CastSan: Efficient Detection of Polymorphic C++ Object Type Confusions with LLVM

Paul Muntean, Sebastian Wuerl, Jens Grossklags, and Claudia Eckert

Technical University of Munich

Abstract. C++ object type confusion vulnerabilities as the result of illegal object casting have been threatening systems' security for decades. While there exist several solutions to address this type of vulnerability, none of them are sufficiently practical for adoption in production scenarios. Most competitive and recent solutions require object type tracking for checking polymorphic object casts, and all have prohibitively high runtime overhead. The main source of overhead is the need to track the object type during runtime for both polymorphic and non-polymorphic object casts. In this paper, we present CastSan, a C++ object type confusion detection tool for polymorphic objects only, which scales efficiently to large and complex code bases as well as to many concurrent threads. To considerably reduce the object type cast checking overhead, we employ a new technique based on constructing the whole virtual table hierarchy during program compile time. Since CastSan does not rely on keeping track of the object type during runtime, the overhead is drastically reduced. Our evaluation results show that complex applications run insignificantly slower when our technique is deployed, thus making CastSan a real-world usage candidate. Finally, we envisage that based on our object type confusion detection technique, which relies on ordered virtual tables (vtables), even non-polymorphic object casts could be precisely handled by constructing *auxiliary* non-polymorphic function table hierarchies for static classes as well.

Keywords: static cast, type confusion, bad casting, type safety, type casting.

1 Introduction

Real-world security-critical applications (*e.g.,* Google's Chrome, Mozilla's Firefox, Apple's Safari, etc.) rely on the C++ language as main implementation language, due to the balance it offers between runtime efficiency, precise handling of low-level memory, and the object-oriented abstractions it provides. Thus, among the object-oriented concepts offered by C++, the ability to use object typecasting in order to increase, or decrease, the object scope of accessible class fields inside the program class hierarchy is a great benefit for programmers. However, as C++ is not a managed programing language, and does not offer object type or memory safety, this can potentially lead to exploits.

C++ object type confusions are the result of misinterpreting the runtime type of an object to be of a different type than the actual type due to unsafe typecasting. This misinterpretation leads to inconsistent reinterpretation of memory in different usage contexts. A typical scenario, where type confusion manifests itself, occurs when an object of a parent class is cast into a descendant class type. This is typically unsafe, if the parent class lacks fields expected by the descendant type object. Thus, the program may interpret the non-existent field or function in the descendant class constructor as data, or as a virtual function pointer in another context. Object type confusion leads to undefined behavior according to the `C++` language draft [1]. Further, undefined behavior can lead to memory corruption, which in turn leads to exploits such as code reuse attacks (CRAs) [6] or even to advanced versions of CRAs including the COOP attack [30]. These attacks violate the control flow integrity (CFI) [2,3] of the program, by bypassing currently available OS-deployed security mechanisms such as DEP [26] and ASLR [28]. In summary, the lack of object type safety and, more broadly, memory safety can lead to object type confusion vulnerabilities (*i.e.,* CVE-2017-3106 [12]). The number of these vulnerabilities has increased considerably in the last years, making exploit based attacks against a large number of deployed systems an everyday possibility.

Checker	**Year**	**Poly**	**Non-poly**	**No blacklist**	**Obj. Tracking**	**Threads**
UBSan [15]	2014	✓				✓
CaVer [22]	2015	✓	✓	✓	✓	limited
Clang CFI [8]	2016	✓		✓		✓
TypeSan [18]	2016	✓	✓	✓	✓	✓
HexType [19]	2017	✓	✓	✓	✓	✓
CastSan	2018	✓	future work	✓	not required	✓

Table 1: High-level feature overview of existing C++ object type confusion checkers.

Table 1 depicts the currently available solutions, which can be used for `C++` object type confusion detection during runtime. The tools come with the following limitations: (1) high runtime overhead (mostly due to the usage of a compiler runtime library), (2) limited type checking coverage, (3) lack of support for non-polymorphic classes, (4) absence of threads support, and (5) high maintenance overhead, as some tools require a manually maintained blacklist.

We consider runtime efficiency and coverage to be most impactful for the usage of such tools. While coverage can be incrementally increased by supporting more object allocators (*e.g.,* child `*obj=dynamic_cast<*child>(parent)`, `ClassA *obj=new (buffer) ClassA();`, `char *str=(char) malloc(sizeof(S)); S *obj=reinterpret_cast<*S>(str);`, see TypeSan, HexType, for more details.) and instrumenting them for later object type runtime tracking, increasing performance is more difficult to achieve due to the required runtime of type tracking support on which most tools rely. Reducing runtime overhead is regarded to be far more difficult to achieve, since object type data has to be tracked at runtime and updating data structures at runtime (*i.e.,* red-black trees, etc.) has to be performed during a type check. As such, due to their perceived high runtime overhead, most of the currently available tools do not qualify as production-ready tools. Furthermore, the per-object metadata tracking mechanisms generally represent an overhead bottleneck in case the to-be hardened program contains: (1) a high volume of object allocations, (2) a large number of memory freeing operations, (3) frequent use of object casts, (4) *exotic* object memory allocators (*i.e.,* Chrom's `tcmalloc()`, object pool allocators, etc.) for which the detection tool implementation has to be constantly maintained.

We present CastSan, a Clang/LLVM compiler-based solution, usable as an always-on sanitizer for detecting all types of polymorphic-only object type confusions during runtime, with comparable coverage to Clang-CFI [8]. CastSan has significantly lower runtime performance overhead than existing tools (see Table 2). Its technique is based on the observation, that virtual tables (vtables) of polymorphic classes can be used as a successful replacement for costly metadata storage and update operations, which similar tools heavily rely on. Our main insight is that: (1) program class hierarchies can be used more effectively to store object type relationships than Clang-CFI's bitsets, and (2) the Clang-CFI bitset checks can be successfully replaced with more efficient virtual pointer based range checks. Based on these observations, the metadata that has to be stored and checked for each object during object casting is reduced to zero. Next, the checks only require constant checking time due to the fact that no additional data structures (*i.e.,* TypeSan and HexType use both red-black trees for storing relationships between object types) have to be consulted during runtime. Finally, this facilitates efficient and scalable runtime vptr-based range checks.

	Programs		
Checker	soplex (C++)	xalancbmk (C++)	astar (C++)
Clang-CFI [8]	5.03%	4.49%	0.9%
CastSan	2.07%	1.78%	0.3%
Speed-Up	2.42 times	2.52 times	3 times

Table 2: Object type confusion detection overhead for SPEC CPU2006 benchmark.

CastSan performs the following steps for preparing the required metadata during compile time. First, the value of an object vptr is modified through internal compiler intrinsics such that it provides object type information at runtime. Second, these modified values are used by CastSan to compute range checks that can validate C++ object casts during runtime. Third, the computed range checks are inserted into the compiled program. The main observation, which makes the concept of vptr based range checks work, is that range checks are based on the fact, that any sub-tree of a class inheritance tree is contained in a continuous chunk of memory, which was previously re-ordered by a pre-order program virtual table hierarchy traversal.

CastSan is implemented on top of the LLVM 3.7 compiler framework [24] and relies on support from LLVM's Gold Plug-in [23]. CastSan is intended to address the problem of high runtime overhead of existing solutions by implementing an explicit type checking mechanism based on LLVM's compiler instrumentation. CastSan's goal is to enforce object type confusion checks during runtime in previously compiled programs. CastSan's object type confusion detection mechanism relies on collecting and storing type information used for performing object type checking during compile time. CastSan achieves this without storing new metadata in memory and by solely relying on virtual pointers (vptrs), that are stored with each polymorphic object.

We evaluated CastSan with the Google Chrome [16] web browser, the open source benchmark suite of TypeSan [18], the open source benchmark programs of IVT [5], and all C++ programs contained in the SPEC CPU2006 [31] benchmark. The evaluation results show that, in contrast to previous work, CastSan has considerably lower runtime overhead while maintaining comparable feature coverage (see Table 1 for more details). The evaluation results confirm that CastSan is precise and can help a programmer find real object type confusions.

In summary, we make the following contributions:

– We develop a novel technique for detection of `C++` object type confusions during runtime, which is based on the linear projection of virtual table hierarchies.
– We implement our technique in a prototype, called CastSan, which is based on the Clang/LLVM compiler framework [24] and the Gold plug-in [23].
– We evaluate CastSan thoroughly and demonstrate that CastSan is more efficient than other state-of-the-art tools.

2 Background

Before presenting the technical details of our approach, we review necessary background information.

2.1 C++ Type Casting

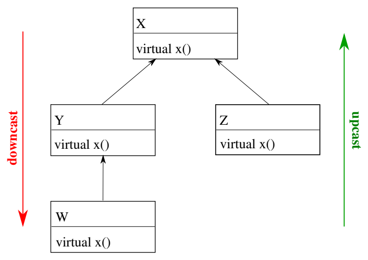
Object type casting in C++ allows an object to be cast to another object, such that the program can use different features of the class hierarchy. Seen from a different angle, object typecasting is a C++ language feature, which augments object-oriented concepts such as inheritance and polymorphism. Inheritance facilitates that one class contained inside the program class hierarchy inherits (gets access) to the functionality of another class that is located above in the class hierarchy. Object casting is different, as it allows for objects to be used in a more general way (*i.e.,* using objects and their siblings, as if they were located higher in the class hierarchy). C++ provides `static`, `dynamic`, `reinterpret` and `const` casts. Note that `reinterpret_cast` can lead to bad casting, when misused and is unchecked "by design", as it allows the programmer to freely handle memory. In this paper, we focus on `static_cast` and `dynamic_cast` (see N4618 [1] working draft), because the misuse of these can result in bad object casting, which can further lead to undefined behavior. This can potentially be exploited to perform, for example, local or remote code reuse attacks on the software.

The terminology of this paper is aligned to the one used by colleagues [18], in order to provide terminology traceability as follows. First, *runtime type* refers to the type of the constructor used to create the object. Second, *source type* is the type of the pointer that is converted. Finally, *target type* is the type of the pointer after the type conversion.

An *upcast* is always permitted if the target type is an ancestor of the source type. These types of casts can be statically verified as safe, as the object source type is always known. Thus, if the source type is a descendant of the target type, the runtime type also has to be a descendant and the cast is legal. On the other hand, a *downcast* cannot be verified during compile time. This verification is hard to achieve, since the compiler cannot know the runtime type of an object, due to intricate data flows (for example, inter-procedural data flows). While it can be assumed that the runtime type is a descendant of the source type, the order of descendancy is not known. As only casts from a lower to a higher (or same) order are allowed, a runtime check is required to check this.

2.2 C/C++ Legal and Illegal Object Type Casts

A type cast in C/C++ is legal only when the destination type is an ancestor of the runtime type of the cast object. This is always true if the destination type is an ancestor of the source type (upcast). In contrast, if the destination type is a descendant of the source type (downcast), the cast could only be legal if the object has been upcast beforehand.



(a) Class hierarchy with four classes.

```
/* downcast */
X *x = new W();
Y *y = static_cast<Y>(x);

/* first upcast*/
Z *z = new Z();
X *x = static_cast<X>(z);

/* second upcast */
Y *y = new W();
X *x = static_cast<X>(y);
```

(b) Downcast and upcast cast in C++.

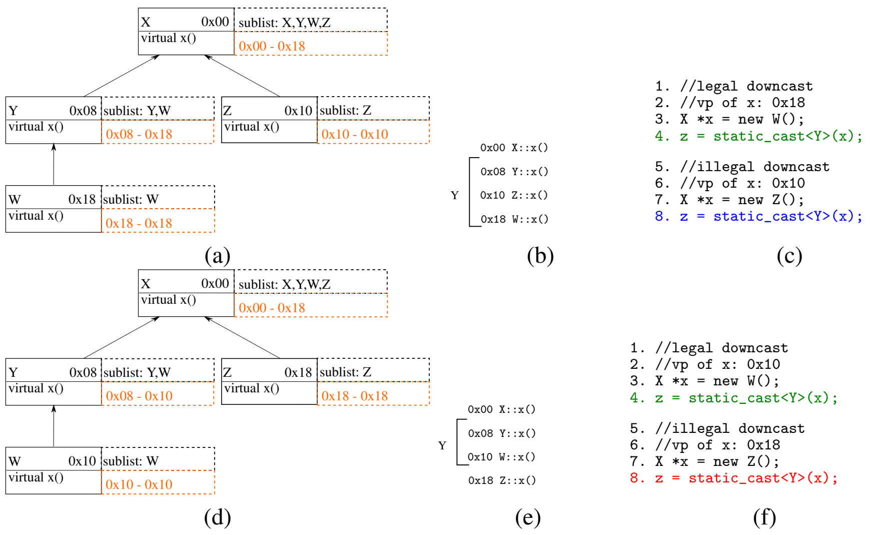
Fig. 1: C++ based object type down-casting and up-casting examples.

Figure 1 depicts upcast and downcast in an example hierarchy. The graph of Figure 1(a) is a simple class hierarchy. The boxes are classes, and the arrows depict inheritance. The code of Figure 1(b) shows how upcast and downcast look in C++. The upcast and downcast arrows besides the graph visualize the same casts that are coded in C++ in Figure 1(a). To verify the cast, the runtime type of the object is needed. Unfortunately, the exact runtime type of an object is not necessarily known to the compiler for each cast, as explained in the previous section. While the source type is known to the compiler for each cast, it can only be used to detect very specific cases of illegal casts (*e.g.,* casts between types that are not related in any way, which means they are not in a descendant-ancestor relationship). All upcasts can be statically verified as safe because the destination type is an ancestor of the runtime type. If the destination type is not an ancestor of the runtime type, then the compiler should throw an error.

2.3 Ordered vs. Unordered Virtual Tables

In this section, we briefly describe the differences between in-memory ordered and unordered vtables and how these can be used to detect object type confusions during runtime.

Figure 2(a), Figure 2(b), and Figure 2(c) highlight the case in which an illegal object cast would not be detected if the vtables are not ordered (see blue shaded code in line number eight), while Figure 2(d), Figure 2(e), and Figure 2(f) show how a legal (see green shaded code in line number four) and an illegal (see red shaded code in line number eight) object cast can be correctly identified by using the object vptr in case the vtables are ordered in memory.



```
1. //legal downcast
2. //vp of x: 0x18
3. X *x = new W();
4. z = static_cast<Y>(x);

5. //illegal downcast
6. //vp of x: 0x10
7. X *x = new Z();
8. z = static_cast<Y>(x);
```

```
1. //legal downcast
2. //vp of x: 0x10
3. X *x = new W();
4. z = static_cast<Y>(x);

5. //illegal downcast
6. //vp of x: 0x18
7. X *x = new Z();
8. z = static_cast<Y>(x);
```

Fig. 2: Illegal and legal object casts vs. ordered and unordered virtual tables.

On the one hand, Figure 2(c) shows the vptr value as it would be present in the unordered case of Figure 2(b) and Figure 2(a). The object x, that is constructed at line number seven with the constructor of Z (runtime type) has a vptr of value `0x18` in the unordered case. x is referenced by a pointer of type X (source type) and at line number eight it is cast to Y (destination type). This is an illegal object cast, as Z does not inherit from Y. The vptr of x is in the range of Y built from the unordered vtable layout of Figure 2(b). A range check would, therefore, falsely conclude that the cast is legal.

On the other hand, Figure 2(f) depicts the same objects as constructed after ordering according to Figure 2(e) and Figure 2(d). At line number three, the object x is instantiated having (runtime) type W. The object, therefore, has a vptr with value `0x10` according to Figure 2(d). The object is referenced by a pointer of type X (source type) and at line number four, the object x is cast to Y (destination type). This cast is a legal object cast, as the vptr `0x10` has a value between the vtable address of Y `0x08` and the address value of the last member of the sub-list of Y `0x10`. Note that this memory range is depicted in Figure 2(e). Further, at line number seven, the object x is newly allocated with the constructor of Z. Next, the object is cast to Y at line number eight. As x's vptr is `0x18`, which is the vtable address of Z, it can be observed that the cast is illegal. The reason is that the vptr value `0x18` is larger than the largest value of the sub-list of Y, which is the vtable address of W, `0x10`. Thus, in this way the object type confusion located at line number eight can be correctly detected.

Finally, note that the range checks, which we will use in our implementation, are precise, when the vtables of all program hierarchies are ordered with no gaps in memory according to, for example, their pre-order traversal. In case this is not guaranteed, then the range checks could generate false positives as well as false negatives (see the blue shaded code in Figure 2(c)).

3 Threat Model

The threat model used by CastSan resembles HexType's threat model. Specifically, we assume a skilled attacker who can exploit any type of object type confusion vulnerability, but who does not have the capability to make arbitrary memory writes. CastSan's instrumentation is part of the executable program code and thus assumed to be write-protected through data execution protection (DEP) or another mechanism. Further, CastSan does not rely on information hiding; as such the attacker is assumed to be able to perform arbitrary reads. This is not a limitation, as CastSan does not rely on randomization or code shuffling as other CFI schemes [10,33]. As CastSan focuses exclusively on `C++` object down-cast type confusions, we assume that other types of memory corruptions (*i.e.,* buffer overflows, etc.) are combated with other types of protection mechanisms and that CastSan can work along these complementary defense mechanisms. Finally, we assume that for any large existing source code base, which is affected by object type confusions (*e.g.,* [11]), this cannot currently be fixed solely by inspecting the source code statically or manually and that the attacker has access to the source code of this vulnerable application.

4 Design and Implementation

In Section 4.1, we present the architecture of CastSan, and in Section 4.2, we explain how virtual table inheritance tree projections are used by CastSan, while in Section 4.3, we describe our object type confusion detection checks. Finally, in Section 4.4, we outline CastSan's implementation.

4.1 Architecture Overview

CastSan's main analysis steps. CastSan instruments object casts as follows: (1) source code files are fed into the Clang compiler, which adds several intrinsics needed to mark all possible cast locations in the code, (2) CastSan uses the vtable metadata and the virtual table hierarchies, which were embedded in each object file in the Clang front-end, (3) placeholder intrinsic-based instructions are used for recuperating the vptr and the mangled name of the object type which will be later cast, and (4) placeholder intrinsic-based instructions for the final pre-cast checks are inserted, containing the per object cast range. The intrinsics will be removed before runtime and will be converted to concrete instruction sequences used to perform the object type cast check. The placeholder intrinsics are used by CastSan since part of the information needed for the checking of illegal casts is not available during compile time (the vptr value is computed during runtime). Finally, during link time optimization (LTO) [25], the following operations are performed: (1) the virtual table hierarchy is constructed and decomposed into primitive vtable trees, and (2) the placeholder intrinsics used to check for down-cast violations are inserted based on the analysis of the previous primitive vtable trees.

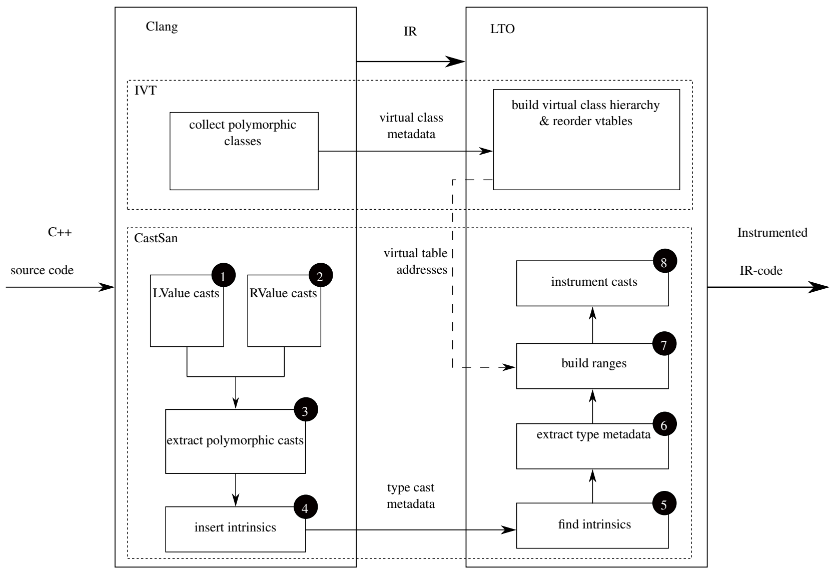


Fig. 3: CASTSAN system architecture.

Figure 3 depicts the placement of CASTSAN's components within the Clang/LLVM compiler framework and the analysis flow indicated by circled numbers.

Building virtual pointer based range checks. First, the `LValue` (LLVM data type) ❶ and `RValue` (LLVM data type) ❷ casts are instrumented inside the Clang compiler with additional `C++` code. Second, only the polymorphic casts are selected from these casts ❸. Third, the polymorphic casts are flagged for instrumentation using an LLVM intrinsic ❹ during LTO. Fourth, the intrinsics inserted by CASTSAN with the help of Clang are detected ❺ for later usage during LTO. Fifth, the metadata of the intrinsics is read out ❻ to acquire all necessary information about an object cast-site. Sixth, the ranges necessary for checking object type confusions are built in ❼. Note that an object range is computed by using the virtual address of the object destination type and the count of all nodes (vtables) inheriting from the destination type. Finally, the object cast-sites are instrumented with a range check ❽.

4.2 Virtual Table Inheritance Tree Projection

CASTSAN computes virtual table inheritance trees for each class hierarchy contained in the analyzed program. Next, CASTSAN uses these vtable inheritance trees to determine if the ancestor-descendant relation between the types of the cast objects holds. The ancestor-descendant relations between object types rely on several properties of these ordered vtable inheritance trees, which we will explain next. The root of such a virtual table inheritance tree is a polymorphic class that does not inherit from other polymorphic classes (root type). Note that a class has only one vtable associated to it. Further, each such vtable is broken into multiple primitive vtables. Also note that these vtables can occupy different places in this ordering. The children of any node in the vtable tree are all types that directly inherit from the ancestor class and are located underneath this class in the program class hierarchy. If a class inherits from multiple vtables, it has a node in any tree that the ancestor types are a part of. The leaves of a vtable tree are vtables, which have no descendants. CastSan will put the vtables that are in any type of a descendant-ancestor relation to each other in a single virtual inheritance tree. Next, we show how a virtual table projection list is computed.

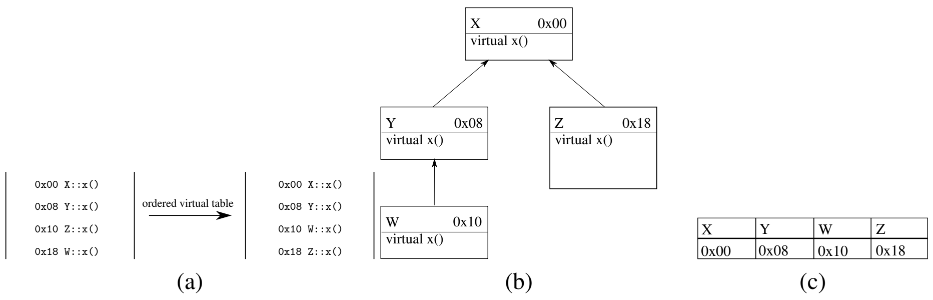


Fig. 4: Unordered and ordered (a) vtables of the tree rooted in X. The tree (b) contains the vptr of each type after ordering. (c) depicts the projected list corresponding to (b).

Figure 4(a) depicts the memory layout of the vtables of the class represented by the primitive hierarchy in Figure 4(b). The vtables contain their addresses as these are laid out in memory (*i.e.,* consider address `0x08`) along with the pointers to the virtual functions (*i.e.,* `Y::x()`). Note that in the unordered table located on the left side of Figure 4(a), there is no relationship between the addresses of the vtables and the class hierarchy. For simplicity reasons, we opted in Figure 4(a) to depict each box of the vtable hierarchy to contain a single entry. In general, when there are multiple entries in each vtable contained in the vtable hierarchy, the vtables will be interleaved to ensure that their base pointers are consecutive addresses in memory. After ordering the values of the addresses of the vtables (right table in Figure 4(a)) the addresses are in ascending order (*e.g.,* `W` inherits from `Y` directly, thus it comes directly after `Y` in the vtable). Further, after interleaving the addresses of the vtables, their values are in ascending order corresponding to the depth-first traversal, as shown in the projected list depicted in Figure 4(c). Next, CastSan uses a pre-order traversal of each vtable inheritance tree in order to construct a list of vtables, which represents a projection of a tree hierarchy onto a list. For example, if the type of a vtable (first row in a box, see Figure 4(b)) is the descendant of another type, it is inserted after the other type in the list. Further, any sub-tree of each tree is represented as a continuous sub-list of virtual tables by CastSan. This means that the types that inherit from the root type of the sub-tree will be inserted into the list in direct succession to the sub-tree root. Finally, the projected list will be used to compute object cast ranges which will subsequently be used to determine legal and illegal relations between the object types during a cast operation.

4.3 Object Type Confusion Detection

Virtual Pointer Usage as Runtime Object Type Identifier. CastSan uses the virtual pointer (vptr) of an object to identify its type at runtime. Note that any polymorphic type contains a set of virtual methods that are reachable from any object using its vptr. The vptr of a type is saved in any polymorphic object that is created using the type's constructor. By type constructor, we mean the function which is called when an object of a certain type is allocated. Furthermore, note that each legally cast instance of a polymorphic object can be uniquely identified by its vptr since the vptr of an object is always the first field of that object. CastSan therefore reads the vptr of any object at runtime to uniquely identify its runtime type. CastSan does this by loading the first 64-bit of the object into a register using an intermediate representation (IR) load instruction. This load instruction is inserted by CastSan during LTO for runtime usage.

Determine Object Type Inheritance at Runtime. As previously mentioned, CastSan checks object casts by using the projected virtual table hierarchy list (see Figure 4(c) for more details). A projected class hierarchy consists of ordered vtable addresses. The runtime type of an object must inherit from the destination type of the cast in order for the cast to be legal. This happens if the vtable of the runtime type is a child in the sub-tree of the vtable of the destination type. Further, if this is the case, the runtime type comes after the destination type in the depth-first list of the tree. Since all nodes of a sub-tree are placed successively in the projected list, this means that these nodes are located before the last element of the sub-tree in the list. Therefore, CastSan does not need to traverse the whole sub-list representing the sub-tree of the destination type to check if the runtime type is part of it. It is enough to check whether it is anywhere between the first and the last element in the list. This holds because the type of the object holding the vptr has to have a vtable in the sub-tree of the destination type, which means it inherits from the destination type. Otherwise, if the vptr is not in the range, it has no vtable inheriting from the vtable of the destination type and therefore its type does not inherit from the destination type. Therefore, the object cast is illegal in this situation. CastSan implements this mechanism at runtime using range checks on the vtable pointer of an object and additionally by using the values of the vtable addresses of the destination type sub-tree. CastSan checks during runtime if the value of the vptr is larger than the vtable address of the destination type and smaller than the address value of the last vtable entry located in the sub-list corresponding to the destination type. If this holds, then the runtime type must inherit from the destination type; therefore, the cast is legal. Otherwise, if the vptr value is not contained between the above mentioned boundaries, then the runtime type does not inherit from the destination type, thus the object cast is not legal.

Virtual Table based Range Checks. CastSan uses vtable based range checks in order to check if the vptr of an object resides between two allowed values. CastSan's range check is based on the observation that the addresses of the ordered vtables are re-arranged by interleaving them through a pre-order traversal of the inheritance trees in which these vtables are contained. Therefore, the addresses of any sub-tree lay continuously and gapless in memory. By continuously and gapless we mean that there is no starting address of another vtable not belonging to the sub-tree in between the addresses of a sub-tree, and the starting addresses of the vtable lie consecutively in memory, respectively. Further, if the vptr points to any address between the first and the last address of the sub-tree, then it has to be in the list of all addresses located in the sub-tree and therefore the cast is legal. In this way, CastSan can simplify the type check to a range check. CastSan builds a range check by using the vtable address V of the destination type X and the count c of all classes that inherit from X. V and c can be statically determined at compile time for each object cast performed in the program. To perform the check at runtime, the vptr value P is extracted from the object before the cast. Next, the following expression is evaluated by CastSan during runtime. *If* $V + c \geq P \geq V$ holds, then the cast is legal, otherwise the cast is illegal and program execution will be terminated or an error log output can be produced depending on the employed CastSan usage mode flag. Note that CastSan offers the possibility to include in the *else*-branch of the inserted cast check the option to log back-trace information instead of terminating the program which is obviously not always desired (see Figure 5 for more details).

The generated object cast range check has the following advantages compared to other state-of-the-art techniques. First, in terms of memory overhead, CastSan does not require any additional metadata at runtime to be recorded, deleted or updated in order to determine class hierarchy relationships. Second, the range check needed for the sub-typing check has $O(1)$ runtime cost compared to $O(n)$ runtime cost of other tools due to traversals of additional data structures (*e.g.,* red-black tree).

Instrumenting a C++ Object Cast. CastSan replaces the cast check intrinsics inserted into the code within the Clang compiler with a range based cast check (see ❸ depicted in Figure 3 for more details) during LTO. The check is substituted with an equality check if the count of vtables in the range is one. The equality check matches the vtable address of the range with the vptr of the object. If the addresses are equal, then the cast is legal, otherwise it is illegal. In case the range has more elements than one, then a range check will be inserted. The steps for building and inserting the final range check are as follows. First, the value of the start address of the range is subtracted from the vptr value by CastSan. Further, if the pointer value was lower than the start address of the vtable, then the result is negative and the cast is illegal. Second, the result of the subtraction is next rotated by three bits to the right to remove the empty bits that define the pointer length. If the result of the subtraction was negative, this rotation shifts the sign of the result to the right, making it the most significant bit. Therefore, if the cast is illegal, then the result of the bit rotation is a large number. More specifically, the number is then larger than any result of a valid cast. This holds because the most significant bit, where the sign was shifted due to the rotation, would have been shifted to the right. This would make the number smaller than the illegal case. The result is either the distance of the destination type from the runtime type within the vtable hierarchy or an invalid large number. Finally, the value is compared to the number of vtables in the range. If the value is less than or equal to the count, then the cast is legal and program execution can continue, otherwise an illegal cast is reported. By using these instructions, the range check can ensure three preconditions for a legal cast using only one branch. If any of the following preconditions do not hold, CastSan will report an illegal cast. This is the case if the value of the vptr is: (1) higher than the last address in the range (*i.e.,* the type of the object is not directly related to the destination type), (2) lower than the first value of the vtable address range (*i.e.,* the runtime type of the object is an ancestor of the destination type), resulting in the negative bit being shifted to a significant bit of the subtraction result, or (3) not aligned to the pointer length (*i.e.,* the pointer is corrupted). Note that in (3) the unaligned bit is rotated to one of the significant bits or to the signing bit. Since the comparison is unsigned, the number would then again be larger than the last address in the vtable range.

Further, note that the vptr of an object can always be used to perform the check in the primary inheritance tree of the object source type. Finally, the primary inheritance tree, represents the tree which contains the virtual table of the object types as primary parent.

```
1 X *x=new W();
2 Y *y=static_cast<Y>(x);
3 y->x();
```

(a)

```
1 0x400fe0 mov   %r15,%rdi
2 0x400fe3 callq *(%rax)
```

(b)

```
1 0x400fc0 ud2
2 0x400fcb mov   $0x401080,%ecx
3 0x400fcf mov   %rax,%rdx
4 0x400fd1 sub   %rcx,%rdx
5 0x400fd6 rol   $0x3d,%rdx
6 0x400fda cmp   $0x2
7 0x400fde ja    0x400fc0
8 0x400fe0 mov   %r15,%rdi
9 0x400fe3 callq *(%rax)
```

(c)

Fig. 5: Instrumented polymorphic C++ object type cast.

Figure 5 depicts a C++ object type cast at line number two in Figure 5(a), the uninstrumented assembly code in Figure 5(b), and the assembly code instrumentation added by CastSan in Figure 5(c) (the range check is highlighted in gray shaded color). In Figure 5(a), without line number three the compiler generates does not generate code since the Clang/LLVM compiler is designed to not generate specific code for object casts. Only for the object dispatch (see line number three), assembly code is generated. The assembly code in Figure 5(b) corresponds to the object dispatch depicted in Figure 5(a) at line number 3. Finally, we assume that the OS provides an $W \oplus X$ protection mechanism (*e.g.,* data execution prevention (DEP)) and thus the assembly code depicted in Figure 5(c) cannot be modified (rewritten) by an attacker.

Next, we present the operations performed by the instructions contained in the range check (gray shaded code in Figure 5) in order to better understand how the check operates. First, the vtable address of type X (corresponding to line number one in Figure 5(a)) `0x401080` is loaded. In line number two, in Figure 5(c), the fixed value of the address is moved to the register `%rcx`. This is done in order to load the first value of the range. Second, the vptr of the object x is moved to register `%rdx` depicted in line number three. This is done in order to provide the second value of the subtraction of the range check. Note that the object pointer itself was already loaded in register `%rax`. This is not depicted in Figure 5 for reasons of brevity. Third, the `sub` instruction performs the subtraction of the vtable address (stored in `%rcx`) from the vptr (stored in `%rdx`). At line number five, depicted in Figure 5(c), the pointer alignment is removed from the result by using a rotation (*i.e.,* `rol`) instruction. This is done to obtain the distance of the vptr from the vtable address of the destination type located in the vtable hierarchy. Note that if the number of all types inheriting from the destination type is higher or equal to the distance, the cast is legal. Finally, the result is compared to the constant `$0x2`, which is the number of all types inheriting from the destination type Y, specifically these are Y and W. Then, the program execution either jumps to the address of the instruction `ud2` located at line number one in Figure 5(c) (address `0x400fc0`), which terminates the program; otherwise, the object dispatch (line number three in Figure 5(c)) will be performed similar as in Figure 5(b) and the program continues its execution.

4.4 Implementation

Components. CastSan is implemented as two module passes for the Clang/LLVM compiler [24] infrastructure by extending LLVM (v.3.7) and relies on the Gold plug-in [23]. CastSan is based on the virtual table interleaving algorithm presented by Bounov *et al.* [5] from which it reuses its interleaved vtable metadata, by transporting it from the Clang compiler front-end to the LTO phase via new metadata nodes inserted into LLVM's IR code. More specifically, CastSan's implementation is split between the Clang compiler front-end, and a new link-time pass used for analysis and generating the final intrinsic based compiler cast checks. CastSan's transformations operate on LLVM's intermediate representation (IR), which retains sufficient programming language semantic information at link time to perform whole program analysis and identify all possible types of polymorphic C++ casts in order to instrument them.

Usage Modes. CastSan's implementation provides three operation modes with corresponding compiler flags. First, *attack prevention mode* can be used in shipped program binaries to customers. This mode can be used, if desired, to terminate program execution when an illegal cast is detected, thus providing an effective mechanism for avoiding undefined behavior which may lead to vulnerability based CRAs. Second, *software testing mode* can be used during program testing in order to detect type confusion errors and to help fix them before the software is shipped by subjecting the analyzed program to a test suite with different possible goals (*i.e.,* program path coverage, etc.). Finally, *relaxed mode* can be used to detect and log illegal casts detected during development or deployment. This last mode is mainly intended as a replacement for the situation that it is not safe to stop program execution which is mainly the case for real-world programs.

5 Evaluation

We evaluated CastSan by instrumenting various open source programs and conducting a thorough analysis with the goal to show its effectiveness and practicality. The experiments were performed using the open source benchmarks TypeSan [18], IVT [5], Google's Chrome (v.33.0.1750.112) web browser, and SPEC CPU2006 benchmark (only for the C++ based programs), which were also used by HexType [19]. If not otherwise stated, we used the Clang `-O2` compiler flag for all our experiments. In our evaluation, we addressed the following research questions (RQs).

RQ1: What is the **runtime overhead** of CastSan (Section 5.1)

RQ2: How **precise** is CASTSAN? (Section 5.2)
RQ3: How **effective** is CASTSAN? (Section 5.3)
RQ4: How can CASTSAN **assist a programmer** during a bug bounty? (Section 5.4)

Comparison Method. In addition to the runtime overhead and binary blow-up, the coverage and precision of HexType is compared to that of CASTSAN. For benchmarking SPEC CPU2006, the benchmark script of TypeSan, and the micro-benchmark of ShrinkWrap [17] was used.

Preliminaries. The script of TypeSan (approx. 606 Bash LOC) sets up a full environment consisting of: Binutils, Bash, Coreutils, CMake, Pearl. These are used for instrumenting the SPEC CPU2006, and UBench (consisting of 10 intricate `C++` test-cases). After the benchmark is set up, the script compiles the programs and checks each program by starting it and checking it to see if it executed successfully.

The script of IVT (approx. 200 Python LOC) is used to compile up to 50 `C++` programs. Some of the programs contain object type confusions. After each instrumented program execution, the script checks if the program executed successfully or not.

Experimental Setup. We evaluated CASTSAN on an AMD Ryzen R7 1800x CPU using 8 cores with 16 GB of RAM running the Debian 8 Jessie OS. All benchmarks were executed 10 times to obtain reliable mean values.

5.1 Performance Overhead (RQ1)

Benchmark	Vanilla	CastSan	Overhead
soplex	207.14	211.43	2.07%
povray	123.34	125.28	1.57%
omnetpp	269.14	270.06	0.34%
astar	334.96	335.96	0.30%
dealII	186.71	188.47	0.94%
xalanckbmk	413.67	421.03	1.78%
namd	266.42	266.43	0.00%
average			1.0%
geomean			0.92%

Table 3: Benchmark results of running various C++ programs contained in the SPEC CPU2006 benchmark with CASTSAN enabled and disabled (vanilla). The values represent the mean time needed to finish running the benchmark program over 10 runs.

Table 3 depicts the overall runtime overhead on only the relevant `C++` programs contained in the SPEC CPU2006 benchmark. The geomean value of the overhead in these benchmarks is under 1% (0.92%). As an outlier, `soplex` showed an overhead of 2.07%. For most benchmarks, the overhead is lower than 1.0%. Some SPEC CPU2006 benchmarks like `astar` do not contain static casts and thus no check is performed. These results show that the overhead is within the margin of error. This is to be expected as CASTSAN does not need to execute additional code on execution when no checkable casts are present in the code.

Benchmark	High/Low	Vanilla	CastSan	Overhead
gc-sunspider [32]	$<$	123.4	124.1	0.57%
gc-octane [27]	$>$	29885	29889	-0.01%
gc-drom-js [14]	$>$	1987.21	1991.58	-2.18%
gc-balls [4]	$>$	216	215	0.47%
gc-kraken [21]	$<$	933.1	941.2	0.87%
gc-jetstream [20]	$<$	184.06	184.44	0.21%
average				-0.01%
geomean				0.31%

Table 4: Runtime overhead on Chrome with CastSan enabled and disabled (vanilla).

Table 4 depicts the average and geomean runtime overheads of CastSan in seven of the most popular JavaScript benchmarks. The greater/less symbols (in High/Low) next to the name describe if higher ($>$) or lower ($<$) values are better in the benchmark. More precisely, higher is better for jetstream, octane, balls and dromaeo benchmarks; lower is better in sunspider and kraken. The numbers in columns Vanilla and CastSan represent aggregate benchmark scores and have no particular intrinsic meaning. The average value of the overhead of CastSan in these benchmarks is -0.01%, which is in the margin of error. The low overhead obtained when running JavaScript benchmarks in the instrumented Chrome demonstrates that CastSan can efficiently scale to large code bases with complex class hierarchies.

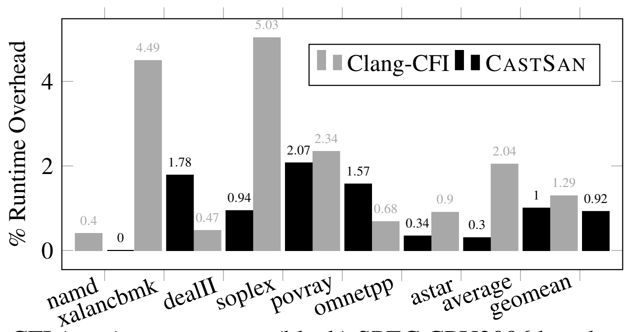


Fig. 6: Clang-CFI (gray) vs. CastSan (black) SPEC CPU2006 benchmark overhead.

Figure 6 depicts the average and geomean runtime overheads of CastSan in comparison with the Clang-CFI cast checker when ran on several C++ programs contained in the SPEC CPU2006 benchmark with the following compiler flags: `-fsanitize=cfi-cast-strict`, `-fsanitize=cfi-derived-cast`, and `-fsanitize= cfi-unrelated-cast`. Note that the Clang-CFI cast checker instruments the same set of static object casts as CastSan. We compared the Clang-CFI and CastSan runtime overhead w.r.t. the baseline LLVM 3.7 compilations. Note that for the baseline compilation no additional compiler flags and no LTO support (we compiled without the Clang's `-flto` compiler flag) was used. Finally, it can be observed that the overhead of CastSan is about two times lower on average than the overhead of Clang-CFI when running on the SPEC CPU2006 programs.

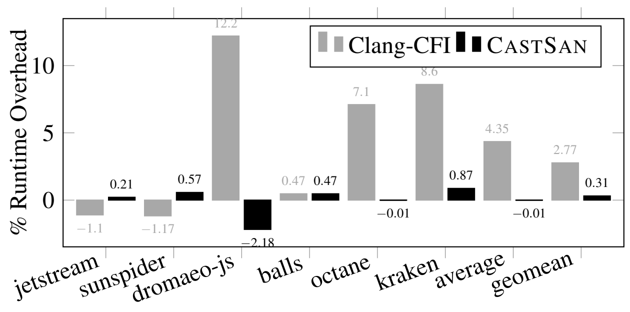


Fig. 7: Clang-CFI (gray) vs. CastSan (black) Chrome runtime overhead.

Figure 7 depicts the runtime overhead of Chrome when ran on several JavaScript benchmarks. First, we compiled with Clang-CFI, and second, with CastSan enabled and with the following compiler flags enabled: `-fsanitize=cfi-cast-strict`, and `-fsanitize=cfi-derived-cast`. We did not use the `-fsanitize=cfi-unrelated -cast` compiler flag, since Chrome was not able to start (crashed during start) after applying this flag. In total, the same amount of object casts where instrumented by each of the tools. However, we can observe that compared to Clang-CFI, the geomean and average overheads of CastSan are better on large code bases such as the Chrome browser. The lowest runtime overhead value, -2.18%, was obtained with CastSan when running the Dromaeo-js benchmark, while the lowest overhead, -1.17%, was obtained by Clang-CFI when running the Sunspider JavaScript benchmark. Overall, we observed a 54 times speed-up on average and 8.9 times speed-up in geomean for CastSan when compared to Clang-CFI cast checker.

5.2 Precision (RQ2)

We evaluated the precision of CastSan by using complex class hierarchies of programs contained in the open-source micro-benchmark of TypeSan [18] and the benchmark programs (in total more than 50 programs) provided by the IVT tool. This benchmark includes: (1) casts to secondary parents, (2) casts within a diamond inheritance, and (3) casts from unrelated trees.

The results indicate that each cast that is covered by CastSan can be precisely checked and the implementation leaves no room for unmitigated corner cases. Moreover, CastSan did not show the imprecisions described in the ShrinkWrap paper. There, the authors show specific cases of class inheritances (*e.g.,* diamond inheritance) where vtable based function call sanitizers allow calls to illegitimate functions of sibling classes. Finally, CastSan was able to cope with all complex class hierarchies contained in these benchmarks and no false negatives or false positives were reported. Thus, we conclude that CastSan is precise and leaves no space for untreated corner cases.

5.3 Effectiveness (RQ3)

We evaluated the effectiveness of CastSan by selecting the last ten type confusions reported in Google Chrome which had common weakness enumeration (CVE) reports associated. All these type confusions have been reported and partially fixed in the current Chrome browser version. The goal of this experiment is to show that CastSan can find object type confusions in real-world software.

We recompiled the Chrome web browser with the CastSan checks in place and ran all JavaScript benchmarks, which we also used to check the performance of Chrome (see Figure 7 for more details). In total, out of the ten object type confusions, CastSan was able to report three type confusions at the correct location. We further investigated the other undetected type confusions and found out that these were not detected since the used JS benchmarks do not interact with the code of Chrome which contains these bugs. As such, this is an issue which can be addressed with more extensive test suites which reach the other bugs not previously detected. Finally, we conclude that CastSan is effective in detecting real-world type confusions.

5.4 Programmer Assistance (RQ4)

We evaluated how useful CastSan is in helping a programmer to find and fix a type confusion bug. For this reason, we used a well-known type confusion bug and depict the error log in order to show how the programmer is guided when fixing a type confusion bug. The goal of this experiment is to show that CastSan can effectively help a programmer to pinpoint the exact bug location. Figure 8 depicts the backtrace that CastSan prints out when running the `xalancbmk` program contained in the SPEC CPU2006 benchmark. The SPEC CPU2006 `xalancbmk` has a known type confusion vulnerability, as mentioned in [5], which CastSan is able to detect. Thus, on execution, it prints the back-trace leading to the illegal cast. Line numbers 1 to 27 are the verbose output of CastSan, notifying the user that an illegal cast happened during execution. In lines 25, 26 and 27 the mangled name of the exact function containing the illegal object cast is printed. Using the offset printed in the square brackets at the end of the line, a developer can find the

```
 1 ./Illegal Cast Detected. Printing Backtrace:
 2 ./Xalan_base[0x5a3de8]
 3 ./Xalan_base(_ZNK11xercesc_2_511DOMTextImpl13getParentNodeEv+0x6)[0x5cb1c6]
 4 ./Xalan_base(_ZN11xercesc_2_513DOMParentNode12insertBeforeEPNS_7DOMNodeES2_+0x25a)[0x5bae2a]
 5 ./Xalan_base(_ZN11xercesc_2_511DOMAttrImpl8setValueEPKt+0xb0)[0x59e6a0]
 6 ./Xalan_base(_ZN11xercesc_2_512XSDDOMParser12startElementERKNS_14XMLElementDeclEjPKtRKNS_
 7   11RefVectorOfINS_7XMLAttrEEEjbb+0x520)[0x6e15e0]
 8 ./Xalan_base(_ZN11xercesc_2_512IGXMLScanner14scanStartTagNSERb+0x1a0f)[0x61134f]
 9 ./Xalan_base(_ZN11xercesc_2_512IGXMLScanner11scanContentEv+0x171)[0x60e451]
10 ./Xalan_base(_ZN11xercesc_2_512IGXMLScanner12scanDocumentERKNS_11InputSourceE+0x67)[0x60e0c7]
11 ./Xalan_base(_ZN11xercesc_2_517AbstractDOMParser5parseERKNS_11InputSourceE+0x22)[0x5779e2]
12 ./Xalan_base(_ZN11xercesc_2_512IGXMLScanner20resolveSchemaGrammarEPKtS2_+0x685)[0x61b8c5]
13 ./Xalan_base(_ZN11xercesc_2_512IGXMLScanner19parseSchemaLocationEPKt+0xe0)[0x61b1a0]
14 ./Xalan_base(_ZN11xercesc_2_512IGXMLScanner28scanRawAttrListforNameSpacesEi+0x4b7)[0x61ad47]
15 ./Xalan_base(_ZN11xercesc_2_512IGXMLScanner14scanStartTagNSERb+0x3b2)[0x60fcf2]
16 ./Xalan_base(_ZN11xercesc_2_512IGXMLScanner11scanContentEv+0x171)[0x60e451]
17 ./Xalan_base(_ZN11xercesc_2_512IGXMLScanner12scanDocumentERKNS_11InputSourceE+0x67)[0x60e0c7]
18 ./Xalan_base(_ZN11xercesc_2_517SAX2XMLReaderImpl5parseERKNS_11InputSourceE+0x25)[0x6537f5]
19 ./Xalan_base(_ZN10xalanc_1_828XalanSourceTreeParserLiaison14parseXMLStreamERKN11xercesc_2_
20   511InputSourceERKNS_14XalanDOMStringE+0x120)[0x7c8070]
21 ./Xalan_base(_ZN10xalanc_1_824XalanDefaultParsedSourceC1ERKN11xercesc_2_511InputSourceEbPNS1_
22   12ErrorHandlerEPNS1_14EntityResolverEPKtSA_+0x1da)[0x7cb8ea]
23 ./Xalan_base(_ZN10xalanc_1_816XalanTransformer11parseSourceERKNS_15XSLTInputSourceERPKNS_
24   17XalanParsedSourceEb+0x303)[0x7cc983]
25 ./Xalan_base(_ZN10xalanc_1_816XalanTransformer9transformERKNS_15XSLTInputSourceES3_RKNS_
26   16XSLTResultTargetE+0x2a)[0x7ccaea]
27 ./Illegal instruction
```

Fig. 8: Type confusion back-trace for the `xalancbmk` program.

line in the code where the illegal object cast was defined. The error log depicted in Figure 8 demonstrates that CastSan is able to detect real type confusion bugs in applications by running a program in backtrace-mode. Finally, we conclude that CastSan can help developers during bug bounties [34], and can protect against exploitable type confusions.

6 Discussion

In this section, we present CastSan's limitations and discuss how to address these.

Non-Polymorphic Classes. CastSan provides type safety for objects stemming from polymorphic classes and low runtime overhead. Further, CastSan cannot check casts between non-polymorphic objects. This is because only polymorphic objects have a virtual pointer (vptr). The vptr is an integral requirement for checking object type casts using CastSan. This means CastSan cannot mitigate all types of object type confusion vulnerabilities. A possible way to address this limitation is to construct for static classes an artificial virtual-table-like metadata on which CastSan's technique can be based such that our technique becomes usable for non-polymorphic object type casts.

Reinterpret-Cast. In C++, not only `static_cast` can lead to object type confusion. The misusage of `reinterpret_cast` can also pose threats. HexType addresses this threat by extending its type cast checking to `reinterpret_cast` in addition to `static_cast`. While this can effectively hinder a type confusion vulnerability from occurring, it is debatable if checking `reinterpret_cast` is viable. This question arises, as `reinterpret_cast` can be used as a legitimate way of breaking class hierarchy boundaries, if the memory layout of the cast types match. In this case, a type cast check based on class hierarchy information cannot be made. Therefore, if `reinterpret_cast` is checked for type safety, its purpose can potentially be circumvented. Similarly, as other object type confusion detection tools handle `reinterpret_cast`, we could use compiler runtime checking support for checking for this type of confusions.

Increasing Tool Coverage. The incremental research work between TypeSan and HexType shows that the main path for increasing object type confusion detection coverage is to support more types of memory allocators (*i.e.,* jmalloc, tcmalloc, etc.) or other more exotic ones. Further, the coverage of CastSan can be increased by supporting all types of `C++` program locations (*i.e.,* statement types) where such vulnerabilities could manifest. Thus, CastSan's coverage can be consistently increased by instrumenting all these source code locations with the needed checks in place in order to check during runtime for object type confusions.

Finding New Vulnerabilities. Finding new object type confusion vulnerabilities is directly linked to increasing the tool coverage and is mainly driven by three lines of research. These are: (1) check new program locations which were previously not possible to be instrumented, (2) support new memory allocators (*e.g.,* object pool allocators, etc.), and (3) reduce the runtime overhead of an object type detection technique such that the technique becomes applicable in real-world deployment. Thus, in future work we want to increase the coverage of CastSan by addressing the above mentioned points.

7 Related Work

Virtual Table Pointer-based Tools. Clang-CFI [7,9] (cast checker) is similar to CastSan in that it uses no runtime library and all cast check detection metadata is computed during compile time. However, there are no publicly available evaluation results of Clang-CFI, and therefore we evaluated Clang-CFI in Section 5 independently. Clang-CFI relies on bitsets in order to model the class hierarchy of a program. Clang-CFI uses these bitsets to encode the valid virtual table start addresses for each class. Compared to CastSan, Clang-CFI has a higher runtime overhead, as the bit-set checking technique on which it relies apparently is less efficient than our virtual table based technique.

C++ Object Type Runtime Tracking. All currently available polymorphic and non-polymorphic object type confusion detection tools (except Clang-CFI) rely on dynamic checks (*i.e.,* LLVM's Compiler-RT is mostly used) for several key reasons, as follows. First, the object type has to be tracked during runtime. Second, this is due to the limited precision of static analysis techniques, which cannot recuperate the object type or a set of possible types before program runtime, Third, the object type confusions manifest only during runtime. Finally, object type confusions are hard to replicate statically (*i.e.,* compile time or through symbolic execution, without running the program).

However, the most significant reason is the fact that the types of casted objects, referenced by pointers, may be program input dependent and thus only precisely obtainable during runtime. On the one hand, in the best case the allocation of the object being cast can be tracked during compile time (*e.g.,* if the runtime path from allocation to cast is linear). On the other hand, in the worst case the object type cannot be approximated (*e.g.,* the object was given via a void-pointer from an external function previously).

Compiler-based Tools. UBsan [15], CaVer [22], TypeSan [18], and HexType [19] are compiler based tools that perform object type confusion detection at runtime for C++ based programs. Since HexType is the successor of TypeSan, the tools are very similar to each other from a technical perspective. These two tools and CaVer rely on a runtime metadata service and can reach a high coverage while imposing a considerable performance overhead. CastSan, on the other hand, uses metadata that is statically created at compile-time and can therefore apply very performant checks at runtime. CastSan can protect against polymorphic casts by using vtable hierarchy based ranges and without using a black list. Compared to TypeSan, CastSan partially shares the instrumentation layer, which is unavoidable, but it uses completely different metadata without storing data at runtime. More precisely, CastSan uses the vtables of polymorphic classes. These tables, that need to be in memory at runtime anyways, already provide a view on the class hierarchy. That is enough for CastSan to perform runtime checks without relying on further metadata as maintained by HexType. HexType, on the other hand, reaches a higher coverage, as it can check non-polymorphic objects as well. CastSan is more runtime-efficient than CaVer and HexType, which both require a red-black tree to be traversed (only for the slow path) during each check.

Binary-based Tools. Dewey *et al.* [13] were able to recuperate vtables from program binaries and detect object type confusions indirectly by checking the bounds of a virtual function call. This was achieved by enforcing a policy to check if the vptr lies inside some legitimate bounds. As suggested by the authors, their analysis is imprecise because for example—as also demonstrated by Prakash *et al.* [29]—determining the end of a vtable in binaries without RTTI information is not trivial. Thus, false positives and false negatives are raised, and as such this type of tool is in the best case usable before system deployment.

8 Conclusion and Future Work

C++ object type casting confusions have an important role in modern exploits as demonstrated by recent attacks against Mozilla's Firefox and Google's Chrome web browsers.

In this paper, we presented CastSan, a new polymorphic only object type confusion detection tool. CastSan's novel technique is based on an efficient and time constant virtual pointer range check which is possible by extracting virtual table inheritance trees out of a previously constructed virtual table inheritance hierarchy. CastSan constructs linear projections out of virtual table inheritance trees, which are subsequently used do build runtime object cast checks. Our evaluation results show that CastSan is more efficient than state-of-the-art tools (*i.e.,* Clang-CFI cast checker), and has comparable checking coverage with other state-of-the-art tools, which—in contrast—rely on runtime intensive type tracking for checking type confusions for both polymorphic and non-polymorphic objects.

In future work, we want to use our static metadata based technique to extended existing purely runtime based object type confusion detection tools such as TypeSan and HexType. These tools use for both polymorphic and non-polymorphic object type checking a runtime library which adds considerable runtime overhead due to updates, search, and deletion of object type meta-data data. We think that our approach can be used to avoid the tracking of metadata for polymorphic objects. Further, a complementary artificial virtual table like metadata class hierarchy can be built for non-polymorphic objects as well. Finally, in this way our technique becomes usable also in this context, thus avoiding or considerable reducing the overhead introduced by the runtime compiler checking support.

Acknowledgement

We thank Mathias Payer from Ecole Polytechnique Fédérale de Lausanne (EPFL) for fruitful discussions which helped to improve the quality of this paper. We thank Dimitar Bounov from the University of California, San Diego, and Benjamin Johnson from the Technical University of Munich, for reviewing an early version of this paper. Jens Grossklags' research is supported by DIVSI. Further, we want to thank the anonymous reviewers for their valuable feedback.

References

1. "2016 Working Draft, Standard for Programming Language C++ N4618", `https://goo.gl/PPJ5QC`
2. Martin Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti, "Control Flow Integrity", In *CCS*, 2005.

3. Martin Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti, "Control Flow Integrity Principles, Implementations, and Applications", In *TISSEC*, 2009.
4. Balls Browser Benchmark, http://bubblemark.com/, 2017.
5. Dimitar Bounov, Rami Gökhan Kici, and Sorin Lerner, "Protecting C++ Dynamic Dispatch Through VTable Interleaving", In *NDSS*, 2016.
6. Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage, "When Good Instructions Go Bad: Generalizing Return-oriented Programming to RISC", In *CCS*, 2008.
7. Clang. "Clang 3.9 Documentation - Control Flow Integrity", https://goo.gl/gnmoHU
8. Clang. "Clang 5 Documentation - Control Flow Integrity", https://goo.gl/bW4DyS, 2017.
9. Clang-CFI Cast Checker Metadata, https://goo.gl/JkGDjL.
10. Stephen Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz, "It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks", In *CCS*, 2015.
11. "CVE-2016-1612: Bug Description and reward", https://goo.gl/9SxjEA, 2016.
12. "CVE-2017-3106: Object Type Confusion in Adobe F. Player v. 26.0.0.137", https://goo.gl/gakD25, 2017.
13. David Dewey and Jonathon Giffin, "Static Detection of C++ VTable Escape Vulnerabilities in Binary Code", In *NDSS*, 2012.
14. Dromaeo Browser Benchmark, http://dromaeo.com/?v8, 2017.
15. Google, "Undefined Behavior Sanitizer", https://goo.gl/ELrNKj, 2017.
16. Google, "The Chromium Projects, Chromium", https://goo.gl/uE486n, 2017.
17. Istvan Haller, Enes Goktas, Elias Athanasopoulos, Georgios Portokalidis, and Herbert Bos, "ShrinkWrap: VTable Protection Without Loose Ends", In *Annual Computer Security Applications Conference (ACSAC)*, 2015.
18. Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, and Cristiano Giuffrida, "TypeSan: Practical Type Confusion Detection", In *CCS*, 2016.
19. Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer, "HexType: Efficient Detection of Type Confusion Errors for C++", In *CCS*, 2017.
20. JetStream Browser Benchmark, http://browserbench.org/JetStream/, 2017.
21. Kraken JavaScript Benchmark, https://krakenbenchmark.mozilla.org/, 2017.
22. Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee, "Type Casting Verification: Stopping an Emerging Attack Vector", In *USENIX Security*, 2015.
23. LLVM, "The LLVM Gold Plugin", https://goo.gl/UjFxih, 2017.
24. LLVM, "LLVM Team, The LLVM compiler infrastructure project", http://llvm.org/.
25. LLVM, "LLVM link time optimization: Design and implementation", https://goo.gl/r3RH2U.
26. Microsoft, "Changes to Functionality in Microsoft Windows XP SP 2", https://goo.gl/928ihY
27. Octane Browser Benchmark, https://chromium.github.io/octane/, 2017.
28. PaX Team. Address Space Layout Randomization, https://goo.gl/Sab9YE, 2001.
29. Aravind Prakash, Xunchao Hu, and Heng Yin, "Strict Protection for Virtual Function Calls in COTS C++ Binaries", In *NDSS*, 2015.
30. Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz, "Counterfeit Object-oriented Programming", In *S&P*, 2015.
31. Std. Performance Evaluation Corp., "SPEC CPU 2006", https://goo.gl/NtmYy8, 2017.
32. SunSpider 1.0.2 JavaScript Benchmark, https://goo.gl/qk9uqg, 2017.
33. Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou, "Practical Control Flow Integrity & Randomization for Binary Executables", In *S&P*, 2013.
34. Mingyi Zhao, Jens Grossklags, and Peng Liu, "An Empirical Study of Web Vulnerability Discovery Ecosystems", In *CCS*, 2015.