert slightly fewer checks in Firefox, more in Xalanc and the same number in Soplex. This may be due to differences in versions or compiler (settings). We generate more type information than CaVer on Firefox and Soplex, but less on Xalanc. This may be due to differences in representation of type information. In all cases, we insert more instrumentation than UBSan does [17].

Table 8 shows the typecast coverage of TypeSan compared to state-of-the-art solutions, CaVer [17] in particular. Coverage percentages are computed as the fraction of non-null casts correctly checked by the system—missing checks are due to inability to correctly track the corresponding object. As such, it is an indication of the security provided by the system. TypeSan reports over 89% coverage on each of the relevant SPEC benchmarks, and 100.0% on all but one. Unfortunately, coverage is not as high on Firefox. We have found that this is due to the widespread use of pool allocators, which violate the assumption made by our system (as well as other object tracking systems) that objects are allocated individually. This issue could be solved by modifying Firefox to allocate objects directly. This may be viable performance-wise due to the allocation performance offered by tcmalloc. While CaVer does not report per-benchmark results, they report a total of 1,077k verified casts. As such, our coverage is approximately more than 300,000 times as high. For both SPEC and Firefox, Table 8 shows that we track many more allocated objects than CaVer. This explains that we are able to check more casts despite a similar number of instrumentation sites—casts can only be checked if type information metadata was stored at allocation time. As shown in Table 8, we provide security far superior to the current state of the art while improving performance at the same time.

10. RELATED WORK

To the best of our knowledge, UBSan [20] and CaVer [17] are the only other systems that perform verification at cast time like TypeSan. Our system is inspired by CaVer and has considerable similarities with it. In particular, it shares the same benefits with regard to UBSan: we do not rely on runtime type information (RTTI) and therefore we can handle non-polymorphic classes and protect binaries without the need for manually maintained blacklists. Moreover, as we have shown in our evaluation, we introduce less overhead than CaVer, which in turn has shown by its authors to be more efficient than UBSan.

Compared to CaVer, we have a similar instrumentation layer based on an LLVM instrumentation pass, but we have completely redesigned the metadata storage mechanism. In particular, we use a *uniform* variable compression ratio memory shading scheme with off-the-shelf allocation strategies, rather than a purpose built custom memory allocator of the heap and the red-black trees used for the stack in CaVer. Our approach is more efficient for both insertions (object allocations) and lookups (typecast checks) because it does not require identifying the type of the allocation (due to its uniform nature) and it does not incur the significant and non-linear overhead that red-black trees bring to trivial stack allocations. Moreover, our solution is not affected by thread concurrency. This is a major simplification compared to CaVer, which uses per-thread red-black trees.

Another solution that achieves similar goals as ours is preventing calls through incorrect virtual method tables (vtables). For example, Bounov et al. [3] present an approach that can efficiently verify for each virtual call that the vtable is valid for the static type through which the call is performed. This mitigates some type confusion vulnerabilities, but such solutions cannot protect non-polymorphic classes because they do not have vtables. Moreover, this solution only detects type confusion when the object is subjected to a virtual call, thus missing potential memory corruption from a mismatched layout in other parts of the code. Clang CFI [4] uses such a system to check cast operations involving polymorphic classes, but there is no publicly available evaluation of the system and it is still restricted to a subset of downcast operations.

On binaries without source code, Dewey and Giffin [5] show how data flow (reaching definition) analysis may help to determine bounds on vtables and detect type confusion statically by ensuring that a virtual function call does not stray beyond the bounds of the vtable. As noted by the authors, their analysis is prone to false positives and false

negatives and therefore more suited to reducing the number of type confusion bugs prior to deployment.

Finally, CFI [1] and other advanced protection mechanisms for forward edges in C++ programs [21, 23, 22, 14] limit the wiggle room that attackers have to divert control via indirect control transfers. However, as type confusion is mostly a data problem, such solutions only address it partly. Similarly, VTable protection schemes [25, 11], may check the types of virtual calls or the sanity of vtable pointers, but do not prevent the misuse of type confusion in general.

11. CONCLUSION

Type confusion vulnerabilities play an important role in modern exploits as shown in recent attacks against Google Chrome or Mozilla Firefox. Existing solutions that detect type confusion exploits are (i) incomplete, missing a large number of typecasts and (ii) prohibitively slow, thereby hindering general adoption.

We presented TypeSan, an LLVM-based type-confusion detector that leverages an optimized allocator to store metadata in an efficient way to reduce the overhead for updating metadata. Building on several optimizations for both the underlying type checks and the metadata handling, we reduce the performance overhead by a factor 3–6 compared to the state of the art. Our performance figures suggest TypeSan can be used as an always-on solution in practical settings. In addition, TypeSan is complete and no longer misses typecasts on either the stack or between C and C++ object interactions. As we show in the SPEC CPU2006 benchmarks, such interoperability issues between programming languages cause prior work to miss a large number of casts.

12. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. This work was supported, in part, by NSF CNS-1464155 and CNS-1513783, the European Commission through project H2020 ICT-32-2014 "SHARCS" under Grant Agreement No. 64457, and the Netherlands Organisation for Scientific Research through the grant NWO 639.023.309 VICI "Dowsing".

13. REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS*, 2005.

[2] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security*, 2009.

[3] D. Bounov, R. G. Kıcı, and S. Lerner. Protecting c++ dynamic dispatch through vtable interleaving. In *NDSS*, 2016.

[4] Clang. Clang 3.9 documentation - control flow integrity. http://clang.llvm.org/docs/ControlFlowIntegrity.html.

[5] D. Dewey and J. Giffin. Static detection of c++ vtable escape vulnerabilities in binary code. In *NDSS*, 2012.

[6] T. M. Foundation. Dromaeo, javascript performance testing. https://www.webkit.org/perf/sunspider/sunspider.html.

[7] S. Ghemawat and P. Menage. Tcmalloc: Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html, 2009.

[8] E. Goktas, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos. Undermining information hiding (and what to do about it). In *USENIX Security*, 2016.

[9] Google. Octane benchmark. https://code.google.com/p/octane-benchmark.

[10] Google. Sunspider benchmark. https://www.webkit.org/perf/sunspider/sunspider.html.

[11] I. Haller, E. Goktas, E. Athanasopoulos, G. Portokalidis, and H. Bos. Shrinkwrap: Vtable protection without loose ends. In *ACSAC*, 2015.

[12] I. Haller, E. van der Kouwe, C. Giuffrida, and H. Bos. METAlloc: Efficient and comprehensive metadata management for software security hardening. In *EuroSec*, 2016.

[13] J. L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[14] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *OSDI*, 2014.

[15] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–86. IEEE, 2004.

[16] B. Lee, C. Song, T. Kim, and W. Lee. Caver source code. https://github.com/sslab-gatech/caver.

[17] B. Lee, C. Song, T. Kim, and W. Lee. Type casting verification: Stopping an emerging attack vector. In *USENIX Security*, 2015.

[18] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! *ACM Sigplan Notices*, 44(3):265–276, 2009.

[19] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida. Poking holes in information hiding. In *USENIX Security*, 2016.

[20] G. C. Project. Undefined behavior sanitizer. https://www.chromium.org/developers/testing/undefinedbehaviorsanitizer.

[21] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security*, 2014.

[22] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical Context-Sensitive CFI. In *CCS*, 2015.

[23] V. van der Veen, E. Goktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *IEEE S&P*, 2016.

[24] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder. High system-code security with low overhead. In *IEEE S&P*, 2015.

[25] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song. Vtrust: Regaining trust on virtual calls. In *NDSS*, 2016.