HexType: Efficient Detection of Type Confusion Errors for C++

Yuseok Jeon
Purdue University
jeon41@purdue.edu

Priyam Biswas
Purdue University
biswas12@purdue.edu

Scott Carr
Purdue University
carr27@purdue.edu

Byoungyoung Lee
Purdue University
byoungyoung@purdue.edu

Mathias Payer
Purdue University
mathias.payer@nebelwelt.net

ABSTRACT

Type confusion, often combined with use-after-free, is the main attack vector to compromise modern C++ software like browsers or virtual machines.

Typecasting is a core principle that enables modularity in C++. For performance, most typecasts are only checked statically, i.e., the check only tests if a cast is allowed for the given type hierarchy, ignoring the actual runtime type of the object. Using an object of an incompatible base type instead of a derived type results in type confusion. Attackers abuse such type confusion issues to attack popular software products including Adobe Flash, PHP, Google Chrome, or Firefox.

We propose to make all type checks explicit, replacing static checks with full runtime type checks. To minimize the performance impact of our mechanism HexType, we develop both low-overhead data structures and compiler optimizations. To maximize detection coverage, we handle specific object allocation patterns, e.g., placement new or `reinterpret_cast` which are not handled by other mechanisms.

Our prototype results show that, compared to prior work, HexType has at least 1.1 − 6.1 times higher coverage on Firefox benchmarks. For SPEC CPU2006 benchmarks with overhead, we show a 2 – 33.4 times reduction in overhead. In addition, HexType discovered 4 new type confusion bugs in Qt and Apache Xerces-C++.

CCS CONCEPTS

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

KEYWORDS

Type confusion; Bad casting; Type safety; Typecasting; Static_cast; Dynamic_cast; Reinterpret_cast

1 INTRODUCTION

C++ is well suited for large software projects as it combines high level modularity and abstraction with low level memory access and performance. Common examples of C++ software include Google Chrome, MySQL, the Oracle Java Virtual Machine, and Firefox, all of which form the basis of daily computing uses for end-users.

The runtime performance efficiency and backwards compatibility to C come at the price of safety: enforcing memory and type safety is left to the programmer. This lack of safety leads to type confusion vulnerabilities that can be abused to attack programs, allowing the attacker to gain full privileges of these programs. Type confusion vulnerabilities are a challenging mixture between lack of type and memory safety.

Generally, type confusion vulnerabilities are, as the name implies, vulnerabilities that occur when one data type is mistaken for another due to unsafe typecasting, leading to a reinterpretation of the underlying type representation in semantically mismatching contexts.

For instance, a program may cast an instance of a parent class to a descendant class, even though this is neither safe nor allowed at the programming language level 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, it may use data, say, as a regular field in one context and as a virtual function table (vtable) pointer in another. Such type confusion vulnerabilities are not only wide-spread (e.g., many are found in a wide range of software products, such as Google Chrome (CVE-2017-5023), Adobe Flash (CVE-2017-2095), Webkit (CVE-2017-2415), Microsoft Internet Explorer (CVE-2015-6184) and PHP (CVE-2016-3185)), but also security critical (e.g., many are demonstrated to be easily exploitable due to deterministic runtime behaviors).

Previous research efforts tried to address the problem through runtime checks for static casts. Existing mechanisms can be categorized into two types: (i) mechanisms that identify objects through existing fields embedded in the objects (such as vtable pointers) [6, 14, 29, 38]; and (ii) mechanisms that leverage disjoint metadata [15, 21]. First, solutions that rely on the existing object format have the advantage of avoiding expensive runtime object tracking to maintain disjoint metadata. Unfortunately, these solutions only support polymorphic objects which have a specific form at runtime that allows object identification through their vtable pointer. As most software mixes both polymorphic and non-polymorphic objects, these solutions are limited in practice — either developers must manually blacklist unsupported classes or programs end up having unexpected crashes at runtime. Therefore, recent state-of-the-art detectors leverage disjoint metadata for type information. Upon object allocation, the runtime system records the true type of the object in a disjoint metadata table. This approach indeed does not

CCS'17, Oct. 30–Nov. 3, 2017, Dallas, TX, USA.

ISBN 978-1-4503-4946-8/17/10…$15.00
DOI: http://dx.doi.org/10.1145/3133956.3134062

suffer from non-polymorphic class issues, because type information can be accessed without referring vtable pointers.

However, disjoint metadata schemes have to overcome two challenges: (i) due to C++'s low level nature it is hard to identify all object allocations and (ii) the lookup through this disjoint metadata table results in prohibitive overhead. Existing approaches with disjoint metadata precisely exhibit these drawbacks. Because it is difficult to handle all C++ language quirks imposed by developers, they only protect a small fraction of typecasts in practice. Due to the complexity of metadata tracking, existing approaches introduce prohibitive overheads (TypeSan [15] has up to 71.2% overhead for Firefox with a geometric mean of 30.8%; note that TypeSan already improves performance over CaVer [21]). Control-Flow Integrity (CFI) techniques [20, 34–36] verify all indirect control-flow transfers within a program to detect control-flow hijacking. However, these techniques address the type confusion problem only partially if control flow is hijacked, i.e., they detect usage of the corrupted vtable pointer, ignoring any preceding data corruption. Similarly, vtable protection schemes [14, 38] protect virtual calls from vtable hijacking attacks but do not block type confusion attacks. Memory safety mechanisms [24, 26, 32] protect against spatial and temporal memory safety violations but incur prohibitively high overhead in practice. Also, these mechanisms do not protect against type confusion, e.g., they do not stop an `int` array of the correct size from being used in place of an object. Control-flow hijacking protection and memory safety are therefore orthogonal to type confusion detection. Type confusion may be used to cause a memory safety violation. Detecting type confusion allows earlier detection of security violations for these cases.

We propose HexType, a mechanism that protects C++ software from type confusion by making all casts explicit. Each cast in the source language (explicit or implicit, static or dynamic) is turned into a dynamic runtime check. HexType records the type of each object and specific casts are replaced with our instrumentation. We fundamentally address the challenges of earlier work by (i) increasing coverage of typecasting checks and (ii) drastically reducing overhead.

Our prototype implementation of HexType vastly outperforms state-of-the-art type confusion detectors, increasing coverage and often lowering overhead. Our reduced overhead is the result of novel optimization techniques and using an efficient type metadata structure. We leverage an analysis that identifies types that are used in typecasting, allowing us to remove tracing overhead for any objects that are never cast. For the type metadata structure, we design a two-layered data structure that combines a hash table (fast-path) and red-black tree (slow-path) in order to reduce object tracing overhead. Despite performance, our mapping scheme also overcomes limitations of existing work such as relying on fixed addresses for metadata which may run into compatibility issues if applications try to reuse the same addresses

To address the low coverage of related work, we developed allocation detectors that track reuse of pre-allocated memory space cases for new objects (through placement new) and transferring objects through `reinterpret_cast`. Additionally, HexType increases coverage for `dynamic_cast` and `reinterpret_cast` and goes beyond `static_cast` unlike all the previous works. In the case of `dynamic_cast`, HexType replaces the existing inefficient typecasting verification routine with a fast lookup using our metadata. HexType supports reinterpret_cast to increase object tracing coverage and find additional bugs.

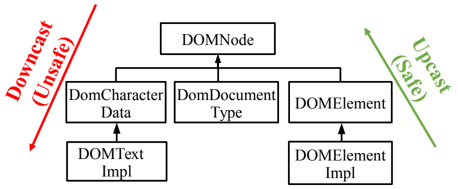


Figure 1: Visualization of an example C++ type hierarchy, showing the directions of (safe) upcasting and (unsafe) downcasting.

Due to our increased coverage, we discovered four new type confusion vulnerabilities (which evaded previous approaches) in two widely-used open source libraries (Qt Base library and Apache Xerces-C++) during our evaluation. For the Firefox benchmarks, HexType increases coverage by 1.1 – 6.1 times compared to TypeSan with some increased performance overhead due to the vast increase in coverage. For SPEC CPU2006 benchmarks with overhead, we show a 2 – 33.4 times reduction in overhead.

Our major contributions can be summarized as:

(1) An open source type confusion detector with low overhead and high coverage (outperforming state-of-the-art detectors);
(2) A novel optimization that greatly reduces the number of objects that need to be tracked (as much as 54% – 100% on SPEC CPU2006), thus reducing overhead;
(3) Design of efficient data structures that use a fast-path ($O(1)$ time complexity) for type information insertion and lookup (with a hit rate of 94.09% and 99.99% on the SPEC CPU2006 and 98.76% and 95.20% for Firefox respectively);
(4) Robust allocation identification implementation that greatly increases coverage (1.1 - 6.1 times over TypeSan on Firefox) combined with also covering alternate casting methods such as placement new;
(5) Discovery of four new vulnerabilities in QT Base library and Apache Xerces-C++;

2 BACKGROUND

In this section, we provide background information on C++'s type system, various cast operations, and previous type confusion detection tools necessary to understand the design and implementation of HexType.

2.1 C++ Classes and Inheritance

C++ is an object-oriented programming language, with classes as the primary abstraction. Classes allow the programmer to define new types. A class can inherit from multiple ancestor classes. The descendent class has all the same members (methods and variables) as its ancestor(s) and optionally additional members defined in the descendent class definition.

In C++, a pointer of type A can be cast into a pointer of another type, type B. This effectively tells the compiler to treat the pointed-to object as being type B.

The crucial question is: when is a typecast safe? The answer depends on the type of the pointed-to object and the destination type (type B in the previous example). Focusing on casting between class types, the security objective of this work, casting from descendant class to ancestor class is always safe since the members of the descendant class are a superset of the members of the ancestor class. This operation is called upcasting. For example, as shown in Figure 1, if we visualize the type hierarchy with the ancestor class at the top and descendants at the bottom, moving up the hierarchy (upcasting) is safe. On the other hand, downcasting, casting from ancestor to descendant, may not be safe if the ancestor misses any member of the descendant class. This is depicted in Figure 2. Such downcasting has been abused by attackers in a wide-range of popular C++ programs, which lead to complete compromises of an underlying system, as recently shown for, e.g., Google Chrome (CVE-2017-5023), Adobe Flash (CVE-2017-2095), Webkit (CVE-2017-2415), Microsoft Internet Explorer (CVE-2015-6184) or PHP (CVE-2016-3185).

2.2 C++ Cast Operations

The C++ syntax allows four different types of casts to meet different requirements of the developer. Each casting type performs unique casting operations, imposing non-trivial security implications. In the following, we provide detailed information on each casting type, particularly focusing on its security aspects in terms of type confusion issues.

The example in Figure 1 shows a cast using `static_cast`, but there are other cast in C++ and their details are important to this work. The other cast types we are concerned with are `dynamic_cast`, `reinterpret_cast`, and C-style typecasting.

```
static_cast<type>(expression)
dynamic_cast<type>(expression)
reinterpret_cast<type>(expression)
const_cast<type>(expression)
```

Static Cast. A `static_cast` casts an object of type A to an object of type B. The check is executed purely at compile time and no runtime check is performed. Due to the static nature of this check, the runtime type of the object is not considered and the check is limited to check if the two types are compatible, i.e., there is a path in the type hierarchy from *expression*'s type and *type* that involves upcasting and/or downcasting.

While not incurring any performance overhead, the safety guarantees of static casts are limited. Therefore, the programmer is responsible that an object of the correct type is used, e.g., guaranteeing that the downcasted object is actually an object of the derived type. In practice, since it is challenging to figure out such compatibility at compile time, this has led to the unfortunate fact that type confusions are dominating vulnerabilities in modern C++ programs [23].

Dynamic Cast. A `dynamic_cast` can safely convert types between classes in the same class hierarchy. Whereas `static_cast` only performs a compile time check, it performs an additional runtime check using heavy-weight metadata, Run Time Type Information (RTTI). As, in general, the dynamic runtime type of an object cannot be determined statically, `dynamic_cast` must leverage runtime type information such as RTTI. RTTI encodes all type related information, and a compiler generates this RTTI per type such that each type has its dedicated RTTI entry in a compiled binary. The RTTI entry essentially forms a recursive structure in that each RTTI entry points to another RTTI entry to represent the class hierarchy. A compiler further appends a reference to the RTTI entry at the end of each virtual function table, so that the RTTI entry can be retrieved at runtime using any virtual address pointing to an object. In other words, since the first field in an object is typically filled with a virtual function table pointer, `dynamic_cast` can find the RTTI entry given an object address using the virtual function table pointer. After locating the corresponding RTTI entry, `dynamic_cast` starts to recursively traverse RTTI to verify the casting correctness (i.e., that the types are compatible). If there is a path on the type hierarchy between *expression*'s type and the target type, then the types are compatible. The types are compatible whenever the type of expression is an descendant of *type* (upcast). The types can also be compatible when *type* is the exact type of the object pointed to by *expression*. If the casting is incorrect (i.e., the type of *expression* and *type* are incompatible), the cast fails in one of two ways:

- If *type* is a pointer type, it returns *NULL*.
- If *type* is a reference type, it throws a pre-defined exception (i.e., `std::bad_cast`).

Due to the design of `dynamic_cast`, its usage is strictly limited to polymorphic objects. As mentioned before, `dynamic_cast` relies on a virtual function table to locate RTTI, but the virtual function table is only present in polymorphic objects. Note that, given these limitations, `dynamic_cast` can only be used for polymorphic types. Thus, compilers simply generate a compile-time error if a `dynamic_cast` is used for a non-polymorphic type. Note that runtime errors are still possible.

Reinterpret Cast. A `reinterpret_cast` converts between any two (potentially incompatible) types. It instructs the compiler to reinterpret the underlying bit pattern of the cast objects. Because it does neither create a copy nor perform any runtime check, a `reinterpret_cast` always incurs zero overhead. From the security standpoint, programmers are responsible to ensure the correctness of `reinterpret_cast` similar to the case in `static_cast`. Since `reinterpret_cast` only changes the object's type, it simply returns the same address. This behavior can cause problems for polymorphic classes or classes with multiple inheritance. For polymorphic classes, `reinterpret_cast` returns a pointer to an object with potentially the wrong vtable pointer as `reinterpret_cast` does not change the memory of the object. If the object uses multiple inheritance, then a pointer to a base class may have the wrong value (not a pointer to the object itself) [5]. However, if the exact source object type information is known then `reinterpret_cast` can be used to: (1) efficiently construct an object without executing the constructor (reusing an old object of the same type) and (2) restoring the actual type if a function returns a `void*` pointing to an object.

```
class Ancestor { int x; };
class Descendant : Ancestor {
  double y;
};
Ancestor *A = new Ancestor();
Descendant *D;
D = static_cast<Ancestor*>(A);
D->y; // error
```

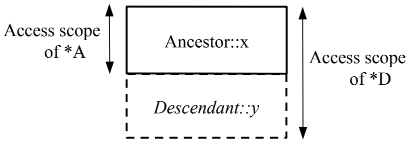
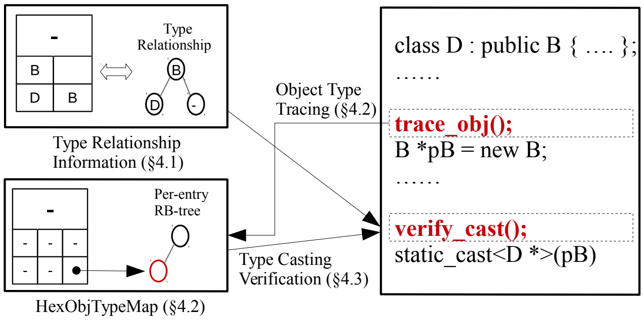


Figure 2: A code example and diagram of a type confusion problem where an ancestor class is incorrectly accessed using a pointer to a descendent class. The static cast results in type confusion and accessing the field D->y results in a memory safety violation.



```
class D : public B { .... };
......
trace_obj();
B *pB = new B;
......
verify_cast();
static_cast<D *>(pB)
```

Figure 3: A system overview of HexType. HexType consists of several modules that analyze type relationship information and insert object tracing and typecasting instrumentation to verify typecasting operation.

Const Cast. A `const_cast` drops cv-qualifier (i.e., const or volatility) from an object specified in the expression. Unlike previously mentioned static and dynamic casts, `const_cast` does not impact type confusion issues because type hierarchies are not involved in this case. `const_cast` still may introduce security issues (5.2.9/11 in ISO/IEC 14882 [19]) if it is used in the wrong context—a read-only object has been accidentally const cast, and thus overwrites such a write-protected object. These issues can be addressed using other memory safety techniques [18, 18, 25, 25] through enforcing per-object write protection. Protecting const casts is therefore orthogonal to the scope of this paper.

C-style Typecast. Although C-style casts are discouraged in C++ programs, compilers allow them to keep backward compatibility. More precisely, if C-style casting (5.4 in ISO/IEC 14882 [19]) is encountered, the compiler translates the cast into a sequence of casts: (i) `const_cast`, (ii) `static_cast`, and (iii) `reinterpret_cast`. In other words, compilers try to cast the objects using the sequence of casts above and use the result of the first cast that succeeds without a compilation error. This in fact implies that, from the standpoint of detecting type confusion issues, it is no different from handling the above three cast types as C-style casting will finally be translated into one of them.

2.3 Defenses against type confusion

Type confusion is a pressing problem and several mechanisms have been proposed to detect and protect against type confusion. As mentioned earlier, the existing defenses can be grouped into two categories: (i) those based on identifying objects based on existing fields embedded in the object themselves (such as vtable pointers) [6, 14, 29, 38]; and (ii) those based on disjoint metadata [15, 21].

CaVer [21] uses disjoint metadata for *all* allocated objects to support non-polymorphic classes without blacklisting. CaVer is the first typecasting detection tool (based on disjoint metadata) that can verify type-casting for non-polymorphic objects. However, CaVer suffers from both security and performance issues — low safety coverage on castings and high runtime overhead.

TypeSan [15] reduces the performance overhead by a factor of 3 – 6 compared to CaVer and increases detection coverage by including C-style allocation (e.g., `malloc`). However, the overhead of both disjoint metadata approaches is still high due to inefficient metadata tracking, e.g., tracking most live objects. Also, while increasing coverage compared to CaVer, TypeSan still has an overall low coverage rate. Especially, TypeSan has 12 ∼ 45% coverage rate for Firefox. These limitations motivated us to design HexType, which overcomes the aforementioned limitations — namely reducing per-cast check overhead, increasing coverage, and providing additional features.

3 THREAT MODEL

Our threat model assumes that the underlying application is benign but contains a type confusion error that an attacker can find and exploit. The primary goal of our defense mechanism is to prevent such type confusion attacks. Our defense mechanism automatically detects such exploitation attempts, avoiding any negative security ramifications. We further assume that the attacker may read arbitrary memory, and thus our detection mechanism is designed not to rely on information hiding or randomization. Attacks not based on type confusion, including control-flow hijacking, integer overflow, and memory corruption, are out of scope and these can be protected by other security hardening techniques. We assume that our instrumentation cannot be removed by the attacker, i.e., our instrumented code is on a non-writable page. The underlying operating system, program loader, and system libraries are in the Trusted Computing Base (TCB).

4 HEXTYPE DESIGN AND IMPLEMENTATION

HexType is a Clang/LLVM-based type confusion detector for C++ programs. During compilation of a target program, HexType generates a HexType-hardened program. During runtime, if HexType detects a type confusion error, the program is terminated with a detailed bug report.

Figure 3 illustrates an overview of HexType. Given the source code as input, HexType generates a type table containing all type relationship information (§4.1) and, at runtime, information about the true types of each allocated object is collected in the object mapping table (§4.2). HexType verifies the correctness of each cast using both the type relationship information and object mapping table (§4.3). HexType leverages a set of optimization techniques to reduce performance overhead during the above processes (§4.4).

4.1 Type Relationship Information

In order to verify typecasting operations, HexType needs to know a valid set of destination types that can be cast from a given source type. Note that compilers keep this information readily available during compilation to check the validity of casts statically, but such checks are inherently limited as the true source type of an object is only known at runtime. C++ applications generally do not keep explicit information about the type hierarchy. This subsection describes how HexType generates and maintains a hierarchical type information for executables and shared libraries. We call this information type table.

During compilation, HexType extracts all type relationship information and prepares metadata for each type. For example, as shown in Figure 1, for the type `DOMElementImpl`, HexType first collects all types that are allowed to be cast (i.e., `DOMElement` and `DOMNode`), each of which is basically a parent class of `DOMElementImpl`. Instead of simply storing a type name in the type table, HexType stores a string hash of the type name to avoid expensive string match operations, enabling O(1) comparisons. HexType exports, per type, a list of hash values as a global variable during the compilation, allowing other libraries to reuse this information. These lists of hash values are sorted to efficiently search value from the target list using binary search during runtime type casting verification. HexType generates one such global variable per type.

```
DOMElementImpl: H(DOMElement), H(DOMNode), ...
```

In order to provide compatibility, HexType allows and the type table includes phantom classes. A phantom class is a parent-child relationship where the data layout of the child is equivalent to the data layout of the parent. HexType allows downcasts from such a child to the parent as such phantom classes are frequently used in practical environments to support interoperability between C and C++.

To manage the type table efficiently, HexType only records each type's relationship information once, following the one definition rule (ODR) [19] of the C++ standard. According to this rule, the type definition of each object must be identical (each object's parent information is always the same) among all source code which will be merged. Therefore, each type will have a uniquely identical list of hash values among all source codes except for phantom classes. Since each object can have a derived class as a phantom class and the set of the phantom classes cannot be determined when each object type is defined (we also have to rely on information from each object's derived class defined site), HexType only needs to update this phantom class information.

4.2 Object Type Tracing

In order to verify typecasting operations at runtime, HexType needs to locate the type information based on the underlying object identified by the source pointer address in the casting operation. Unlike `dynamic_cast`, HexType does not utilize RTTI to retrieve type information due to the following limitations of RTTI: (i) RTTI only provides type information for polymorphic objects (not supporting typecasting verification of non-polymorphic objects); (ii) RTTI incurs expensive typecasting verification costs due to its recursive structure; and (iii) RTTI significantly blows up the size of the compiled binary.

For these reasons, HexType designs a new set of techniques, which aims at maximizing security coverage and minimizing performance overhead. In the following, we first describe how HexType captures the underlying memory semantics with respect to the type information. HexType systematically identifies all object allocation sites, which significantly elevates the coverage for typecasting operations (§4.2.1). Next, we illustrate how HexType maintains such memory semantics at runtime. In order to perform efficient lookup operations, HexType employs a new data structure, type table, which supports both a fast-path for performance efficiency and a slow-path for completeness (§4.2.2).

4.2.1 Tracing Object Type Allocation. The C++ type system is not strongly constrained and thus developers can easily change the object type at runtime as required. This flexibility, though it is one of the main reason of C++'s popularity, introduces several challenges when tracking type information. More precisely, HexType must identify the correct type information imposed to certain runtime memory objects, but dynamic type changes complicate the identification processes.

HexType comprehensively identifies all the sites that assign types, which can be generally categorized into the following two cases depending on when the type assignment is performed—(1) at the time of creating an object and (2) at the time of transferring an object. The first case includes the well known new operator which allocates object memory space through typical system memory allocator (i.e., `malloc()`) and initializes the object by invoking its associated constructor function. The first case also includes *placement new*, which reuses specified memory space and simply invokes the constructor for initialization. For these type allocation sites at object creation time, HexType registers the type of the object in the type table by passing the type information and the base pointer to the registration function. The runtime library function updates the type table with this information

We describe more details how HexType maintains information in §4.2.2.

The second case of type assignments happen while hard-copying objects that have already been constructed. In C++, it is common to copy or move memory objects in memory space for, e.g., object marshaling or when passing objects between allocation spaces. Once the memory object is relocated in memory space, a developer is responsible to reassign the type of underlying memory objects.

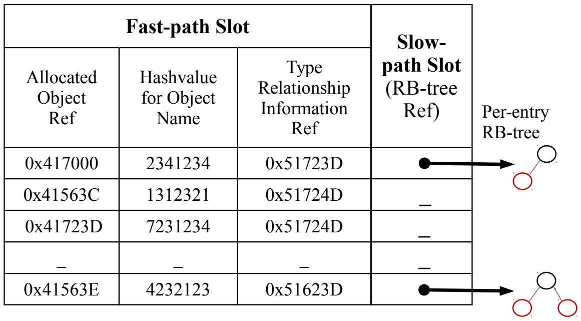
```
1  template<class T, std::size_t N>
2  class static_vector
3  {
4   // properly aligned uninitialized storage for N T's
5   size_t Size = sizeof(T);
6   size_t Align = alignof(T);
7   typename std::aligned_storage<Size, Align>::type d[N];
8   ......
9
10 public:
11  template<typename ...Args> void insert(Args&&... args)
12  {
13    ......
14    // Create an object using placement new
15    new(d+m_size) T(std::forward<Args>(args)...);
16    ......
17  }
18
19  const T& operator[](std::size_t pos) const
20  {
21    // Access an object using repinterpret_cast
22    return *reinterpret_cast<const T*>(d+pos);
23  }
24  ......
25 };
```

Figure 4: Code example for std::aligned_storage using placement new and `reinterpret_cast` to manage type allocation.

Developers can rely on move or copy operators in C++ if the underlying object is a C++ class object constructed through new or placement new operators. Alternatively, they can explicitly specify the type of underlying memory objects using `reinterpret_cast`. This second case is commonly used to work around system constraints. For example, when an object is marshaled and unmarshaled to pass it between different components, `reinterpret_cast` can efficiently construct an object without explicitly executing the class constructor again. To handle `reinterpret_cast`, HexType instruments `reinterpret_cast` to call a runtime function with two pieces of information: (i) destination type and (ii) source address information. In the runtime library function, HexType inserts this information into the type table only if there is no matching entry with `reinterpret_cast`'s source address.

For example, Figure 4 shows how aligned_storage creates and accesses objects. In the initial step, aligned_storage creates uninitialized memory blocks (line 5). In this uninitialized storage, the objects are created using placement new (line 13). Then, we can access the allocated objects using `reinterpret_cast` (line 20).

In fact, previous work including UBSan, CaVer, and TypeSan all fail to generally handle type assignment sites. In the case of UBSan, it cannot capture the type information of non-polymorphic objects as it has to rely on RTTI, resulting in unexpected crashes at runtime. In the case of CaVer and TypeSan, they only consider new operator as type assignment sites and thus they miss all other assignment sites mentioned above. As we will clearly demonstrate in the evaluation section, HexType showed 1.1 – 6.1 times higher coverage on Firefox benchmarks compared to TypeSan.



Fast-path Slot			Slow-path Slot (RB-tree Ref)
Allocated Object Ref	Hashvalue for Object Name	Type Relationship Information Ref	
0x417000	2341234	0x51723D	●
0x41563C	1312321	0x51724D	_
0x41723D	7231234	0x51724D	_
_	_	_	_
0x41563E	4232123	0x51623D	●

Figure 5: A snapshot example of object mapping table, showing how it maps an object using a combination of fast-path and slow-path slots. When HexType looks up type information, an object address is used to obtain the reference to the corresponding object mapping table entry. HexType first matches the fast-path slot. If not present in the fast-path slot, HexType then searches the corresponding red-black tree to find type information (slow-path) and updates the fast-path accordingly.

4.2.2 Mapping Objects to type table. HexType maintains an object mapping table, which maps runtime objects to its associated type information in the type table. More specifically, a key in the object mapping table is an object address and its mapped value is an address pointing to the associated entry within the type table. It is performance critical for HexType to efficiently design this object mapping table, because this mapping process through object mapping table is performed every time HexType attempts to verify the typecasting operations.

We found various object tracking methods in previous works [15, 21]. TypeSan [15] uses a memory shadowing scheme to track global, heap, and stack objects. However, the TypeSan memory shadowing scheme has three limitations: (i) TypeSan uses a fixed address for the metadata table (to enable faster lookups) which may result in compatibility problems if applications reuse the same address, e.g., due to ASLR, which we observed in practice;

(ii) TypeSan only updates objects in the "object to type" mapping table when objects are allocated (it does not delete information when an object is deleted from memory). Therefore stale metadata can create additional problems; (iii) TypeSan's memory shadowing scheme uses more memory resources compared to other non-memory shadowing schemes. CaVer [21] uses a red-black tree to keep track of global and stack objects. However, overhead becomes prohibitive for, e.g., stack objects, as stack objects incur frequent insertions and deletions. Since a red-black tree generally shows $O(logN)$ time complexity to delete, insert, and search.

Toward this end, HexType leverages a new data structure to reduce performance overheads in mapping operations. The key insight for object mapping table is that some objects are accessed much more frequently than others. We therefore designed a data structure that splits object lookup into a fast pass using a hash table and a slow path using a red-black tree, see Figure 5. The first level of our data structure is a hash table. We use the object's address and a simple hash function to locate the entry for a given

```
1  class Base1 { ... };
2  class Base2 { ... };
3
4  // multiple inheritence
5  class Derived: public Base1, public Base2 { ... };
6
7  Derived obj;
8  Derived* dp = &obj;
9
10 // indicates the Dervied's Base2 object
11 Base2* b2p = dp;
12 // static_cast restores the original pointer value
13 Derived* dps = static_cast<Derived*>(b2p);
14 // reinterpret_cast perserves the new pointer value
15 Derived* dpr = reinterpret_cast<Derived*>(b2p);
```

Figure 6: An example of how `reinterpret_cast` results in a type confusion problem.

object. Each hash table entry holds two slots: (1) the fast-path slot for the least recently cast object, which holds the reference to the object (to check if the object matches), the hash of the object's type, and the reference to object's type relationship information (collects all destination types that are allowed to be cast) and (2) the slow-path slot, which holds a reference to a per-slot red-black tree maintaining a complete set of objects that map to the hash table entry. In other words, once HexType locates a hash table entry, it simply reuses the value in the fast-path slot if the object's address in the fast-path slot matches. Otherwise, HexType walks through the red-black tree pointed by the slow-path slot to address collisions. Whenever a lookup in the red-black tree is performed, the fast-path is updated with the most recent object. As a result, our mapping scheme with object mapping table imposes $O(1)$ time complexity for fast-path accesses and $O(logN)$ for slow-path accesses (where N is the number of values in the per-slot red-black tree). In the SPEC CPU2006 C++ benchmarks, our approach uses the fast-path 99.68% of time to update metadata and 100% of the time to lookup information from the type table. We demonstrate that these design choices for the object mapping table are reasonable in the evaluation section in §5.

4.3 Type Casting Verification

We now describe the final step of HexType, typecasting verification, which checks the safety of casting. HexType instruments typecasting operations with additional verification code at compile time. At runtime, this instrumentation locates the object's true type information in the type table and then compares the target type with the expected type at the cast site to determine if the cast is legal.

HexType instruments all type casting operations related to type confusion issues. As described in §2.2, these include `static_cast` sites, where its casting operation performs downcasting. More precisely, HexType instruments additional code invoking a runtime verification function while passing necessary information with respect to casting verification (i.e., a base object pointer subjected to casting and a hash value of a destination type).

Additionally, HexType also verifies `reinterpret_cast`. As mentioned in §2.2, `reinterpret_cast` forces the casting operation by copying the memory bits of a pointer value even though casting types are not compatible. Thus, this operation is security critical if misused. For example, as shown in Figure 6, since `reinterpret_cast` simply returns the same unchanged address (line 15), a pointer to a base class points to a semantically different object, which results in access to an unexpected memory area. In other words, `reinterpret_cast` does not properly adjust the pointer according to the class hierarchy (line 5) as it simply hard-copies a to-be-cast value, compared to `static_cast` which adjusts the pointer.

Once a runtime verification function is invoked at runtime, HexType first locates the object mapping table. Given the base pointer address of an object, HexType computes the hash index within the object mapping table, which returns a reference to the corresponding type table walking through either fast-path or slow-path (§4.2). Using this type table as well as the provided destination type information, HexType reasons about whether the underlying object can be indeed a sub-object of the destination type such that the casting itself is correct in the end. If HexType detects type confusion at runtime, it displays a detailed report that includes allocated object type, expected object type, and the type casting location. This information allows the developer to triage the type casting issue quickly.

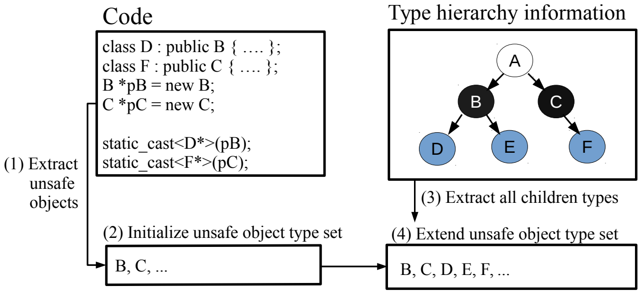
4.4 Optimization

Type casting verification supported by HexType may impose non-negligible performance overhead as it involves additional computation. In order to make HexType a truly practical security tool, we implement a set of performance optimization techniques, namely only tracing unsafe objects, only verifying unsafe casting, and efficient dynamic casting.

Only Tracing Unsafe Object. HexType only traces type information on potentially unsafe objects and does not trace safe objects. We define T as a safe object type if and only if T is never subject to typecasting. T is a potentially unsafe object type otherwise. Since safe object types will never be used for casting at all, HexType does not need to keep track of them to check casting validity. We assume that the source pointer of a safe object always references an object of the correct type as no casting operation in the program exists that breaks this assumption. As illustrated in Figure 7, HexType performs the following two steps to identify unsafe objects. First, it identifies a typecasting-related object set, which can be used for typecasting operations. HexType identifies all type information both at the casting site and for the cast object. An object that is cast can be of type X or any of the child classes of type X. The type casting site therefore must accept all possible subtypes. Next, when instrumenting object allocation sites, HexType selectively instruments allocation only for typecasting-related objects.

While evaluating HexType, we found that tracking stack objects is the most critical performance bottleneck. Thus, considering allocation characteristics of stack objects, we apply a special optimization scheme to conservatively distinguish safe stack objects from unsafe stack objects.

First, we apply CaVer's optimization technique which is based on the observation that the lifetime of a stack object can be relatively well defined with respect to a set of functions the object is active — a function (that the subjected stack object is declared) and all of its


Code
Type hierarchy information
A
B
C
D
E
F
(1) Extract unsafe objects
(3) Extract all children types
(2) Initialize unsafe object type set
(4) Extend unsafe object type set
B, C, ...
B, C, D, E, F, ...

```
class D : public B { .... };
class F : public C { .... };
B *pB = new B;
C *pC = new C;

static_cast<D*>(pB);
static_cast<F*>(pC);
```

Figure 7: An example of how HexType creates a potentially unsafe object type set. In the example, we assume that objects of type B and C are typecast. HexType will identify these potentially unsafe types and all its children types as unsafe.

callee functions, if there are no out-going indirect calls. Thus, only if there are no out-going indirect calls, we perform an escape analysis for the set of those functions so as to ensure that any reference to the stack object never leaves the analyzed functions. Further more, if there are no typecasting operations within these clustered and side-effect free functions, then the analyzed stack objects will never be used for typecasting. In this case, it is truly a safe stack object that does not need to be tracked at runtime.

We apply a more fine-grained analysis for functions that did not pass the previous check: (i) we check whether each stack object in the function is a local variable using SafeStack which is a component of CPI/CPS [20], since SafeStack supports local variables detection and (ii) if these local stack objects are not used for any typecasting operation within this function, we do not need to trace these stack objects.

Only Verifying Unsafe Casting. Clearly HexType does not need to perform runtime verification for a casting operation if it can be proven safe during compilation. We call such a provably safe cast operation a *safe casting*, and *unsafe casting* otherwise. Since HexType supports runtime casting verification, we can leverage an optimization that relies on an imprecise yet conservative static analysis to distinguish these two categories. In other words, given a casting operation, HexType determines if it is safe casting only if HexType can be completely certain at compile time. If HexType cannot determine it is safe in a compile time, HexType simply considers it unsafe casting and falls back to a runtime check.

HexType leverages a conservative backward dataflow analysis to identify safe casting. Starting from a casting site, HexType reasons about type information of an underlying object, i.e., how the underlying object has been allocated. To answer this question, we perform an inter-procedural use-def chain analysis, where the use point is defined as a casting site and the def point is defined as any object allocation sites.

For example, as shown in Figure 8, if the source of typecasting operation uses the address-of operator(&) or array name directly to get the address of the object, HexType can easily determine the source object type and verify the typecasting operation at compile time. Also, we can predict the object type through the use-def chain analysis. In these cases, we can remove HexType's typecasting verification instrumentation and verify typecasting operations during compile time. However, if we cannot determine the source type, HexType will again fall back to a runtime check.

```
1  class T : public S { ... };
2
3  void safe_casting_ex() {
4    S test1;
5    S test2[1000];
6
7    // safe casting : always cast from class S
8    // (case 1)
9    static_cast<T*>(&test1);
10
11   // (case 2)
12   static_cast<T*>(test2);
13
14   // (case 3)
15   S *local_obj_ptr = &test1;
16   static_cast<T*>(local_obj_ptr);
17 }
18
19 void unsafe_casting_ex() {
20   // unsafe casting : type is hard to dertermine
21   S* obj_ptr = external_func();
22   static_cast<T*>(obj_ptr);
23 }
```

Figure 8: An example for safe and unsafe casting. The three examples (line 9, 12, and 16) are all safe casting. For the first two examples, each typecasting operation obtains the object address using the address-of operator and the array name. In the third example (line 14), we can simply determine the object type using use-def chain analysis. The last example (line 22) is an unsafe casting case when we cannot track the object type (i.e., external function).

Efficient Dynamic Casting. Since HexType offers efficient runtime casting verification, the existing `dynamic_cast` can be optimized accordingly. HexType therefore replaces each `dynamic_cast` with our fast lookup. In order to preserve the runtime semantics of `dynamic_cast` as dictated by the C++ standard, HexType takes additional steps in response to an incorrect casting detected in runtime. As described in §2.2, HexType returns NULL for a pointer-typed casting and throws an exception for a reference-typed casting. This optimization can be especially useful if applications heavily rely on dynamic casting.

4.5 Implementation

We have implemented HexType, as shown in Figure 10, based on the LLVM Compiler infrastructure project [22] (version 3.9.0). The HexType implementation consists of 4,677 lines of code that we added to Clang, an LLVM Pass, and our compiler-rt runtime library. HexType's LLVM Pass (i) creates type relationship information, and (ii) instruments allocations of unsafe objects to record allocated object type information into our object mapping table. Also, we modify Clang to (i) instrument all downcast sites (the pointer type of casting operation is one of the parent objects of destination type),

```
Class T : public S { ... };

// (1) if type is a pointer and invalid downcast,
//     it returns NULL
S *c_obj = new S;
T* d = dynamic_cast<T*>(c_obj);

if(!d) {
        // invalid downcast
}

// (2) if type is a reference type and invalid downcast,
//     it throws a exception
try
{
    S b;
    T& rd = dynamic_cast<T&>(b);
}
catch (std::bad_cast& bc)
{
        // invalid downcast
}
```

Figure 9: An example of how to handle type confusion errors using dynamic_cast. HexType will provide the same error handling behaviors when HexType detects type confusion errors.

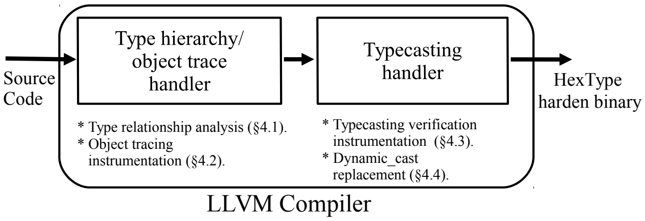

Source Code
Type hierarchy/ object trace handler
* Type relationship analysis (§4.1).
* Object tracing instrumentation (§4.2).
Typecasting handler
* Typecasting verification instrumentation (§4.3).
* Dynamic_cast replacement (§4.4).
HexType harden binary
LLVM Compiler

Figure 10: Overview of HexType's implementation. HexType consists of several compiler passes in clang and LLVM that insert object tracing and typecasting instrumentation and a corresponding runtime library.

and (ii) handle dynamic_cast, reinterpret_cast, and placement new. At runtime, the instrumentation invokes HexType's runtime library functions to update object allocation information into object mapping table, and verifies typecasting operation using type relationship information and object mapping table.

5 EVALUATION

In this section, we evaluate HexType focusing on following aspects: (i) the detection coverage (§5.1); (ii) newly discovered vulnerabilities by HexType (§5.2); (iii) the efficiency of object mapping table (§5.3); and (iv) runtime overhead (§5.4).

Experimental Setting. All evaluations were performed on Ubuntu 16.04.2 LTS with a quad-core 3.60GHz CPU (Inter i7-4790), 250GB SSD-based storage, 1TB HDD, and 16GB RAM.

Evaluation Target Programs. We have applied HexType to the following programs: all seven C++ benchmarks from SPEC CPU2006 [7] and Firefox [11]. For Firefox, we use Octane [13] and Dromaeo [10] benchmark suites. Moreover, in order to compare HexType with previous work, we applied TypeSan as well, and ran these programs under the same configuration. For CaVer, we use the numbers from the paper since CaVer was developed almost three years ago and we encountered compatibility issues with the current test environment and software (i.e., Firefox).

	# of casting	TypeSan	HexType-no-opt		HexType	
		%	%	×	%	×
omnetpp	2,014m	100	100	1	100	1
xalancbmk	283m	89.5	99.8	1.1	99.8	1.1
dealII	3,596m	100	100	1	100	1
soplex	209k	100	100	1	100	1
ff-octane	623m	12	73.3	6.1	56.5	4.7
ff-drom-js	4,229m	23	80.1	3.5	59.4	2.6
ff-drom-dom	10,786m	45	88.8	2	54.9	1.2

Table 1: The evaluation of typecasting verification coverage against SPEC CPU2006 and browser benchmarks. Columns with % present a coverage ratio and columns with × present a coverage improvement ratio (i.e., HexType's coverage divided by TypeSan's coverage).

5.1 Coverage on Typecasting

One of the primary goals of HexType is in increasing the typecasting coverage such that HexType can ensure that all different typecasting operations are correctly performed. To evaluate typecasting coverage, we counted how many typecasting operations were verified at runtime (shown in Table 1). We used two different versions of HexType in this experiment, where each version either turned off or on the optimization techniques presented in §4.4 (denoted as HexType-no-opt and HexType, respectively). For TypeSan, we referred to the evaluation numbers presented in the paper [15].

For SPEC CPU2006, HexType verifies almost all typecasting operations — 100% for omnetpp, dealII, and soplex, and 99.8% for xalancbmk. Compared to TypeSan, HexType improves the coverage number on xalancbmk (i.e., improved from 89% to 99.8%). This is because xalancbmk heavily uses placement new to allocate objects, for which TypeSan looses information about the object at runtime. Thus TypeSan fails to resolve type information associated with such objects. However, as described in §4.2.1, HexType correctly handles these new operator allocations, which significantly raised the coverage ratio.

For Firefox, depending on the benchmark suite, HexType successfully covers typecasting operations: ranging from 73% to 88% with HexType-no-opt; and ranging from 54% to 59% with HexType. During our evaluation, we found that HexType's coverage rate drops after applying our optimizations due to interactions with Firefox's complex object allocation patterns and how our optimizations handle and track allocations in LLVM/Clang. While we are investigating and plan to fix this issue in the future, HexType with optimization still shows better coverage rate than TypeSan. Most of the missing type casts in xalancbmk and Firefox result from application-specific allocation patterns. More specifically, Firefox creates a custom storage pool (typed as an array of char), and manipulates the pool using memcpy or direct object initialization (e.g., `data.key = key; data.index = index; ..`). Xalancbmk also uses a special storage pool (SerializeEngine) that manages objects directly without calling memory allocation functions. As these allocation patterns cannot be detected by HexType during the instrumentation phase, HexType cannot track runtime object types that are allocated through these patterns. Handling these missing allocations is challenging. A naive approach would trace the custom storage pool (allocated as char array) and its low-level allocation patterns using memcpy or direct object initialization. This would unfortunately result in high overhead. Alternatively, we propose to modify the few locations in Firefox and annotate the object allocation accordingly. Although this coverage ratio in Firefox may not be as impressive as HexType's result of the SPEC CPU2006 benchmarks, we emphasize it is significantly improved from the state-of-the art tool, TypeSan. TypeSan only covered 27.75% of Firefox's typecasting on average, 52.98% and 29.18% less than HexType-no-opt and HexType, respectively, highlighting HexType's advantage in identifying allocation sites (§4.2.1).

5.2 Newly Discovered Vulnerabilities

During the course of evaluating HexType by running the set of target programs, we discovered four new type confusion vulnerabilities. In particular, HexType reported four vulnerable cases in the Qt base library while evaluating Wireshark and Apache Xerces-C++, all of which have been confirmed and patched by the corresponding developer communities. For Apache Xerces-C++, HexType found two new vulnerabilities. These vulnerabilities occurred due to type confusion issues between `DOMNodeImpl` (indicated by `DOMNode` type pointer) and `DOMTextImpl`. Since the DOMNodeImpl object is allocated using placement new from a pre-allocated memory pool previous approaches cannot trace these objects. Therefore, these vulnerabilities were not detected by previous schemes such as CaVer or TypeSan.

In addition, HexType found two new vulnerabilities in the Qt-based library. The Qt team already patched our reported type confusion bugs [30]. HexType reported type confusion issues when Qt performs a casting from `QMapNodeBase` (base class) to `QMapNode` (derived class). Since QMapNode is not a subobject of QMapNodeBase, it violates C++ standard rules 5.2.9/11 [19] (down casting is undefined if the object that the pointer to be cast points to is not a subobject of down casting type) and causes undefined behavior.

These new vulnerabilities discovered by HexType clearly demonstrate the security advantage of HexType, especially compared to other previous work including TypeSan and CaVer. We would like to further point out that these new type confusion vulnerabilities were discovered only with basic benchmark workloads. In the future, we plan to run HexType under a fuzzing framework such as AFL [37]. to discover more security critical vulnerabilities related to type confusions.

5.3 Efficiency of Object Tracing

Recall that the key runtime functions that HexType performs are (1) keeping track of object types (at the time of object allocation) and (2) looking up an object type (at the time of type casting). As described in §4.2.2, we designed object mapping table to efficiently handle these operations leveraging both a fast-path and a slow-path. Therefore, the performance efficiency of object mapping table clearly relies on the hit ratio of the fast-path (i.e., the number of operations that only access the hash table) such that HexType does not need to consult the slow-path (i.e., accessing not only the hash table but also the corresponding red-black tree) in most cases.

	allocated objects			fast-path hit ratio (%) (update)	fast-path hit ratio (%) (lookup)
	stack	heap	global		
omnetpp	1m	478m	601	99.999	100
xalancbmk	3,150m	45m	3,098	99.998	99.999
dealII	497m	283m	200	99.988	100
soplex	21m	639m	197m	99.691	100
ff-octane	593m	7m	125k	98.820	98.649
ff-drom-js	2,875m	11m	125k	99.645	98.426
ff-drom-dom	34,900m	607m	125k	99.706	94.099

Table 2: The number of traced objects and its fast-path hit ratio when HexType lookup/update these objects into our the object mapping table.

	# of object	safe casting related object (%)	safe stack objects (%)	total safe objects rate (%)
omnetpp	480m	54.76	0.107	54.767
xalancbmk	3,196m	99.42	3.50	99.42
dealII	781m	83.81	51.07	83.81
soplex	858m	97.75	0.76	97.87
povray	6,550m	100	0.18	100
astar	28m	100	1.31	100
namd	2m	100	0	100
ff-octane	600m	42.69	1.96	44.11
ff-drom-js	2,491m	39.99	1.42	40.26
ff-drom-dom	37,538m	21.33	0.95	21.54

Table 3: The number of safe objects identified by HexType's optimization algorithm. HexType does not keep track of these safe objects.

Table 2 lists the fast-path hit ratios while running the set of evaluation target programs. Overall, most of operations showed high fast-path hit ratios, ranging from 98.820% and 99.999% to update object mapping table and from 94.099% and 100% to lookup object mapping table. This high fast-path hit ratio was also maintained when HexType was running large-scale programs such as Firefox, which creates more than 37,000 million objects at runtime. This result implies that the design decision of the object mapping table is efficient enough to support a wide range of programs, which in turn significantly helped HexType to reduce runtime impact.

	CaVer	Type San	HexType		
	%	%	%	$\times_1$	$\times_2$
omnetpp	NA	49.13	9.69	NA	5.1
xalancbmk	29.6	41.35	1.25	23.7	33.1
dealII	NA	78.23	13.13	NA	6
soplex	20.0	1.16	0.76	26.3	1.5
astar	NA	0.36	0.34	NA	1.1
namd	NA	-0.37	-0.37	NA	1
povray	NA	26.73	0.8	NA	33.4
ff-octane	45	19.37	30.87	1.5	-1.6
ff-drom-js	40	25.18	25.89	1.5	-1.03
ff-drom-dom	55	97.15	126.03	-2.3	-1.3

Table 4: SPEC CPU2006 and browser benchmark performance overhead for CaVer, TypeSan, and HexType. The $\times_1$ column denotes the ratio between CaVer and HexType and $\times_2$ denotes between TypeSan and HexType.

5.4 Performance Overhead

In order to understand performance impacts imposed by HexType, this subsection measures performance overhead in terms of runtime speed. Table 4 shows the performance overhead on the SPEC CPU2006 and Firefox, handling placement new and `reinterpret_cast`. For all seven C++ benchmarks in SPEC CPU2006, HexType outperformed previous work in all cases. This is largely because of HexType's optimization algorithms (§4.4) as well as object mapping table designs (§4.2). To clearly understand these, Table 3 reports how many objects HexType identified as safe objects. With the help of the optimization algorithm, HexType was able to dramatically reduce the number of objects to be traced — reduced from 83% to 100% of tracing for all cases except omnetpp. For omnetpp, the number of casting related classes (unsafe objects) is higher than other cases. However, we can reduce almost 54% object of the tracing overhead. Interestingly, out of the seven SPEC CPU2006 C++ benchmarks that we ran, povray, astar, and namd do not perform any typecasting operation that HexType has to verify at runtime. This implicates that HexType will have zero overhead for these cases since there are no object tracing and typecasting operation. In comparison, TypeSan imposes 26.73% overhead for povray.

In the case of omnetpp and dealII, HexType has shown significantly better performance due to HexType's optimization on replacing `dynamic_cast` (§4.4). This optimization technique can show strong performance improvements, particularly for the applications heavily relying on `dynamic_cast`. We analyzed programs in our evaluation set, and found that two SPEC CPU2006 C++ benchmarks, dealII and omnetpp, perform a huge number of `dynamic_cast`, 206 M and 47 M number, respectively. Therefore, we replaced `dynamic_cast` in our verification routines which reduced the dealII's performance overhead by 4%.

For Firefox, HexType showed similar or higher overhead than TypeSan. Note that, when assessing performance, HexType vastly extends coverage compared to TypeSan (past the differences in coverage).

Moreover, while HexType reduced object tracing by nearly 52 – 100% in SPEC CPU2006, it only reduced the number of traced objects by about 21 – 44% in Firefox. We also suspect this is because of the Firefox's runtime characteristic — almost all objects in Firefox, as shown in Table 2, are allocated on the stack. We note that TypeSan's object mapping scheme comes with a security rist as it never removes object type information. As a result, if the stack location of a former properly allocated object is used in a casting operating, it may be interpreted as a valid object. However, since HexType properly deletes those information when the stack returns, HexType does not suffer from these security issues.

6 DISCUSSION

Coverage. This paper extends the coverage of type confusion detection, particularly in `dynamic_cast` and `reinterpret_cast`. In the future, it is also possible to handle more subtle type confusion issues such as `const_cast`'s undefined behavior or union's type confusion problem. In the case of `const_cast`, it is only safe if we are casting a variable that was originally non-const. Otherwise, a program may modify an object that should be non-mutable, as `const_cast` removes type-based write protection imposed on const objects. Union can also cause a type confusion problem when the attribute value which indicates the type of union is misused. A union data type is similar to a struct data type since it consists of a number of members with different names and types, each of which can be referred to individually. However, unlike the struct type, since union members all occupy the same location in memory, developers should use them mutually exclusively. The existence of these types therefore introduces the possibility of mistakenly referring to a member of a union that is invalid. For example, this problem can lead to an information leak [1].

Fuzzing for type violations. Since HexType can identify type confusion issues at runtime, HexType can be utilized to find new type confusion vulnerabilities with the help of fuzzing frameworks such as AFL [37]. AFL is typically deployed with ASan [32], an LLVM-based sanitizer that checks for (partial) memory safety violations, to increase fuzzing throughput and precision. Without ASan, AFL detects only memory safety violations that result in a segmentation fault and cannot detect silent corruption. As AFL already supports ASan, integrating another sanitizer like HexType will be straight-forward, thereby extending AFL to trigger and detect dangerous type confusion vulnerabilities as well.

7 RELATED WORK

In this section, we summarize previous research works on typecast verification. HexType focuses on type confusion attacks that violate pointer semantics in typecasting operations. CaVer [21] first addressed such exploits due to type casting verification and identified eleven security vulnerabilities due to bad typecasting. Next, TypeSan [15] improved the performance and coverage over CaVer. Similar to HexType, both CaVer and TypeSan are implemented on top of the LLVM compiler framework, instrumenting code during compile time. For metadata allocation both CaVer and TypeSan use a disjoint metadata scheme. TypeSan uses a shadow memory scheme for metadata and CaVer implements a per thread red-black tree for stack objects and shadow memory for the heap. However, these schemes inhibit the identification of overall object allocation and increase the overhead. HexType uses a global, whole address space two layer object-to-type mapping scheme to reduce overhead and supports additional object allocation patterns through placement new and `reinterpret_cast`. Hence HexType vastly increases coverage compared to the aforementioned approaches. Performance is comparable despite the increased coverage.

UBSan [29], another typecast verification framework, works only for polymorphic classes. It relies on runtime type information (RTTI) and instruments only static_cast and checks the casting during runtime. Thus it can only handle polymorphic classes problem as well as requires manual source modification. This makes it difficult to use in large projects.

Several Control-Flow Integrity (CFI) techniques [2, 3, 8, 12, 34, 40] ensure the integrity by checking any invalid control-flow transfer within the program. However, these techniques address the type confusion problem only partially if control-flow hijacking is performed via type exploitation. Similarly, defenses [17, 31, 38, 39] that protect virtual calls from vtable hijacking attacks considers only the type of the virtual calls. These schemes do not address the overall bad casting problems. Another control-flow hijack mitigation technique is Code Pointer Integrity (CPI) [4, 20], which guarantees the integrity of all the code pointers in a program. This approach can prevent the accessibility of corrupted pointers, but does not block type casting attacks.

Bad type casting can lead to memory corruption attacks where an attacker can potentially get access to out-of-bounds memory of the cast object. Such attacks can be identified by existing mechanisms. Defense techniques focusing on memory corruption [9, 16, 18, 25, 27, 28, 33] can detect exploits if a type confusion attack leads to memory access past the cast object. These techniques efficiently detect such attacks, but unlike HexType they cannot address type confusion issues. another kind of memory corruption

8 CONCLUSION

Type casting vulnerabilities are a prominent attack vector that allows exploitation of large modern software written in C++. While allowing encapsulation and abstraction, object oriented programming as implemented in C++ does not enforce type safety. C++ offers several types of type casts and some are only checked statically and others not at all, at runtime an object of a different type can therefore incorrectly pass a type cast. To detect these illegal type casts, defenses need to both track the true allocated type of each object and replace all casts with an explicit check.

HexType tracks the true type of each object by supporting various allocation patterns, several of which (such as placement new and `reinterpret_cast`) were not handled in previous work. While previous work focused only `static_casts`, HexType also covers `dynamic_cast` and `reinterpret_cast`. To limit the overhead of these online type checks, HexType both reduces the amount of incurred checks by removing checks that can be proven correct statically and limiting the overhead per check due to a set of optimizations.

Our prototype results show that HexType has at least 1.1 – 6.1 times higher coverage on Firefox benchmarks. For SPEC CPU2006 benchmarks with overhead, we show a 2 – 33.4 times reduction in overhead. In addition, HexType discovered 4 new type confusion bugs in Qt and Apache Xerces-C++. The open-source version of HexType is available at https://github.com/HexHive/HexType.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers, Nathan Burow, and Scott Carr for their detailed and constructive comments. This material is based in part upon work supported by the National Science Foundation under awards CNS-1513783 and CNS-1657711, by ONR award N00014-17-1-2513, and by Intel Corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] Online; accessed 17-May-2017. Webkit CSS Type Confusion. http://em386.blogspot.com/2010/12/webkit-css-type-confusion.html. (Online; accessed 17-May-2017).
[2] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *CCS*.
[3] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2018, preprint: https://arxiv.org/abs/1602.04056. Control-Flow Integrity: Precision, Security, and Performance. *Comput. Surveys* 50, 1 (2018, preprint: https://arxiv.org/abs/1602.04056). DOI: https://doi.org/10.1145/3054924
[4] Scott A. Carr and Mathias Payer. 2017. DataShield: Configurable Data Confidentiality and Integrity. In *AsiaCCS: ACM Symp. on InformAtion, Computer and Communications Security*. DOI: https://doi.org/10.1145/3052973.3052983
[5] CERT. Online; accessed 17-May-2017. the CERT C++ Coding Standard (5 The Void section). https://www.securecoding.cert.org/confluence/display/cplusplus/5+The+Void/. (Online; accessed 17-May-2017).
[6] Clang. Online; accessed 17-May-2017. Clang 3.9 documentation - Control Flow Integrity. http://clang.llvm.org/docs/ControlFlowIntegrity.html. (Online; accessed 17-May-2017).
[7] Standard Performance Evaluation Corporation. Online; accessed 17-May-2017. SPEC CPU 2006. http://www.spec.org/cpu2006. (Online; accessed 17-May-2017).
[8] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Oakland: IEEE Symp. on Security and Privacy*.
[9] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. 2007. Secure virtual architecture: A safe execution environment for commodity operating systems. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 351–366.
[10] The Mozilla Foundation. Online; accessed 17-May-2017. DROMAEO, JavaScript Performance Testing. https://www.webkit.org/perf/sunspider/sunspider.html. (Online; accessed 17-May-2017).
[11] The Mozilla Foundation. Online; accessed 17-May-2017. Mozilla Firefox. https://www.mozilla.org/firefox. (Online; accessed 17-May-2017).
[12] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. 2016. Fine-Grained Control-Flow Integrity for Kernel Software. In *EuroSP: IEEE European Symp. on Security and Privacy*.
[13] Google. Online; accessed 17-May-2017. Octane Benchmark. https://code.google.com/p/octane-benchmark. (Online; accessed 17-May-2017).
[14] Istvan Haller, Enes Goktas, Elias Athanasopoulos, Georgios Portokalidis, and Herbert Bos. 2015. ShrinkWrap: VTable protection without loose ends. In *ACSAC*.
[15] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. 2016. TypeSan: Practical Type Confusion Detection. In *23rd ACM SIGSAC Conference on Computer and Communications Security*.
[16] Reed Hastings and Bob Joyce. 1991. Purify: Fast detection of memory leaks and access errors. In *In proc. of the winter 1992 usenix conference*. Citeseer.
[17] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2014. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks.. In *NDSS*.
[18] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C.. In *USENIX Annual Technical Conference, General Track*. 275–288.
[19] JTC1/SC22/WG21. Online; accessed 17-May-2017. ISO/IEC 14882:2014 Programming Language C++. http://www.iso.org/iso/catalogue_detail.htm?csnumber=64029. (Online; accessed 17-May-2017).
[20] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-pointer Integrity. In *OSDI*.
[21] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. 2015. Type Casting Verification: Stopping an Emerging Attack Vector. In *USENIX Security*. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/

presentation/lee

[22] llvm. Online; accessed 17-May-2017. The LLVM Compiler Infrastructure Project. http://llvm.org/. (Online; accessed 17-May-2017).

[23] Microsoft. Online; accessed 17-May-2017. Microsoft Security Intelligence Report. https://www.microsoft.com/security/sir. (Online; accessed 17-May-2017).

[24] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. In *ACM Sigplan Notices*, Vol. 44. ACM, 245–258.

[25] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. *ACM Sigplan Notices* 44, 6 (2009), 245–258.

[26] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *ACM Sigplan Notices*, Vol. 45. ACM, 31–40.

[27] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27, 3 (2005), 477–526.

[28] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, Vol. 42. ACM, 89–100.

[29] Google Chromium Project. Online; accessed 17-May-2017. Undefined Behavior Sanitizer. https://www.chromium.org/developers/testing/undefinedbehaviorsanitizer. (Online; accessed 17-May-2017).

[30] QT Code Review. Online; accessed 17-May-2017. Type confusion: From QMapNodeBase to QMapNode. https://codereview.qt-project.org/#/c/191188/. (Online; accessed 17-May-2017).

[31] Pawel Sarbinowski, Vasileios P Kemerlis, Cristiano Giuffrida, and Elias Athanasopoulos. 2016. VTPin: practical VTable hijacking protection for binaries. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 448–459.

[32] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX, Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[33] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker.. In *USENIX Annual Technical Conference*. 309–318.

[34] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-edge Control-flow Integrity in GCC & LLVM. In *USENIX Security*. 15. http://dl.acm.org/citation.cfm?id=2671225.2671285

[35] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical Context-Sensitive CFI. In *CCS*.

[36] Victor van der Veen, Enes Goktas, Moritz Contag, Andre Pawlowski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A Tough Call: Mitigating Advanced Code-Reuse Attacks At The Binary Level. In *IEEE S&P*.

[37] M. Zalewski. Online; accessed 17-May-2017. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/. (Online; accessed 17-May-2017).

[38] Chao Zhang, Scott A Carr, Tongxin Li, Yu Ding, Chengyu Song, Mathias Payer, and Dawn Song. 2016. VTrust: Regaining Trust on Virtual Calls. In *NDSS*.

[39] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. 2015. VTint: Protecting Virtual Function Tables' Integrity.. In *NDSS*.

[40] Mingwei Zhang and R Sekar. 2013. Control flow integrity for COTS binaries. In *SEC: USENIX Security Symposium*.

A PSEUDO CODE

```
# (1) Insert typecasting verification instrumentation (Clang)
def emitiHexTypeCastVerification():
  for targetTypeCast in range(allTypeCastSet):
    emitTypeCastVerifyInstrumenation(targetTypeCast):

# (2) Verify typecasting (runtime library)
def verifyTypecasting(src_addr, dst_addr, dstTyHashValue):
  # Check parentset (safecast)
  # if destination type is one of source object's parent type,
  # this is a SAFECAST.
  srcType = getTypeInfoFromObjTypeMap(src_addr):
  srcParentSet = srcType.ParentSetRef:
  if srcParentSet.binarysearch(dstTyHashValue):
      return SAFECAST:

  # Check phantom (safecast)
  # Although destination type is not of
  # source object's parent type,
  # it would be phantom class.
  dstPhantomSet = getPhantomObjSet(dstTyHashValue):
  if dstPhantomSet.binarysearch(srcType.hashValue):
      return SAFECAST:

  return BADCAST:
```

Figure 11: Algorithm for verifying typecasting

```
# (1) Stack object tracing (LLVM pass)
def stackObjTracing():
  for function in range(module):
    for block in range(function):
      for inst in range(block):
        if isStackObjAllocaInst(inst):
          addObjUpdateInstrumentation(getAllocTypeInfo(inst)):
          # for stack object Remove
          StackObjTracingSet.insert(getAllocTypeInfo(inst)):
    # Call remove instrumentation handling function
    stackObjRemove(function, stackObjTracingSet):

def stackObjRemove(function, stackObjTracingSet):
  for targetObjInfo in range(stackObjTracingSet):
    addObjRemoveInstrumentation(targetObjInfo):

# (2) Heap object tracing (LLVM pass)
def heapObjTracing():
  for function in range(module):
    for block in range(function):
      for inst in range(block):
        if isHeapAllocCall(inst):
          addObjUpdateInstrumentation(getAllocTypeInfo(inst)):
        if isFreeCall(inst):
          addObjRemoveInstrumentation(getAllocTypeInfo(inst)):

# (3) Global object tracing (LLVM pass)
def globalObjTracing():
  GlobalFun = FunctionCreate():
  for targetObjInfo in range(getAllGlobalsObjInfo()):
    addObjUpdateInstrumentation(targetObjInfo):
  # Insert this function (which has object add instrumentation)
  # into global constructor
  appendToGlobalCtors(module, GlobalFun):

# (4) Update object alloc/remove information (runtime library)
# Add object allocation information
# into the HexObjTypeMap (two-level table)
def updateHexObjTypeMap(objAllocInfo):
  index = getHashValue(objAllocInfo.addr):
  # Use fast-path slot (when objTypeMap[index] is empty)
  if objTypeMap[index].addr == NULL:
    updateObjTypeMap(index, objAllocInfo):
  # use slow-path slot (Red-black Tree)
  else:
    # Move collision object (old object) info into RB-Tree
    # Then, insert new object into the objTypeMap
    rbTreeInsert(ObjTypeMap[index]):
    updateObjTypeMap(index, objAllocInfo):

# Remove object allocation information from
# the HexObjTypeMap (two-level table)
def removeHexObjTypeMap(objAddr):
  index = getHashValue(objAddr):
  # Use fast-path slot
  if objTypeMap[index].addr == objAddr:
    objTypeMap[index].addr = NULL:
  # Use slow-path slot
  else:
    rbTreeRemove(ObjTypeMap[index]):
```

Figure 12: Algorithm for tracing objects at runtime

B NEW TYPE CONFUSION BUGS

```
// New Apache Xerces-C++ type confusion vulnerability
// (Code location) xercesc/dom/imple/DOMCasts.hpp, line 146
// (Description) p is pointing to the object allocated as
//  DOMTextImpl, and it is casted into DOMElementImpl.
//  Since DOMElementImpl is not a subobject (parent) of
//  DOMTextImpl, it is violating C++ standard rules 5.2.9/11
//  in [expr.static.cast] (down casting is undefined if
//  the object that the pointer to be casted points to is
//  not a suboject (parent) of down casting type) and
//  causes undefined behaviors.
static inline DOMNodeImpl *castToNodeImpl(const DOMNode *p) {
  DOMElementImpl *pE = (DOMElementImpl *) p;
  return &(pE->fNode);
}

// HexType type confusion report
== Type confusion Report ==
FileName : xercesc/dom/impl/DOMCasts.hpp Line: 99
[From] (hashValue: 1670590304: DOMTextImpl)
[To] (hashValue: 2789966681: DOMElementImpl)

(Call Stack Info)
0x7f1206da5c92:
(xercesc_3_1::castToNodeImpl(xercesc_3_1::DOMNode const*)+0x42)
0x7f1206da643d:
(xercesc_3_1::DOMParentNode::
appendChildFast(xercesc_3_1::DOMNode*)+0x2d)
........
0x7f1203335830:
(__libc_start_main+0xf0)
0x4096e9:
(_start+0x29)
```

Figure 13: A type confusion bug in Apache Xerces-C++ discovered by HexType

```
// New QT type confusion vulnerability
// (Code location) qt5/QtCore/qmap.h, line: 189
// (Description) Header(QMapNodeBase) is casted into QMapNode.
//  However, since QMapNode is not a subobject of QMapNodeBase,
//  it is violating C++ standard rules 5.2.9/11
//  and causes undefined behaviors.
template <class Key, class T>
struct QMapData : public QMapDataBase
{
  typedef QMapNode<Key, T> Node;
  ..............
  const Node *end() const {return static_cast<const Node *>(&header);}
  Node *end() {return static_cast<Node *>(&header);}
  .............
}

// HexType type confusion report
== Type confusion Report ==
FileName : /usr/include/x86_64-linux-gnu/qt5/QtCore/qmap.h Line: 189
[From] (hashValue: 980699179: QMapNodeBase)
[To] (hashValue: 1458345177: QMapNode)

(Call Stack Info)
0x62fa7c:
(QMapData<double, QCPData>::end()+0x3c)
................
0x7fc274403830:
(__libc_start_main+0xf0)
0x47ed19:
(_start+0x29)
```

Figure 14: A type confusion bug in QT discovered by HexType