# Type Casting Verification: Stopping an Emerging Attack Vector

Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee,
*Georgia Institute of Technology*

https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/lee

This paper is included in the Proceedings of the
24th USENIX Security Symposium

August 12–14, 2015 • Washington, D.C.

ISBN 978-1-939133-11-3

# Type Casting Verification: Stopping an Emerging Attack Vector

Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee

School of Computer Science
Georgia Institute of Technology

## Abstract

Many applications such as the Chrome and Firefox browsers are largely implemented in C++ for its performance and modularity. Type casting, which converts one type of an object to another, plays an essential role in enabling polymorphism in C++ because it allows a program to utilize certain general or specific implementations in the class hierarchies. However, if not correctly used, it may return unsafe and incorrectly casted values, leading to so-called *bad-casting* or *type-confusion* vulnerabilities. Since a bad-casted pointer violates a programmer's intended pointer semantics and enables an attacker to corrupt memory, bad-casting has critical security implications similar to those of other memory corruption vulnerabilities. Despite the increasing number of bad-casting vulnerabilities, the bad-casting detection problem has not been addressed by the security community.

In this paper, we present CAVER, a runtime bad-casting detection tool. It performs program instrumentation at compile time and uses a new runtime type tracing mechanism—the type hierarchy table—to overcome the limitation of existing approaches and efficiently verify type casting dynamically. In particular, CAVER can be easily and automatically adopted to target applications, achieves broader detection coverage, and incurs reasonable runtime overhead. We have applied CAVER to large-scale software including Chrome and Firefox browsers, and discovered 11 previously unknown security vulnerabilities: nine in GNU `libstdc++` and two in Firefox, all of which have been confirmed and subsequently fixed by vendors. Our evaluation showed that CAVER imposes up to 7.6% and 64.6% overhead for performance-intensive benchmarks on the Chromium and Firefox browsers, respectively.

## 1 Introduction

The programming paradigm popularly known as object-oriented programming (OOP) is widely used for developing large and complex applications because it encapsulates the implementation details of data structures and algorithms into objects; this in turn facilitates cleaner software design, better code reuse, and easier software maintenance. Although there are many programming languages that support OOP, C++ has been the most popular, in particular when runtime performance is a key objective. For example, all major web browsers—Internet Explorer, Chrome, Firefox, and Safari are implemented in C++.

An important OOP feature is type casting that converts one object type to another. Type conversions play an important role in polymorphism. It allows a program to treat objects of one type as another so that the code can utilize certain general or specific features within the class hierarchy. Unlike other OOP languages—such as `Java`—that always verify the safety of a type conversion using runtime type information (RTTI), C++ offers two kinds of type conversions: `static_cast`, which verifies the correctness of conversion at *compile time*, and `dynamic_cast`, which verifies type safety at *runtime* using RTTI. `static_cast` is much more efficient because runtime type checking by `dynamic_cast` is an expensive operation (e.g., 90 times slower than `static_cast` on average). For this reason, many performance critical applications like web browsers, Chrome and Firefox in particular, prohibit `dynamic_cast` in their code and libraries, and strictly use `static_cast`.

However, the performance benefit of `static_cast` comes with a security risk because information at compile time is by no means sufficient to fully verify the safety of type conversions. In particular, upcasting (casting a derived class to its parent class) is always safe, but downcasting (casting a parent class to one of its derived classes) may not be safe because the derived class may not be a subobject of a truly allocated object in downcasting. Unsafe downcasting is better known as *bad-casting* or *type-confusion*.

Bad-casting has critical security implications. First, bad-casting is *undefined behavior* as specified in the C++ standard (5.2.9/11 [26]). Thus, compilers cannot guarantee the correctness of a program execution after bad-casting occurs (more detailed security implication analysis on undefined behavior is provided in §2). In addition to undefined behavior, bad-casting is similar to memory corruption vulnerabilities like stack/heap overflows and use-after-free. A bad-casted pointer violates a programmer's intended pointer semantics, and allows an attacker to corrupt memory beyond the true boundary of an object. For example, a bad-casting vulnerability in Chrome (CVE-2013-0912) was used to win the Pwn2Own 2013 competition by leaking and corrupting a security sensitive memory region [31]. More alarmingly, bad-casting is not only security-critical but is also common in applications. For example, 91 bad-casting vulnerabilities have been reported over the last four years in Chrome. Moreover, over 90% of these bad-casting bugs were rated as *security-high*, meaning that the bug can be directly exploited or indirectly used to mount arbitrary code execution attacks.

To avoid bad-casting issues, several C++ projects employ custom RTTI, which embeds code to manually keep type information at runtime and verify the type conversion safety of `static_cast`. However, only a few C++ programs are designed with custom RTTI, and supporting custom RTTI in existing programs requires heavy manual code modifications.

Another approach, as recently implemented by Google in the Undefined Behavior Sanitizer (UBSAN) [42], optimizes the performance of `dynamic_cast` and replaces all `static_cast` with `dynamic_cast`. However, this approach is limited because `dynamic_cast` only supports polymorphic classes, whereas `static_cast` is used for both polymorphic and non-polymorphic classes. Thus, this simple replacement approach changes the program semantics and results in runtime crashes when `dynamic_cast` is applied to non-polymorphic classes. It is difficult to identify whether a `static_cast` operation will be used for polymorphic or non-polymorphic classes without runtime information. For this reason, tools following this direction have to rely on manual *blacklists* (i.e., opt-out and do not check all non-polymorphic classes) to avoid runtime crashes. For example, UBSAN has to blacklist 250 classes, ten functions, and eight whole source files used for the Chromium browser [9], which is manually created by repeated trial-and-error processes. Considering the amount of code in popular C++ projects, creating such a blacklist would require massive manual engineering efforts.

In this paper, we present CAVER, a runtime bad-casting detection tool that can be seamlessly integrated with large-scale applications such as commodity browsers. It takes a program's source code as input and automatically instruments the program to verify type castings at runtime. We designed a new metadata, the Type Hierarchy Table (`THTable`) to efficiently keep track of rich type information. Unlike RTTI, `THTable` uses a disjoint metadata scheme (i.e., the reference to an object's `THTable` is stored outside the object). This allows CAVER to overcome all limitations of previous bad-casting detection techniques: it not only supports both polymorphic classes and non-polymorphic classes, but also preserves the C++ ABI and works seamlessly with legacy code. More specifically, CAVER achieves three goals:

- **Easy-to-deploy.** CAVER can be easily adopted to existing C++ programs without any manual effort. Unlike current state-of-the-art tools like UBSAN, it does not rely on manual blacklists, which are required to avoid program corruption. To demonstrate, we have integrated CAVER into two popular web browsers, Chromium and Firefox, by only modifying its build configurations.
- **Coverage.** CAVER can protect all type castings of both polymorphic and non-polymorphic classes. Compared to UBSAN, CAVER covers 241% and 199% more classes and their castings, respectively.
- **Performance.** CAVER also employs optimization techniques to further reduce runtime overheads (e.g., type-based casting analysis). Our evaluation shows that CAVER imposes up to 7.6% and 64.6% overheads for performance-intensive benchmarks on the Chromium and Firefox browsers, respectively. On the contrary, UBSAN is 13.8% slower than CAVER on the Chromium browser, and it cannot run the Firefox browser due to a runtime crash.

To summarize, we make three major contribution as follows:

- **Security analysis of bad-casting.** We analyzed the bad-casting problem and its security implications in detail, thus providing security researchers and practitioners a better understanding of this emerging attack vector.
- **Bad-casting detection tool.** We designed and implemented CAVER, a general, automated, and easy-to-deploy tool that can be applied to any C++ application to detect (and mitigate) bad-casting vulnerabilities. We have shared CAVER with the Firefox team [1] and made our source code publicly available.
- **New vulnerabilities.** While evaluating CAVER, we discovered *eleven* previously unknown bad-casting vulnerabilities in two mature and widely-used open source projects, GNU `libstdc++` and Firefox. All vulnerabilities have been reported and fixed in these projects' latest releases. We expect that integration with unit tests and fuzzing infrastructure will allow CAVER to discover more bad-casting vulnerabilities in the future.

This paper is organized as follows. §2 explains bad-casting issues and their security implications. §3 illustrates high-level ideas and usages of CAVER, §4 describes the design of CAVER. §5 describes the implementation details of CAVER, §6 evaluates various aspects of

[1] The Firefox team at Mozilla asked us to share CAVER for regression testing on bad-casting vulnerabilities.

CaVer. §7 further discusses applications and limitations of CaVer, §8 describes related work. Finally, §9 concludes the paper.

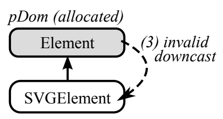
## 2 C++ Bad-casting Demystified

**Type castings in C++.** Type casting in C++ allows an object of one type to be converted to another so that the program can use different features of the class hierarchy. C++ provides four explicit casting operations: `static`, `dynamic`, `const`, and `reinterpret`. In this paper, we focus on the first two types — `static_cast` and `dynamic_cast` (5.2.9 and 5.2.7 in ISO/IEC N3690 [26]) — because they can perform downcasting and result in bad-casting. `static_cast` and `dynamic_cast` have a variety of different usages and subtle issues, but for the purpose of this paper, the following two distinctive properties are the most important: (1) time of verification: as the name of each casting operation implies, the correctness of a type conversion is checked (statically) at compile time for `static_cast`, and (dynamically) at runtime for `dynamic_cast`; (2) runtime support (RTTI): to verify type checking at runtime, `dynamic_cast` requires runtime support, called RTTI, that provides type information of the polymorphic objects.

Example 1 illustrates typical usage of both casting operations and their correctness and safety: (1) casting from a derived class (`pCanvas` of `SVGElement`) to a parent class (`pEle` of `Element`) is valid *upcasting*; (2) casting from the parent class (`pEle` of `Element`) to the original allocated class (`pCanvasAgain` of `SVGElement`) is valid *downcasting*; (3) on the other hand, the casting from an object allocated as a base class (`pDom` of `Element`) to a derived class (`p` of `SVGElement`) is invalid *downcasting* (i.e., a bad-casting); (4) memory access via the invalid pointer (`p->m_className`) can cause memory corruption, and more critically, compilers cannot guarantee any correctness of program execution after this incorrect conversion, resulting in *undefined behavior*; and (5) by using `dynamic_cast`, programmers can check the correctness of type casting at runtime, that is, since an object allocated as a base class (`pDom` of `Element`) cannot be converted to its derived class (`SVGElement`), `dynamic_cast` will return a `NULL` pointer and the error-checking code (line 18) can catch this bug, thus avoiding memory corruption.

**Type castings in practice.** Although `dynamic_cast` can guarantee the correctness of type casting, it is an expensive operation because parsing RTTI involves recursive data structure traversal and linear string comparison. From our preliminary evaluation, `dynamic_cast` is, on average, 90 times slower than `static_cast` on average. For large applications such as the Chrome browser, such performance overhead is not acceptable: simply launching Chrome incurs over 150,000 casts. Therefore, despite its security benefit, the use of `dynamic_cast` is strictly

```
1  class SVGElement: public Element { ... };
2
3  Element *pDom = new Element();
4  SVGElement *pCanvas = new SVGElement();
5
6  // (1) valid upcast from pCanvas to pEle
7  Element *pEle = static_cast<Element*>(pCanvas);
8  // (2) valid downcast from pEle to pCanvasAgain (== pCanvas)
9  SVGElement *pCanvasAgain = static_cast<SVGElement*>(pEle);
10
11 // (3) invalid downcast (-> undefined behavior)
12 SVGElement *p = static_cast<SVGElement*>(pDom);
13 // (4) leads to memory corruption
14 p->m_className = "my-canvas";
15
16 // (5) invalid downcast with dynamic_cast, but no corruption
17 SVGElement *p = dynamic_cast<SVGElement*>(pDom);
18 if (p) {
19     p->m_className = "my-canvas";
20 }
```



**Example 1:** Code example using `static_cast` to convert types of object pointers (e.g., `Element` ↔ `SVGElement` classes). (1) is valid *upcast* and (2) is valid *downcast*. (3) is an invalid *downcast*. (4) Memory access via the invalid pointer result in memory corruption; more critically, compilers cannot guarantee the correctness of program execution after this incorrect conversion, resulting in *undefined behavior*. (5) Using `dynamic_cast`, on the other hand, the program can check the correctness of *downcast* by checking the returned pointer.

forbidden in Chrome development.

A typical workaround is to implement custom RTTI support. For example, most classes in WebKit-based browsers have an `isType()` method (e.g., `isSVGElement()`), which indicates the true allocated type of an object. Having this support, programmers can decouple a `dynamic_cast` into an explicit type check, followed by `static_cast`. For example, to prevent the bad-casting (line 12) in Example 1, the program could invoke the `isSVGElement()` method to check the validity of casting. However, this sort of type tracking and verification has to be manually implemented, and thus supporting custom RTTI in existing complex programs is a challenging problem. Moreover, due to the error-prone nature of manual modifications (e.g., incorrectly marking the object identity flag, forgetting to check the identity using `isType()` function, etc.), bad-casting bugs still occur despite custom RTTI [41].

**Security implications of bad-casting.** The C++ standard (5.2.9/11 [26]) clearly specifies that the behavior of an application becomes *undefined* after an incorrect `static_cast`. Because undefined behavior is an enigmatic issue, understanding the security implications and exploitability of bad-casting requires deep knowledge of common compiler implementations.

Generally, bad-casting vulnerabilities can be exploited via several means. An incorrectly casted pointer will either have wider code-wise visibility (e.g., allowing out-of-bound memory accesses), or become incorrectly adjusted (e.g., corrupting memory semantics because of misalignment). For example, when bad-casting occurs in proximity to a virtual function table pointer (`vptr`), an attacker can directly control input to the member variable (e.g., by employing abusive memory allocation techniques such as heap-spray techniques [16, 38]), overwrite the `vptr` and hijack the control flow. Similarly, an attacker can also exploit bad-casting vulnerabilities to launch non-control-data attacks [7].

The exploitability of a bad-casting bug depends on whether it allows attackers to perform out-of-bound memory access or manipulate memory semantics. This in turn relies on the details of object data layout as specified by the C++ application binary interface (ABI). Because the C++ ABI varies depending on the platform (e.g., Itanium C++ ABI [12] for Linux-based platforms and Microsoft C++ ABI [11] for Windows platforms), security implications for the same bad-casting bug can be different. For example, bad-casting may not crash, corrupt, or alter the behavior of an application built against the Itanium C++ ABI because the base pointer of both the base class and derived class always point to the same location of the object under this ABI. However, the same bad-casting bug can have severe security implications for other ABI implementations that locate a base pointer of a derived class differently from that of a base class, such as HP and legacy g++ C++ ABI [13]. In short, given the number of different compilers and the various architectures supported today, we want to highlight that bad-casting should be considered as a serious security issue. This argument is also validated from recent correspondence with the Firefox security team: after we reported two new bad-casting vulnerabilities in Firefox [4], they also pointed out the C++ ABI compatibility issue and rated the vulnerability as security-high.

**Running example: CVE-2013-0912.** Our illustrative Example 1 is extracted from a real-world bad-casting vulnerability—CVE-2013-0912, which was used to exploit the Chrome web browser in the Pwn2Own 2013 competition. However, the complete vulnerability is more complicated as it involves a multitude of castings (between siblings and parents).

In HTML5, an SVG image can be embedded directly into an HTML page using the `<svg>` tag. This tag is implemented using the `SVGElement` class, which inherits from the `Element` class. At the same time, if a web page happens to contain unknown tags (any tags other than standard), an object of the `HTMLUnknownElement` class will be created to represent this unknown tag. Since both tags are valid HTML elements, objects of these types can be safely casted to the `Element` class. Bad-casting occurs when the browser needs to render an SVG image. Given an `Element` object, it tries to downcast the object to `SVGElement` so the caller function can invoke member functions of the `SVGElement` class. Unfortunately, since not all `Element` objects are initially allocated as `SVGElement` objects, this

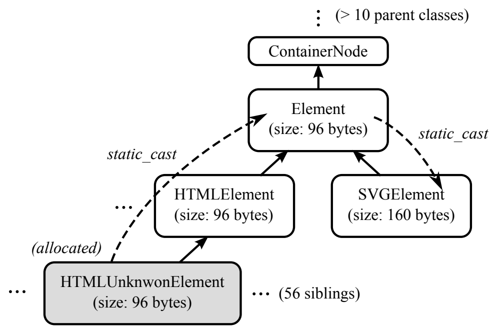


**Figure 1:** Inheritance hierarchy of classes involved in the CVE-2013-0912 vulnerability. MWR Labs exploited this vulnerability to hijack the Chrome browser in the Pwn2Own 2013 competition [31]. The object is allocated as `HTMLUnknownElement` and eventually converted (`static_cast`) to `SVGElement`. After this incorrect type casting, accessing member variables via this object pointer will cause memory corruption.

`static_cast` is not always valid. In the exploit demonstrated in the Pwn2Own 2013 competition [31], attackers used an object allocated as `HTMLUnknownElement`. As the size of an `SVGElement` object (160 bytes) is much larger than an `HTMLUnknownElement` object (96 bytes), this incorrectly casted object pointer allowed the attackers to access memory beyond the real boundary of the allocated `HTMLUnknownElement` object. They then used this capability to corrupt the `vtable` pointer of the object adjacent to the `HTMLUnknownElement` object, ultimately leading to a control-flow hijack of the Chrome browser. This example also demonstrates why identifying bad-casting vulnerabilities is not trivial for real-world applications. As shown in Figure 1, the `HTMLUnknownElement` class has more than 56 siblings and the `Element` class has more than 10 parent classes in WebKit. Furthermore, allocation and casting locations are far apart within the source code. Such complicated class hierarchies and disconnections between allocation and casting sites make it difficult for developers and static analysis techniques to reason about the true allocation types (i.e., alias analysis).

## 3 CAVER Overview

In this paper, we focus on the correctness and effectiveness of CAVER against bad-casting bugs, and our main application scenario is as a back-end testing tool for detecting bad-casting bugs. CAVER's workflow (Figure 2) is as simple as compiling a program with one extra compile and link flag (i.e., `-fcaver` for both). The produced binary becomes capable of verifying the correctness of every type conversion at runtime. When CAVER detects an incorrect type cast, it provides detailed information of the bad-cast: the source class, the destination class, the truly allocated class, and call stacks at the time the bad-cast is captured. Figure 3 shows a snippet of the actual report of CVE-2013-0912. Our bug report experience showed that the report generated by CAVER helped upstream main-

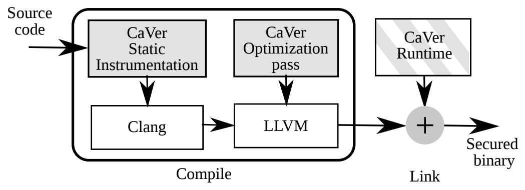


**Figure 2:** Overview of CAVER's design and workflow. Given the source code of a program, CAVER instruments possible castings at compile time, and injects CAVER's runtime to verify castings when they are performed.

```
== CaVer : (Stopped) A bad-casting detected
@SVGViewSpec.cpp:87:12
  Casting an object of 'blink::HTMLUnknownElement'
    from 'blink::Element
    to   'blink::SVGElement'
         Pointer        0x60c000008280
         Alloc base     0x60c000008280
         Offset         0x000000000000
         THTable        0x7f7963aa20d0

#1 0x7f795d76f1a4 in viewTarget SVGViewSpec.cpp:87
#2 0x7f795d939d1c in viewTargetAttribute V8SVGViewSpec.cpp:56
...
```

**Figure 3:** A report that CAVER generated on CVE-2013-0912.

tainers easily understand, confirm, and fix eleven newly discovered vulnerabilities without further examination.

## 4 Design

In this section, we introduce the design of CAVER. We first describe how the THTable is designed to generally represent the type information for both polymorphic and non-polymorphic classes (§4.1), and then explain how CAVER associates the THTable with runtime objects (§4.2). Next, we describe how CAVER verifies the correctness of type castings (§4.3). At the end of this section, we present optimization techniques used to reduce the runtime overhead of CAVER (§4.4).

### 4.1 Type Hierarchy Table

To keep track of the type information required for validating type casting, CAVER incorporates a new metadata structure, called the *Type Hierarchy Table* (THTable). Given a pointer to an object allocated as type T, the THTable contains the set of all possible types to which T can be casted. In C++, these possible types are a product of two kinds of class relationships: *is-a* and *has-a*. The *is-a* relationship between two objects is implemented as class inheritance, the *has-a* relationship is implemented as class composition (i.e., having a member variable in a class). Thus, for each class in a C++ program, CAVER creates a corresponding THTable that includes information about both relationships.

To represent class inheritance, the THTable employs two unique design decisions. First, information on inherited classes (i.e., base classes) is unrolled and serialized. This allows CAVER to efficiently scan through a set of base classes at runtime while standard RTTI requires recursive traversal. Second, unlike RTTI, which stores a mangled class name, the THTable stores the hash value of a class name. This allows CAVER to avoid expensive string equality comparisons. Note, since all class names are available to CAVER at compile time, all possible hash collisions can be detected and resolved to avoid false negatives during runtime. Moreover, because casting is only allowed between classes within the same inheritance chain, we only need to guarantee the uniqueness of hash values within a set of those classes, as opposed to guaranteeing global uniqueness.

The THTable also includes information of whether a base class is a phantom class, which cannot be represented based on RTTI and causes many false alarms in RTTI-based type verification solutions [9]. We say a class P is a phantom class of a class Q if two conditions are met: (1) Q is directly or indirectly derived from P; and (2) compared to P, Q does not have additional member variables or different virtual functions. In other words, they have the same data layout. Strictly speaking, allocating an object as P and downcasting it to Q is considered bad-casting as Q is not a base class of P. However, such bad-castings are harmless from a security standpoint, as the pointer semantic after downcasting is the same. More importantly, phantom classes are often used in practice to implement object relationships with empty inheritances. For these reasons, CAVER deliberately allows bad-castings caused by phantom classes. This is done by reserving a one bit space in the THTable for each base class, and marking if the base class is a phantom class. We will describe more details on how the phantom class information is actually leveraged in §4.3.

In addition, the THTable contains information on composited class(es) to generally represent the type information for both polymorphic and non-polymorphic classes and overcome the limitation of RTTI-based type verification solutions. RTTI-based solutions locate a RTTI reference via the virtual function table (VTable). However, since only polymorphic classes have VTable, these solutions can cause runtime crashes when they try to locate the VTable for non-polymorphic classes. Unlike RTTI, CAVER binds THTable references to the allocated object with external metadata (refer §4.2 for details). Therefore, CAVER not only supports non-polymorphic objects, but it also does not break the C++ ABI. However, composited class(es) now share the same THTable with their container class. Since a composited class can also have its own inheritances and compositions, we do not unroll information about composited class(es); instead, CAVER provides a reference to the composited class's THTable. The THTable also stores the layout information (offset and size) of each composited class to determine whether the given pointer points to a certain composited class.

Other than inheritance and composition information as described above, the THTable contains basic information on the corresponding type itself: a type size to represent object ranges; and a type name to generate user-friendly bad-casting reports.

## 4.2 Object Type Binding

To verify the correctness of type casting, CAVER needs to know the actual allocated type of the object to be casted. In CAVER, we encoded this type information in the THTable. In this subsection, we describe how CAVER binds the THTable to each allocated object. To overcome the limitations of RTTI-based solutions, CAVER uses a disjoint metadata scheme (i.e., the reference to an object's THTable is stored outside the object). With this unique metadata management scheme, CAVER not only supports both polymorphic classes and non-polymorphic classes, but also preserves the C++ ABI and works seamlessly with legacy code. Overall, type binding is done in two steps. First, CAVER instruments each allocation site of an application to pass the allocation metadata to its runtime library. Second, CAVER's runtime library maintains the allocation metadata and supports efficient lookup operations.

**Instrumentation.** The goal of the instrumentation is to pass all information of an allocated object to the runtime library. To bind a THTable to an object, the runtime library needs two pieces of information: a reference to the THTable and the base address of the allocated object.

In C++, objects can be allocated in three ways: in heap, on stack, or as global objects. In all three cases, the type information of the allocated object can be determined statically at compile time. This is possible because C++ requires programmers to specify the object's type at its allocation site, so the corresponding constructor can be invoked to initialize memory. For global and stack objects, types are specified before variable names; and for heap objects, types are specified after the new operator. Therefore, CAVER can obtain type information by statically inspecting the allocation site at compile time. Specifically, CAVER generates the THTable (or reuses the corresponding THTable if already generated) and passes the reference of the THTable to the runtime library. An example on how CAVER instruments a program is shown in Example 2.

For heap objects, CAVER inserts one extra function invocation (trace_heap() in Example 2) to the runtime library after each new operator, and passes the information of the object allocated by new; a reference to the THTable and the base address of an object. A special case for the new operator is an array allocation, where a set of objects of the same type are allocated. To handle this case, we add an extra parameter to inform the runtime library on how many objects are allocated together at the base address.

Unlike heap objects, stack objects are implicitly al-

```
  // Heap objects (dynamically allocated)
  void func_heap_ex() {
    C *p_heap_var = new C;
    C *p_heap_array = new C[num_heap_array];
+   trace_heap(&THTable(C), p_heap_var, 1);
+   trace_heap(&THTable(C), p_heap_array, num_heap_array);
    ...
  }

  // Stack objects
  void func_stack_ex() {
    C stack_var;
+   trace_stack_begin(&THTable(C), &stack_var, 1);
    ...
+   trace_stack_end(&stack_var);
  }

  // Global objects
  C global_var;

  // @.ctors: (invoked at the program's initialization)
  //   trace_global_helper_1() and trace_global_helper_2()
+ void trace_global_helper_1() {
+   trace_global(&THTable(C), &global_var, 1);
+ }

   // Verifying the correctness of a static casting
  void func_verify_ex() {
    B *afterAddr = static_cast<A>(beforeAddr);
+   verify_cast(beforeAddr, afterAddr, type_hash(A));
  }
```

**Example 2:** An example of how CAVER instruments a program. Lines marked with + represent code introduced by CAVER, and &THTable(T) denotes the reference to the THTable of class T. In this example, we assume that the THTable of each allocated class has already been generated by CAVER.

located and freed. To soundly trace them, CAVER inserts two function calls for each stack object at the function prologue and epilogue (trace_stack_begin() and trace_stack_end() in Example 2), and passes the same information of the object as is done for heap objects. A particular challenge is that, besides function returns, a stack unwinding can also happen due to exceptions and setjmp/longjmp. To handle these cases, CAVER leverages existing compiler functionality (e.g., EHScopeStack::Cleanup in clang) to guarantee that the runtime library is always invoked once the execution context leaves the given function scope.

To pass information of global objects to the runtime library, we leverage existing program initialization procedures. In ELF file format files [46], there is a special section called .ctors, which holds constructors that must be invoked during an early initialization of a program. Thus, for each global object, CAVER creates a helper function (trace_global_helper_1() in Example 2) that invokes the runtime library with static metadata (the reference to the THTable) and dynamic metadata (the base address and the number of array elements). Then, CAVER adds the pointer to this helper function to the .ctors section so that the metadata can be conveyed to the runtime library[2].

[2]Although the design detail involving .ctors section is platform dependent, the idea of registering the helper function into the initialization

**Runtime library.** The runtime library of CAVER maintains all the metadata (`THTable` and base address of an object) passed from tracing functions during the course of an application execution. Overall, we consider two primary requirements when organizing the metadata. First, the data structure must support range queries ( i.e., given a pointer pointing to an address within an object ([base, base+size)) CAVER should be able to find the corresponding `THTable` of the object). This is necessary because the object pointer does not always point to the allocation base. For example, the pointer to be casted can point to a composited object. In case of multi-inheritance, the pointer can also point to one of the parent classes. Second, the data structure must support efficient store and retrieve operations. CAVER needs to store the metadata for every allocation and retrieve the metadata for each casting verification. As the number of object allocations and type conversions can be huge (see §6), these operations can easily become the performance bottleneck.

We tackle these challenges using a hybrid solution (see Appendix 2 for the algorithm on runtime library functions). We use red-black trees to trace global and stack objects and an alignment-based direct mapping scheme to trace heap objects[3].

We chose red-black trees for stack and global objects for two reasons. First, tree-like data structures are well known for supporting efficient range queries. Unlike hash-table-based data structures, tree-based data structures arrange nodes according to the order of their keys, whose values can be numerical ranges. Since nodes are already sorted, a balanced tree structure can guarantee $O(logN)$ complexity for range queries while hash-table-based data structure requires $O(N)$ complexity. Second, we specifically chose red-black trees because there are significantly more search operations than update operations (i.e., more type conversion operations than allocations, see §6), thus red-black trees can excel in performance due to self-balancing.

In CAVER, each node of a red-black tree holds the following metadata: the base address and the allocation size as the key of the node, and the `THTable` reference as the value of the node.

For global object allocations, metadata is inserted into the global red-black tree when the object is allocated at runtime, with the key as the base address and the allocation size[4], and the value as the address of the `THTable`. We maintain a per-process global red-black tree without locking mechanisms because there are no data races on the global red-black tree in CAVER. All updates on the global red-black tree occur during early process start-up (i.e., before executing any user-written code) and update orders are well serialized as listed in the .ctors section.

For stack object allocations, metadata is inserted to the stack red-black tree similar to the global object case. Unlike a global object, we maintain a *per-thread* red-black tree for stack objects to avoid data races in multi-threaded applications. Because a stack region (and all operations onto this region) are exclusive to the corresponding thread's execution context, this per-thread data structure is sufficient to avoid data races without locks.

For heap objects, we found that red-black trees are not a good design choice, especially for multi-threaded programs. Different threads in the target programs can update the tree simultaneously, and using locks to avoid data races resulted in high performance overhead, as data contention occured too frequently. Per-thread red-black trees used for stack objects are not appropriate either, because heap objects can be shared by multiple threads. Therefore, we chose to use a custom memory allocator that can support alignment-based direct mapping schemes [3, 22]. In this scheme, the metadata can be maintained for a particular object, and can be retrieved with $O(1)$ complexity on the pointer pointing to anywhere within the object's range.

## 4.3 Casting Safety Verification

This subsection describes how CAVER uses traced information to verify the safety of type casting. We first describe how the instrumentation is done at compile time, and then describe how the runtime library eventually verifies castings during runtime.

**Instrumentation.** CAVER instruments `static_cast` to invoke a runtime library function, `verify_cast()`, to verify the casting. Here, CAVER analyzes a type hierarchy involving source and destination types in `static_cast` and only instruments for downcast cases. When invoking `verify_cast()`, CAVER passes the following three pieces of information: `beforeAddr`, the pointer address before the casting; `afterAddr`, the pointer address after the casting; and `TargetTypeHash`, the hash value of the destination class to be casted to (denoted as `type_hash(A)` in Example 2).

**Runtime library.** The casting verification (Appendix 1) is done in two steps: (1) locating the corresponding `THTable` associated with the object pointed to by `beforeAddr`; and (2) verifying the casting operation by checking whether `TargetTypeHash` is a valid type where `afterAddr` points.

To locate the corresponding `THTable`, we first check the data storage membership because we do not know how the object `beforeAddr` points to is allocated. Checks

function list can be generalized for other platforms as others also support .ctors-like features

[3]The alignment-based direct mapping scheme can be applied for global and stack objects as well, but this is not implemented in the current version. More details can be found in §7.

[4]The allocation size is computed by multiplying the type size represented in `THTable` and the number of array elements passed during runtime.

are ordered by their expense, and the order is critical for good performance. First, a stack object membership is checked by determinig whether the beforeAddr is in the range between the stack top and bottom; then, a heap object membership is checked by whether the beforeAddr is in the range of pre-mapped address spaces reserved for the custom allocator; finally a global object membership is checked with a bit vector array for each loaded binary module. After identifying the data storage membership, CaVer retrieves the metadata containing the allocation base and the reference to the THTable. For stack and global objects, the corresponding red-black tree is searched. For heap objects, the metadata is retrieved from the custom heap.

Next, CaVer verifies the casting operation. Because the THTable includes all possible types that the given object can be casted to (i.e., all types from both inheritances and compositions), CaVer exhaustively matches whether TargetTypeHash is a valid type where afterAddr points. To be more precise, the afterAddr value is adjusted for each matching type. Moreover, to avoid false positives due to a phantom class, CaVer tries to match all phantom classes of the class to be casted to.

### 4.4 Optimization

Since performance overhead is an important factor for adoption, CaVer applies several optimization techniques. These techniques are applied in two stages, as shown in Figure 2. First, offline optimizations are applied to remove redundant instrumentations. After that, additional runtime optimizations are applied to further reduce the performance overhead.

**Safe-allocations.** Clearly, not all allocated objects will be involved in type casting. This implicates that CaVer does not need to trace type information for objects that would never be casted. In general, soundly and accurately determining whether objects allocated at a given allocation site will be casted is a challenging problem because it requires sophisticated static points-to analysis. Instead, CaVer takes a simple, yet effective, optimization approach inspired from C type safety checks in CCured [33]. The key idea is that the following two properties always hold for downcasting operations: (1) bad-casting may happen only if an object is allocated as a child of the source type or the source type itself; and (2) bad-casting never happens if an object is allocated as the destination type itself or a child of the destination type. This is because static_cast guarantees that the corresponding object must be a derived type of the source type. Since CaVer can observe all allocation sites and downcasting operations during compilation, it can recursively apply the above properties to identify *safe-allocation* sites, i.e., the allocated objects will never cause bad-casting.

**Caching verification results.** Because casting verification involves loops (over the number of compositions and the number of bases) and recursive checks (in a composition case), it can be a performance bottleneck. A key observation here is that the verification result is always the same for the same allocation type and the same target type (i.e., when the type of object pointed by afterAddr and TargetTypeHash are the same). Thus, in order to alleviate this potential bottleneck, we maintain a cache for verification results, which is inspired by UBSan [42]. First, a verification result is represented as a concatenation of the address of a corresponding THTable, the offset of the afterAddr within the object, and the hash value of target type to be casted into (i.e., &THTable || offset || TargetTypeHash). Next, this concatenated value is checked for existence in the cache before verify_cast() actually performs verification. If it does, verify_cast() can conclude that this casting is correct. Otherwise, verify_cast() performs actual verification using the THTable, and updates the cache only if the casting is verified to be correct.

## 5 Implementation

We implemented CaVer based on the LLVM Compiler project [43] (revision 212782, version 3.5.0). The static instrumentation module is implemented in Clang's CodeGen module and LLVM's Instrumentation module. The runtime library is implemented using the compiler-rt module based on LLVM's Sanitizer code base. In total, CaVer is implemented in 3,540 lines of C++ code (excluding empty lines and comments).

CaVer is currently implemented for the Linux x86 platform, and there are a few platform-dependent mechanisms. For example, the type and tracing functions for global objects are placed in the .ctors section of ELF. As these platform-dependent features can also be found in other platforms, we believe CaVer can be ported to other platforms as well. CaVer interposes threading functions to maintain thread contexts and hold a per-thread red-black tree for stack objects. CaVer also maintains the top and bottom addresses of stack segments to efficiently check pointer membership on the stack. We also modified the front-end drivers of Clang so that users of CaVer can easily build and secure their target applications with one extra compilation flag and linker flag, respectively.

## 6 Evaluation

We evaluated CaVer with two popular web browsers, Chromium [40] (revision 295873) and Firefox [44] (revision 213433), and two benchmarks from SPEC CPU2006 [39][5]. Our evaluation aims to answer the following questions:

[5] Although CaVer was able to correctly run all C++ benchmarks in SPEC CPU2006, only 483.xalancbmk and 450.soplex have downcast operations.

- How easy is it to deploy CAVER to applications? (§6.1)
- What are the new vulnerabilities CAVER found? (§6.2)
- How precise is CAVER's approach in detecting bad-casting vulnerabilities? (§6.3)
- How good is CAVER's protection coverage? (§6.4)
- What are the instrumentation overheads that CAVER imposes and how many type castings are verified by CAVER? (§6.5)
- What are the runtime performance overheads that CAVER imposes? (§6.6)

**Comparison methods.** We used UBSAN, the state-of-art tool for detecting bad-casting bugs, as our comparison target of CAVER. Also, We used CAVER-NAIVE, which disabled the two optimization techniques described in §4.4, to show their effectiveness on runtime performance optimization.

**Experimental setup.** All experiments were run on Ubuntu 13.10 (Linux Kernel 3.11) with a quad-core 3.40 GHz CPU (Intel Xeon E3-1245), 16 GB RAM, and 1 TB SSD-based storage.

## 6.1 Deployments

As the main design goal for CAVER is automatic deployments, we describe our experience of applying CAVER to tested programs including SPEC CPU 2006 benchmarks, the Chromium browser, and the Firefox browser. CAVER was able to successfully build and run these programs without any program-specific understanding of the code base. In particular, we added one line to the build configuration file to build SPEC CPU 2006, 21 lines to the `.gyp` build configuration to build the Chromium browser, and 10 lines to the `.mozconfig` build configuration file to build the Firefox browser. Most of these build configuration changes were related to replacing `gcc` with `clang`.

On the contrary, UBSAN crashed while running xalancbmk in SPEC CPU 2006 and while running the Firefox browser due to checks on non-polymorphic classes. UBSAN also crashed the Chromium browser without blacklists, but was able to run once we applied the blacklists provided by the Chromium project [9]. In particular, to run Chromium, the blacklist has 32 different rules that account for 250 classes, ten functions, and eight whole source files. Moreover, this blacklist has to be maintained constantly as newly introduced code causes new crashes in UBSAN [10]. This is a practical obstacle for adopting UBSAN in other C++ projects—although UBSAN has been open sourced for some time, Chromium remains the only major project that uses UBSAN, because there is a dedicated team to maintain its blacklist.

## 6.2 Newly Discovered Bad-casting Vulnerabilities

To evaluate CAVER's capability of detecting bad-casting bugs, we ran CAVER-hardened Chromium and Firefox with their regression tests (mostly checking functional correctness). During this evaluation, CAVER found eleven previously unknown bad-casting vulnerabilities in GNU `libstdc++` while evaluating Chromium and Firefox. Table 1 summarizes these vulnerabilities including related class information: allocated type, source, and destination types in each bad-casting. In addition, we further analyzed their security impacts: potential compatibility problems due to the C++ ABI (see §2) or direct memory corruption, along with security ratings provided by Mozilla for Firefox.

CAVER found two vulnerabilities in the Firefox browser. The Firefox team at Mozilla confirmed and fixed these, and rated both as *security-high*, meaning that the vulnerability can be abused to trigger memory corruption issues. These two bugs were casting the pointer into a class which is not a base class of the originally allocated type. More alarmingly, there were type semantic mismatches after the bad-castings—subsequent code could dereference the incorrectly casted pointer. Thus the C++ ABI and Memory columns are checked for these two cases.

CAVER also found nine bugs in GNU `libstdc++` while running the Chromium browser. We reported these bugs to the upstream maintainers, and they have been confirmed and fixed. Most of these bugs were triggered when `libstdc++` converted the type of an object pointing to its composite objects (e.g., `Base_Ptr` in `libstdc++`) into a more derived class (`Rb_Tree_node` in `libstdc++`), but these derived classes were not base classes of what was originally allocated (e.g., `EncodedDescriptorDatabase` in Chromium). Since these are generic bugs, meaning that benign C++ applications will encounter these issues even if they correctly use `libstdc++` or related libraries, it is difficult to directly evaluate their security impacts without further evaluating the applications themselves.

These vulnerabilities were identified with legitimate functional test cases. Thus, we believe CAVER has great potential to find more vulnerabilities once it is utilized for more applications and test cases, as well as integrated with fuzzing infrastructures like ClusterFuzz [2] for Chromium.

## 6.3 Effectiveness of Bad-casting Detection

To evaluate the correctness of detecting bad-casting vulnerabilities, we tested five bad-casting exploits of Chromium on the CAVER-hardened Chromium binary (see Table 2). We backported five bad-casting vulnerabilities as unit tests while preserving important features that may affect CAVER's detection algorithm, such as class inheritances and their compositions, and allocation

| Product | Bug ID | Vulnerable | Types | Security Implication | | |
|---|---|---|---|---|---|---|
| | | Function | Allocation / Source / Destination | ABI | Mem | Rating |
| Firefox | 1074280 [4] | PaintLayer() | BasicThebesLayer / Layer / BasicContainerLayer | ✓ | ✓ | High (CVE-2014-1594) |
| Firefox | 1089438 [5] | EvictContent() | PRCListStr / PRCList / nsSHistory | ✓ | ✓ | High |
| `libstdc++` | 63345 [30] | _M_const_cast() | EncodedDescriptorDatabase / Base_Ptr / Rb_Tree_node | ✓ | - | † |
| `libstdc++` | 63345 [30] | _M_end() | EnumValueOptions / Rb_tree_node_base / Link_type | ✓ | - | † |
| `libstdc++` | 63345 [30] | _M_end() const | GeneratorContextImpl / Rb_tree_node_base / Link_type_const | ✓ | - | † |
| `libstdc++` | 63345 [30] | _M_insert_unique() | WaitableEventKernel / Base_ptr / List_type | ✓ | - | † |
| `libstdc++` | 63345 [30] | operator*() | BucketRanges / List_node_base / Node | ✓ | - | † |
| `libstdc++` | 63345 [30] | begin() | FileOptions / Link_type / Rb_Tree_node | ✓ | - | † |
| `libstdc++` | 63345 [30] | begin() const | std::map / Link_type / Rb_Tree_node | ✓ | - | † |
| `libstdc++` | 63345 [30] | end() | MessageOptions / Link_type / Rb_Tree_node | ✓ | - | † |
| `libstdc++` | 63345 [30] | end() const | Importer / Link_type / Rb_Tree_node | ✓ | - | † |

**Table 1:** A list of vulnerabilities newly discovered by CAVER. All security vulnerabilities listed here are confirmed, and already fixed by the corresponding development teams. Columns under Types represent classes causing bad-castings: allocation, source and destination classes. Columns under Security Implication represents the security impacts of each vulnerability: whether the vulnerability has `C++` ABI incompatibility issues (ABI); whether the vulnerability triggers memory corruption (Mem); and the actual security assessment ratings assigned by the vendor (Rating). †: The GNU `libstdc++` members did not provide security ratings.

size. This backporting was due to the limited support for the `LLVM/clang` compiler by older Chromium (other than CVE-2013-0912). Table 2 shows our testing results on these five known bad-casting vulnerabilities. CAVER successfully detected all vulnerabilities.

In addition to real vulnerabilities, we thoroughly evaluated CAVER with test cases that we designed based on all possible combinations of bad-casting vulnerabilities: (1) whether an object is polymorphic or non-polymorphic; and (2) the three object types: allocation, source, and destination.

$$|\{\text{Poly, non-Poly}\}|^{|\{\text{Alloc, From, To}\}|} = 8$$

Eight different unit tests were developed and evaluated as shown in Table 3. Since CAVER's design generally handles both polymorphic and non-polymorphic classes, CAVER successfully detected all cases. For comparison, UBSAN failed six cases mainly due to its dependency on RTTI. More severely, among the failed cases, UBSAN crashed for two cases when it tried to parse RTTI non-polymorphic class objects, showing it is difficult to use without manual blacklists. Considering Firefox contains greater than 60,000 downcasts, (see Table 4), creating such a blacklist for Firefox would require massive manual engineering efforts.

## 6.4 Protection Coverage

Table 4 summarizes our evaluation of CAVER's protection coverage during instrumentation, including the number of protected types/classes (the left half), and the number of protected type castings (the right half). In our evaluation with `C++` applications in SPEC CPU 2006, Firefox, and Chromium, CAVER covers 241% more types than UBSAN; and protects 199% more type castings.

## 6.5 Instrumentation Overheads

There are several sources that increase a program's binary size (see Table 5), including (1) the inserted functions for tracing objects' type and verifying type castings, (2) the `THTable` of each class, and (3) CAVER's runtime library. Although CAVER did not perform much instrumentation for most SPEC CPU 2006 applications, the file size increase still was noticeable. This increase was caused by the statically linked runtime library (245 KB). The CAVER-hardened Chromium requires 6× more storage compared to Firefox because the Chromium code bases contains more classes than Firefox. The additional `THTable` overhead is the dominant source of file size increases. (see Table 4). For comparison, UBSAN increased the file size by 64% and 49% for Chromium and Firefox, respectively; which indicates that `THTable` is an efficient representation of type information compared to RTTI.

| Name | # of tables | | # of verified cast | |
|---|---|---|---|---|
| | RTTI | `THTable` | UBSAN | CAVER |
| 483.xalancbmk | 881 | 3,402 | 1,378 | 1,967 |
| 450.soplex | 39 | 227 | 0 | 2 |
| Chromium | 24,929 | 94,386 | 11,531 | 15,611 |
| Firefox | 9,907 | 30,303 | 11,596 | 71,930 |

**Table 4:** Comparisons of protection coverage between UBSAN and CAVER. In the # of tables column, `VTable` shows the number of virtual function tables and `THTable` shows the number of type hierarchy tables, each of which is generated to build the program. # of verified cast shows the number `static_cast` instrumented in UBSAN and CAVER, respectively. Overall, CAVER covers 241% and 199% more classes and their castings, respectively, compared to UBSAN.

| Name | File Size (KB) | | | | |
|---|---|---|---|---|---|
| | Orig. | UBSAN | | CAVER | |
| 483.xalancbmk | 6,111 | 6,674 | 9% | 7,169 | 17% |
| 450.soplex | 466 | 817 | 75% | 861 | 84% |
| Chromium | 249,790 | 612,321 | 145% | 453,449 | 81% |
| Firefox | 242,704 | 395,311 | 62% | 274,254 | 13% |

**Table 5:** The file size increase of instrumented binaries: CAVER incurs 64% and 49% less storage overheads in Chromium and Firefox browsers, compared to UBSAN.

| CVE # | Bug ID | Type Names | | | Security Rating | Mitigated |
|---|---|---|---|---|---|---|
| | | Allocation | Source | Destination | | by CAVER |
| CVE-2013-0912 | 180763 | HTMLUnknownElement | Element | SVGElement | High | ✓ |
| CVE-2013-2931 | 302810 | MessageEvent | Event | LocatedEvent | High | ✓ |
| CVE-2014-1731 | 349903 | RenderListBox | RenderBlockFlow | RenderMeter | High | ✓ |
| CVE-2014-3175 | 387016 | SpeechSynthesis | EventTarget | SpeechSynthesisUtterance | High | ✓ |
| CVE-2014-3175 | 387371 | ThrobAnimation | Animation | MultiAnimation | Medium | ✓ |

**Table 2:** Security evaluations of CAVER with known vulnerabilities of the Chromium browser. We first picked five known bad-casting bugs and wrote test cases for each vulnerability, retaining features that may affect CAVER's detection algorithm, including class hierarchy and their compositions, and related classes including allocation, source, and destination types). CAVER correctly detected all vulnerabilities.

(a) CAVER, P Alloc

| From \ To | P | Non-P |
|---|---|---|
| P | ✓ | ✓ |
| Non-P | ✓ | ✓ |

(b) CAVER, Non-P Alloc

| From \ To | P | Non-P |
|---|---|---|
| P | ✓ | ✓ |
| Non-P | ✓ | ✓ |

(c) UBSAN, P Alloc

| From \ To | P | Non-P |
|---|---|---|
| P | ✓ | X |
| Non-P | ✓ | X |

(d) UBSAN, Non-P Alloc

| From \ To | P | Non-P |
|---|---|---|
| P | Crash | X |
| Non-P | Crash | X |

**Table 3:** Evaluation of protection coverage against all possible combinations of bad-castings. P and Non-P mean polymorphic and non-polymorphic classes, respectively. In each cell, ✓marks a successful detection, X marks a failure, and Crash marks the program crashed. (a) and (b) show the results of CAVER with polymorphic class allocations and non-polymorphic class allocations, respectively, and (c) and (d) show the cases of UBSAN. CAVER correctly detected all cases, while UBSAN failed for 6 cases including 2 crashes.

## 6.6 Runtime Performance Overheads

In this subsection, we measured the runtime overheads of CAVER by using SPEC CPU 2006's C++ benchmarks and various browser benchmarks for Chromium and Firefox. For comparison, we measured runtime overheads of the original, non-instrumented version (compiled with `clang`), and the UBSAN-hardened version.

**Microbenchmarks.** To understand the performance characteristics of CAVER-hardened applications, we first profiled micro-scaled runtime behaviors related to CAVER's operations (Table 6). For workloads, we used the built-in input for the two C++ applications of SPEC CPU 2006, and loaded the default start page of the Chromium and Firefox browsers. Overall, CAVER traced considerable number of objects, especially for the browsers: 783k in Chromium, and 15,506k in Firefox.

We counted the number of *verified castings* (see Table 6), and the kinds of allocations (i.e., global, stack, or heap). In our experiment, Firefox performed 710% more castings than Chromium, which implies that the total number of verified castings and the corresponding performance overheads highly depends on the way the application is written and its usage patterns.

**SPEC CPU 2006.** With these application characteristics in mind, we first measured runtime performance impacts of CAVER on two SPEC CPU 2006 programs, xalancbmk and soplex. CAVER slowed down the execution of xalancbmk and soplex by 29.6% and 20.0%, respectively. CAVER-NAIVE (before applying the optimization techniques described in §4.4) slowed down xalancbmk and soplex by 32.7% and 20.8% respectively.

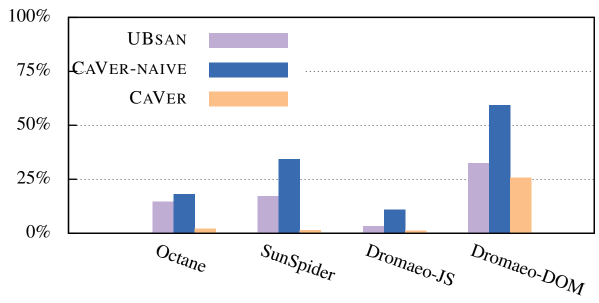


**Figure 4:** Browser benchmark results for the Chromium browser. On average, while CAVER-NAIVE incurs 30.7% overhead, CAVER showed 7.6% runtime overhead after the optimization. UBSAN exhibits 16.9% overhead on average.

For UBSAN, xalancbmk crashed because of RTTI limitations in handling non-polymorphic types, and soplex becomes 21.1% slower. Note, the runtime overheads of CAVER is highly dependent on the application characteristics (e.g., the number of downcasts performed in runtime). Thus, we measured overhead with more realistic workloads on two popular browsers, Chromium and Firefox.

**Browser benchmarks (Chromium).** To understand the end-to-end performance of CAVER, we measured the performance overhead of web benchmarks. We tested four browser benchmarks: Octane [21], SunSpider [47], Dromaeo-JS [29], and Dromaeo-DOM [29], each of which evaluate either the performance of the JavaScript engine or page rendering.

Figure 4 shows the benchmark results of the Chromium browser. On average, CAVER showed 7.6% overhead while CAVER-NAIVE showed 30.7%, which implies the

| Name | Object Tracing | | | | | Verified Castings | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Global | Stack | | Heap | | Global | Stack | Heap | Total |
| | Total | Peak | Total | Peak | Total | | | | |
| 483.xalancbmk | 165 | 32 | 190k | 8k | 88k | 0 | 104 | 24k | 24k |
| 450.soplex | 36 | 1 | 364 | 141 | 658 | 0 | 0 | 0 | 0 |
| Chromium | 3k | 274 | 350k | 79k | 453k | 963 | 338 | 150k | 151k |
| Firefox | 24k | 38k | 14,821k | 213k | 685k | 41k | 524k | 511k | 1,077k |

**Table 6:** The number of traced objects and type castings verified by CAVER in our benchmarks. Under the *Object Tracing* column, *Peak* and *Total* denote the maximum number of traced objects during program execution, and the total number of traced objects until its termination, respectively. *Global*, *Stack*, and *Heap* under the *Verified Casts* represent object's original types (allocation) involved in castings. Note that Firefox heavily allocates objects on stack, compared to Chromium. Firefox allocated 4,134% more stack objects, and performs 1,550% more type castings than Chromium.

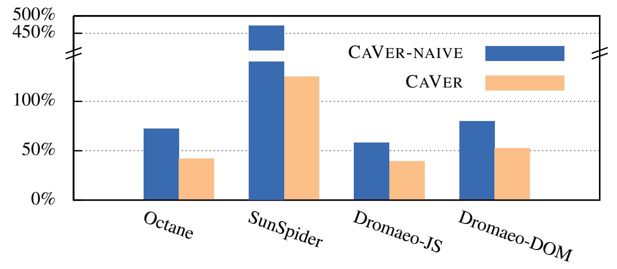


**Figure 5:** Browser benchmark results for the Firefox browser. On average, CAVER and CAVER-NAIVE showed 64.6% and 170.3% overhead, respectively.

| Name | Original | | UBSAN | | CAVER | |
|---|---|---|---|---|---|---|
| | Peak | Avg | Peak | Avg | Peak | Avg |
| 483.xalancbmk | 9 | 8 | crash | crash | 14 | 12 |
| 450.soplex | 2 | 2 | 2 | 2 | 5 | 5 |
| Chromium | 376 | 311 | 952 | 804 | 878 | 738 |
| Firefox | 165 | 121 | crash | crash | 208 | 157 |

**Table 7:** Runtime memory impacts (in KB) while running target programs. UBSAN crashed while running xalancbmk and Firefox due to the non-polymorhpic typed classes. *Peak* and *Avg* columns denote the maximum and average memory usages, respectively, while running the program. CAVER used 137% more memory on Chromium, and 23% more memory on Firefox. UBSAN used 158% more memory on Chromium.

optimization techniques in §4.4 provided a 23.1% performance improvement. This performance improvement is mostly due to the safe-allocation optimization, which identified 76,381 safe-allocation types (81% of all types used for Chromium) and opted-out to instrument allocation sites on such types. Compared to UBSAN, CAVER is 13.8% faster even though it offers more wide detection coverage on type casting. Thus, we believe this result shows that CAVER's `THTable` design and optimization techniques are efficient in terms of runtime performances.

**Browser benchmarks (Firefox).** We applied CAVER to the Firefox browser and measured the performance overhead for the web benchmarks used in evaluating the Chromium browser. On average, CAVER imposed 64.6% overhead while CAVER-NAIVE imposed 170.3% overhead (Figure 5). Similar to the Chromium case, most of performance improvements are from safe-allocation optimization, which identified 21,829 safe-allocation types (72% of all used types for Firefox). UBSAN was unable to run Firefox because it crashed due to the inability of its RTTI to handle non-polymorphic types, so we do not present the comparison number. Compared to CAVER's results on Chromium, the CAVER-enhanced Firefox showed worse performance, mainly due to the enormous amount of stack objects allocated by Firefox (Table 6). In order words, the potential performance impacts rely on the usage pattern of target applications, rather than the inherent overheads of CAVER's approaches.

**Memory overheads.** UBSAN and CAVER achieve fast lookup of the metadata of a given object by using a custom memory allocator that is highly optimized for this purpose, at the cost of unnecessary memory fragmentation. In our benchmark (Table 7), UBSAN used 2.5× more memory at peak and average; and CAVER used 2.3× more memory at peak and average, which is an 8% improvement over UBSAN. Considering CAVER's main purpose is a diagnosis tool and the amount of required memory is not large (< 1 GB), we believe that these memory overheads are acceptable cost in practice for the protection gained.

## 7 Discussion

**Integration with fuzzing tools.** During our evaluations, we relied on the built-in test inputs distributed with the target programs, and did not specifically attempt to improve code coverage. Yet CAVER is capable of discovering dozens of previously unknown bad-casting bugs. In the future, we plan to integrate CAVER with fuzzing tools like the ClusterFuzz [2] infrastructure for Chromium to improve code coverage. By doing so, we expect to discover more bad-casting vulnerabilities.

**Optimization.** In this paper, we focused on the correctness, effectiveness, and usability of CAVER. Although we developed several techniques to improve performance, optimization is not our main focus. With more powerful optimization techniques, we believe CAVER can also be used for runtime bad-casting mitigation.

For example, one direction we are pursuing is to use static analysis to prove whether a type casting is always safe. By doing so, we can remove redundant cast verification.

Another direction is to apply alignment-based direct mapping scheme for global and stack objects as well. Please recall that red-black trees used for global and stack objects show $O(logN)$ complexity, while alignment-based direct mapping scheme guarantees $O(1)$ complexity. In order to apply alignment-based direct mapping scheme for global and stack objects together, there has to be radical semantic changes in allocating stack and global objects. This is because alignment-based direct mapping scheme requires that all objects have to be strictly aligned. This may not be difficult for global objects, but designing and implementing for stack objects would be non-trivial for the following reasons: (1) essentially this may involve a knapsack problem (i.e., given different sized stack objects in each stack frame, what are the optimal packing strategies to reduce memory uses while keeping a certain alignment rule); (2) an alignment base address for each stack frame has to be independently maintained during runtime; (3) supporting variable length arrays (allowed in ISO C99 [18]) in stack would be problematic as the packing strategy can be only determined at runtime in this case.

Furthermore, it is also possible to try even more extreme approaches to apply alignment-based direct mapping scheme—simply migrating all stack objects to be allocated in heap. However, this may result in another potential side effects in overhead.

## 8 Related work

**Bad-casting detection.** The virtual function table checking in Undefined Behavior Sanitizer (UBSAN-vptr) [42] is the closest related work to CAVER. Similar to CAVER, UBSAN instruments `static_cast` at compile time, and verifies the casting at runtime. The primary difference is that UBSan relies on RTTI to retrieve the type information of an object. Thus, as we have described in §4, UBSAN suffers from several limitations of RTTI . (1) Coverage: UBSAN cannot handle non-polymorphic classes as there is no RTTI for these classes; (2) Ease-of-deployments: hardening large scale software products with UBSAN is non-trivial due to the coverage problem and phantom classes. As a result, UBSAN has to rely on blacklisting [9] to avoid crashes.

**RTTI alternatives.** Noticing the difficulties in handling complex C++ class hierarchies in large-scale software, several open source projects use a custom form of RTTI. For example, the LLVM project devised a custom RTTI [27]. LLVM-style RTTI requires all classes to mark its identity once it is created (i.e., in C++ constructors) and further implement a static member function to retrieve its identity. Then, all type conversions can be done with templates that leverage this static member function implemented in every class. Because the static member function can tell the true identity of an object, theoretically, all type conversions are always correct and have no bad-casting issues. Compared to CAVER, the drawback of this approach is that it requires manual source code modification. Thus, it would be non-trivial to modify large projects like browsers to switch to this style. More alarmingly, since it relies on developers' manual modification, if developers make mistakes in implementations, bad-casting can still happen [41].

**Runtime type tracing.** Tracing runtime type information offers several benefits, especially for debugging and profiling. [37] used RTTI to avoid complicated parsing supports in profiling parallel and scientific C++ applications. Instead of relying on RTTI, [15, 28] instruments memory allocation functions to measure complete heap memory profiles. CAVER is inspired by these runtime type tracing techniques, but it introduced the `THTable`, a unique data structure to support efficient verification of complicated type conversion.

**Memory corruption prevention.** As described in §2, bad-casting can provide attackers access to memory beyond the boundary of the casted object. In this case, there will be a particular violation (e.g., memory corruptions) once it is abused to mount an attack. Such violations can be detected with existing software hardening techniques, which prevents memory corruption attacks and thus potentially stop attacks abusing bad-casting. In particular, Memcheck (Valgrind) [34] and Purify [23] are popularly used solutions to detect memory errors. AddressSanitizer [36] is another popular tool developed recently by optimizing the way to represent and probe the status of allocated memory. However, it cannot detect if the attacker accesses beyond red-zones or stressing memory allocators to abuse a quarantine zone [8]. Another direction is to enforce spatial memory safety [14, 25, 32, 33, 48], but this has drawbacks when handling bad-casting issues. For example, Cyclone [25] requires extensive code modifications; CCured [33] modifies the memory allocation model; and SVA [14] depends on a new virtual execution environment. More fundamentally, most only support C programs.

Overall, compared to these solutions, we believe CAVER makes a valuable contribution because it detects the root cause of one important vulnerability type: bad-casting. CAVER can provide detailed information on how a bad-casting happens. More importantly, depending on certain test cases or workloads, many tools cannot detect bad-casting if a bad-casted pointer is not actually used to violate memory safety. However, CAVER can immediately detect such latent cases if any bad-casting occurs.

**Control Flow Integrity (CFI).** Similar to memory corruption prevention techniques, supporting CFI [1, 49–51] may prevent attacks abusing bad-casting as many exploits hijack control flows to mount an attack. Furthermore, specific to C++ domain, SafeDispatch [24] and VTV [45] guarantee the integrity of virtual function calls to prevent hijacks over virtual function calls. First of all, soundly implementing CFI itself is challenging. Recent research papers identified security holes in most of CFI implementations [6, 17, 19, 20]. More importantly, all of these solutions are designed to only protect control-data, and thus it cannot detect any non-control data attacks [7]. For example, the recent vulnerability exploit against `glibc` [35] was able to completely subvert the victim's system by merely overwriting non-control data—`EXIM`'s runtime configuration. However, because CAVER is not relying on such post-behaviors originating from bad-casting, it is agnostic to specific exploit methods.

## 9 Conclusion

The bad-casting problem in C++ programs, which occurs when the type of an object pointer is converted to another that is incorrect and unsafe, has serious security implications. We have developed CAVER, a runtime bad-casting detection tool. It uses a new runtime type tracing mechanism, the Type Hierarchy Table, to efficiently verify type casting dynamically. CAVER provides broader coverage than existing approaches with smaller or comparable performance overhead. We have implemented CAVER and have applied it to large-scale software including the Chromium and Firefox browsers. To date, CAVER has found eleven previously unknown vulnerabilities, which have been reported and subsequently fixed by the corresponding open-source communities.

## Acknowledgment

The authors would like to thank the anonymous reviewers for their helpful feedback, as well as our operations staff for their proofreading efforts. We also would like to thank Abhishek Arya, Kostya Serebryany, Alexey Samsonov, Richard Smith, and Parisa Tabriz for their helpful feedback on the paper. This material is based upon work supported in part by the National Science Foundation under Grants No. CNS-1017265, CNS-0831300, CNS-1149051 and DGE 1500084, by the Office of Naval Research under Grant No. N000140911042, No. N000141512162, by the Department of Homeland Security under contract No. N66001-12-C-0133, by the United States Air Force under Contract No. FA8650-10-C-7025, and by ETRI MSIP/IITP[B0101-15-0644]. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Office of Naval Research, the Department of Homeland Security, or the United States Air Force.

## References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] C. N. Abhishek Arya. Fuzzing for Security (The Chromium Blog). http://blog.chromium.org/2012/04/fuzzing-for-security.html, 2012.

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

[4] Bad casting: From BasicThebesLayer to BasicContainerLayer. Mozilla Bugzilla - Bug 1074280. https://bugzilla.mozilla.org/show_bug.cgi?id=1074280, Nov 2014.

[5] Bad-casting from PRCListStr to nsSHistory. Mozilla Bugzilla - Bug 1089438. https://bugzilla.mozilla.org/show_bug.cgi?id=1089438, Nov 2014.

[6] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX Security Symposium (Security)*, 2014.

[7] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data Attacks Are Realistic Threats. In *USENIX Security Symposium (Security)*, 2005.

[8] Chris Evans. Using ASAN as a protection. http://scarybeastsecurity.blogspot.com/2014/09/using-asan-as-protection.html, Nov 2014.

[9] Chromium. Chromium Revision 285353 - Blacklist for UBsan's vptr. https://src.chromium.org/viewvc/chrome?view=revision&revision=285353, Jul 2014.

[10] Chromium. Chromium project: Log of /trunk/src/tools/ubsan_-vptr/blacklist.txt. https://src.chromium.org/viewvc/chrome/trunk/src/tools/ubsan_vptr/blacklist.txt?view=log, Nov 2014.

[11] Clang Documentation. MSVC Compatibility. http://clang.llvm.org/docs/MSVCCompatibility.html, Nov 2014.

[12] CodeSourcery, Compaq, EDG, HP, IBM, Intel, R. Hat, and SGI. Itanium C++ ABI (Revision: 1.83). http://mentorembedded.github.io/cxx-abi/abi.html, 2005.

[13] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. C++ ABI Closed Issues. www.codesourcery.com/public/cxx-abi/cxx-closed.html, Nov 2014.

[14] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[15] Dai Mikurube. C++ Object Type Identifier for Heap Profiling. http://www.chromium.org/developers/deep-memory-profiler/cpp-object-type-identifier, Nov 2014.

[16] M. Daniel, J. Honoroff, and C. Miller. Engineering Heap Overflow Exploits with JavaScript. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2008.

[17] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *USENIX Security Symposium (Security)*, 2014.

[18] GCC. Arrays of Variable Length. https://gcc.gnu.org/onlinedocs/gcc/Variable-Length.html, Jun 2015.

[19] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming Control-Flow Integrity. In *IEEE Symposium on Security and Privacy (SP)*, 2014.

[20] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *USENIX Security Symposium (Security)*, 2014.

[21] Google. Octane Benchmark. https://code.google.com/p/octane-benchmark, Aug 2014.

[22] Google. Specialized memory allocator for ThreadSanitizer, MemorySanitizer, etc. http://llvm.org/klaus/compiler-rt/blob/7385f8b8b8723064910cf9737dc929e90aeac548/lib/sanitizer_common/sanitizer_allocator.h, Nov 2014.

[23] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Winter 1992 USENIX Conference*, 1991.

[24] D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[25] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference (ATC)*, 2002.

[26] I. JTC1/SC22/WG21. ISO/IEC 14882:2013 Programming Language C++ (N3690). https://isocpp.org/files/papers/N3690.pdf, 2013.

[27] LLVM Project. How to set up LLVM-style RTTI for your class hierarchy. http://llvm.org/docs/HowToSetUpLLVMStyleRTTI.html, Nov 2014.

[28] J. Mihalicza, Z. Porkoláb, and A. Gabor. Type-preserving heap profiler for c++. In *IEEE International Conference on Software Maintenance (ICSM)*, 2011.

[29] Mozilla. DROMAEO, JavaScript Performance Testing. http://dromaeo.com, Aug 2014.

[30] Multiple undefined behaviors (static_cast<>) in libstdc++-v3/include/bits. GCC Bugzilla - Bug 63345. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=63345, Nov 2014.

[31] MWR Labs. MWR Labs Pwn2Own 2013 Write-up: Webkit Exploit. https://labs.mwrinfosecurity.com/blog/2013/04/19/mwr-labs-pwn2own-2013-write-up---webkit-exploit/, 2013.

[32] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[33] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2005.

[34] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[35] Qualys Security Advisory. Qualys Security Advisory CVE-2015-0235 - GHOST: glibc gethostbyname buffer overflow. http://www.openwall.com/lists/oss-security/2015/01/27/9, Jun 2015.

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

[37] S. Shende, A. D. Malony, J. Cuny, P. Beckman, S. Karmesin, and K. Lindlan. Portable profiling and tracing for parallel, scientific applications using c++. In *ACM SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT)*, 1998.

[38] A. Sotirov. Heap Feng Shui in JavaScript. *Black Hat Europe*, 2007.

[39] Standard Performance Evaluation Corporation. SPEC CPU 2006. http://www.spec.org/cpu2006, Aug 2014.

[40] The Chromium Project. http://www.chromium.org/Home, Aug 2014.

[41] The Chromium Project. Chromium Issues - Bug 387016. http://code.google.com/p/chromium/issues/detail?id=387016, Nov 2014.

[42] The Chromium Projects. Undefined Behavior Sanitizer for Chromium. http://www.chromium.org/developers/testing/undefinedbehaviorsanitizer, Nov 2014.

[43] The LLVM Compiler Infrastructure. http://llvm.org, Aug 2014.

[44] The Mozilla Foundation. Firefox Web Browser. https://www.mozilla.org/firefox, Nov 2014.

[45] C. Tice. Improving Function Pointer Security for Virtual Method Dispatches. In *GNU Tools Cauldron Workshop*, 2012.

[46] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2. *TIS Committee*, 1995.

[47] WebKit. SunSpider 1.0.2 JavaScript Benchmark. https://www.webkit.org/perf/sunspider/sunspider.html, Aug 2014.

[48] W. Xu, D. C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of c programs. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2004.

[49] B. Zeng, G. Tan, and G. Morrisett. Combining Control-flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[50] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *IEEE Symposium on Security and Privacy (SP)*, 2013.

[51] M. Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *USENIX Security Symposium (Security)*, 2013.

## Appendix

```
def getTHTableByAddr(addr):
    if isObjectInStack(addr):
        # addr points to stack objects.
        stack_rbtree = getThreadLocalStackRbtree()
        allocBaseAddr, pTHTable = \
                           stack_rbtree.rangedSearch(addr):
        return (allocBaseAddr, pTHTable)

    if isObjectInHeap(addr):
        # addr points to heap objects.
        MetaData = getMetaDataStorage(addr):
        return (MetaData.allocBaseAddr, MetaData.pTHTable)

    if isObjectInGlobal(addr):
        # addr points to global objects.
        allocBaseAddr, pTHTable = \
                           global_rbtree.rangedSearch(addr):
        return (allocBaseAddr, pTHTable)

    # addr points to unknown area.
    return ERROR

# Return True if there exists a good-casting.
#         False if there is no good-castings.
def isGoodCast(addr, allocBaseAddr, THTable, TargetTypeHash):
    # Handle compositions recursively.
    for i in range(THTable.num_composites):
        comp = THTable.comps[i]
        if addr >= allocBaseAddr + comp.offset
          and addr < allocBaseAddr + comp.offset + comp.size:
            if isGoodCast(addr, allocBaseAddr + comp.offset,
                            comp.thtable, TargetTypeHash):
                return True

    # Check bases.
    for i in range(THTable.num_bases):
        base = THTable.bases[i]
        if addr == allocBaseAddr + base.offset
          and base.hashValue == TargetTypeHash:
            return True

    # Check phantom.
    TargetTHTable = getTHTableByHash(TargetTypeHash)
    for i in range(TargetTHTable.num_bases):
        base = TargetTHTable.bases[i]
        if addr == allocBaseAddr + base.offset
          and base.hashValue == THTable.type_hash
          and base.isPhantom:
            return True

    return False

def verify_cast(beforeAddr, afterAddr, TargetTypeHash):
    (allocBaseAddr, pTHTable) = getTHTableByAddr(beforeAddr)
    if pTHTable == ERROR:
        return

    if isGoodCast(afterAddr, allocBaseAddr, \
                  THTable, TargetTypeHash):
        # This is a good casting.
        return

    # Reaching here means a bad-casting attempt is detected.
    # Below may report the bug, halt the program, or nullify
    # the pointer according to the user's configuration.
    HandleBadCastingAttempt()
```

**Appendix 1:** Algorithm for verifying type conversions based on the tracked type information.

```
# global_rbtree is initialized per process.
def trace_global(pTHTable, baseAddr, numArrayElements):
    allocSize = pTHTable.type_size * numArrayElements
    global_rbtree.insert((baseAddr, allocSize), pTHTable)
    return

# stack_rbtree is initialized per thread.
def trace_stack_begin(pTHTable, baseAddr, numArrayElements):
    stack_rbtree = getThreadLocalStackRbtree()
    allocSize = pTHTable.type_size * numArrayElements
    stack_rbtree.insert((baseAddr, allocSize), pTHTable)
    return

def trace_stack_end(baseAddr):
    stack_rbtree = getThreadLocalStackRbtree()
    stack_rbtree.remove(baseAddr)
    return

# Meta-data storage for dynamic objects are reserved
# for each object allocation.
def trace_heap(pTHTable, baseAddr, numArrayElements):
    MetaData = getMetaDataStorage(baseAddr)
    MetaData.baseAddr = baseAddr
    MetaData.allocSize = pTHTable.type_size * numArrayElements
    MetaData.pTHTable = pTHTable
    return
```

**Appendix 2:** Algorithm for tracking type information on objects in runtime.