Don't Waste My Efforts: Pruning Redundant Sanitizer Checks of Developer-Implemented Type Checks

Yizhuo Zhai[1], Zhiyun Qian[1], Chengyu Song[1], Manu Sridharan[1],
Trent Jaeger[1], Paul Yu[2], Srikanth V. Krishnamurthy[1]

[1]University of California, Riverside
[2]U.S. Army Research Laboratory

Abstract

Type confusion occurs when C or C++ code accesses an object after casting it to an incompatible type. The security impacts of type confusion vulnerabilities are significant, potentially leading to system crashes or even arbitrary code execution. To mitigate these security threats, both static and dynamic approaches have been developed to detect type confusion bugs. However, static approaches can suffer from excessive false positives, while existing dynamic approaches track type information for each object to enable safety checking at each cast, introducing a high runtime overhead.

In this paper, we present a novel tool T-PRUNIFY to reduce the overhead of dynamic type confusion sanitizers. We observe that in large complex C++ projects, to prevent type confusion bugs, developers often add their own encoding of runtime type information (RTTI) into classes, to enable efficient runtime type checks before casts. T-PRUNIFY works by first identifying these custom RTTI in classes, automatically determining the relationship between field and method return values and the concrete types of corresponding objects. Based on these custom RTTI, T-PRUNIFY can identify cases where a cast is protected by developer-written type checks that guarantee the safety of the cast. Consequently, it can safely remove sanitizer instrumentation for such casts, reducing performance overhead. We evaluate T-PRUNIFY based on HexType, a state-of-the-art type confusion sanitizer that supports extensive C++ projects such as Google Chrome. Our findings demonstrate that our method significantly lowers HexType's average overhead by 25% to 75% in large C++ programs, marking a substantial enhancement in performance.

1 Introduction

Type confusion [3] is a critical type of vulnerability common in type-unsafe programming languages such as C/C++. It occurs when the program allocates or initializes a resource such as a pointer or object using one type, but later accesses that resource using a type that is incompatible with the original type. When a program experiences type confusion, it can result in undesirable behaviors such as a system crash [10], information leakage [13], and even arbitrary code execution [11].

To mitigate the threats introduced by type confusion vulnerabilities, both static and dynamic analyses have been applied. For example, static type inference and pointer analysis have been developed to determine if a type cast is safe [9]. However, static approaches can generate a large number of false positives because of their inherent imprecision and over-approximation.

Besides static analysis, sanitizer-style runtime checks have also been proposed [19, 22, 24, 26, 30, 31]. The key idea is simple. Instead of checking for type compatibility at compile time, the sanitizer instruments the program and tracks the type of objects at runtime in order to perform precise type checks. Clearly, such approaches are more precise than static approaches, but are also more expensive as they incur runtime overhead. Although these type confusion sanitizers are faster than C++'s built in `dynamic_cast` [22, 24, 26], their runtime overhead is still too high for deployment in production systems.

One promising direction to reduce runtime overhead is to statically recognize safe casts and avoid instrumentation selectively. For example, CaVer [26] and HexType [24] track data flow via pointers from object allocation sites to cast sites; if the cast type is a supertype of the allocation site type, then the corresponding runtime check can be eliminated. However, as discussed in [9], static analyses are in general imprecise and will often make conservative inferences, i.e., treating a safe type cast as potentially unsafe. For example, pointer analysis, relied upon by CaVer [26], is a known hard problem in static analysis. As a result, many unnecessary runtime checks cannot be eliminated.

In this paper, we make the observation that *pointer analysis is not the only way to show a cast is safe*. In particular, we find that developers often encode custom runtime type information (RTTI) directly into a structure or class, especially in complex C++ class hierarchies, to facilitate their own type checks. For example, Chrome defines a class `BasicShape` and many classes inherit from this class, such as `BasicShapeCircle`, `BasicShapeEllipse` and `BasicShapePolygon`. Those subclasses override a virtual function `GetType()`, which returns enumeration constants `kBasicShapeCircleType`, `kBasicShapeEllipseType`, and `kBasicShapePolygonType`, respectively. In addition, we find that before casting a pointer `BasicShape *basic_shape` to a subclass, developers usually insert a type check using `GetType()`, e.g., checking that `basic_shape->GetType() == kBasicShapeCircle` before casting to `BasicShapeCircle`. Such checks ensure the downcast is safe – and are often voluntarily inserted to avoid type confusion bugs [5, 6, 8].

Our key insight is to leverage these developer-implemented type checks to discover opportunities to remove redundant sanitizer checks for type casts. Achieving this goal requires addressing two key challenges. First, we must *identify* developer-encoded RTTI and the corresponding type checks, and *validate* the correspondence between these checks and the class hierarchy. Developer-written type checks may involve arbitrary logic and can vary from one class hierarchy to another, and the source code does not explicitly identify which logic serves as a type check. To this end, we conduct an exploratory investigation of real-world programming conventions in Chrome, distilling them into several general patterns. The most-common patterns encoded custom RTTI as a set of predefined values (i.e., constants) in class definitions. By performing a class-hierarchy-wide static analysis, we could track the definitions and uses of such values and thereby automatically deduce developer-inserted custom RTTI and type checks. Note that our analysis *verifies* the correspondence between the custom RTTI and the type hierarchy – each class needs to take a unique value, allowing it to be distinguished from others in the same hierarchy.

The second key challenge is to discover where a type cast is correctly guarded by a custom type check, thereby proving the cast cannot fail and enabling the removal of redundant sanitizer checks. We tackle this challenge by using a flow- and field-sensitive intra-procedural analysis to track the refined type suggested by developer-implemented type checks, validating if a downcast is always safe under all execution paths.

Based on the above insights, we implemented our techniques in an automated solution called T-PRUNIFY, that conducts static analyses for C++ programs to (1) extract custom RTTI based on understanding and surveying diverse class hierarchies in popular applications, (2) identify developer-implemented type checks based on the identified type information, and (3) further validate the developer-implemented type checks. In addition, T-PRUNIFY also uses the results to prune unnecessary sanitizer checks. Our analysis is designed to be sound with multiple verifications throughout the process and will conservatively prune sanitizer checks only when the developer-implemented checks are deemed safe. We evaluated our solution on the SPEC CPU 2006 C++ programs, LLVM toolchain and Chromium browser (the open-sourced version of Google Chrome), one of the largest C++ programs, which is also known to be prone to type-confusion vulnerabilities [14–17]. The results showed that T-PRUNIFY can indeed prune a large number of sanitizer checks safely and reduce the runtime performance overhead compared to the state-of-the-art sanitizer-based solution.

In summary, our main contributions of this paper are:

- We identify developer-inserted custom runtime type checks as a previously-overlooked source of opportunity to reduce the performance overhead of type confusion mitigation techniques.
- We develop a custom solution to (1) automatically identify developer-implemented custom runtime type checks and (2) leverage them to prove the safety of type casting in C++ programs.
- We develop T-PRUNIFY that packages the insight into a fully-automated system. To facilitate further research in this direction, we have made the system open source and accessible at `https://github.com/seclab-ucr/TPrunify`.
- We evaluate T-PRUNIFY against a state-of-the-art type sanitizer Hextype [24] against popular open-source projects and showed that the relative runtime overhead reductions range from 25% to 75% for Chromium under standard benchmarks, and 35.19% for the LLVM Clang toolchain.

2 Background and Motivation

In this section, we start with some basic background relevant to type confusion. First, we describe the C++ type hierarchy, casting operations and type confusion vulnerabilities. Then we illustrate, with examples, how previous type confusion checks are performed statically or dynamically (including sanitizer checks executed at runtime), and why they are inefficient. This will then motivate the design and implementation of T-PRUNIFY.

2.1 Type Casting in C++

C++ is an object-oriented programming language, which allows programmers to define new types as classes. A class can inherit from multiple ancestor classes. Descendant classes inherit members (methods and variables) from their ancestor(s) and can optionally define additional members [28]. Generally speaking, upcasts (i.e., casting a pointer of a derived class to a pointer of an ancestor class) are considered safe, because the memory scope an ancestor pointer can access is strictly smaller than the memory scope of a descendent class; however, downcasts (i.e., casting a pointer of an ancestor class to a pointer of a descendent class) may introduce memory corruption vulnerabilities, when the underlying allocated memory object has a smaller memory scope than the destination type demands. Figure 1 illustrates such an example; the code allocates a pointer of the base class and subsequently casts it to the derived class (which is always at least as big as the base class). In this example, the derived class includes an additional field `y`, and accessing this field on the improperly-casted `Base` pointer leads to an out-of-bounds memory access.

```
struct Base {int x;};
struct Derived : Base {int y;};
Base *base = new Base();
Derived *derived;
derived = static_cast<Derived*>(base);
derived->y; //<-error
```

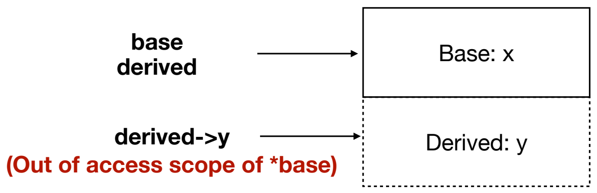


Figure 1: A code example and diagram of a type confusion problem where a base class is incorrectly accessed using a pointer to a derived class. The static_cast results in type confusion and accessing the field `y` is out of the access scope of type `Base`. In the class definition, the members are public, we ignore it to save space.

C++ provides four built-in type casting operations, including `reinterpret_cast`, `static_cast`, `dynamic_cast`, and `const_cast`, which we describe below:

`reinterpret_cast<dest>(src)` is similar to an explicit cast in C, which allows conversion between any two arbitrary types, regardless of their compatibility. Although this primitive grants flexibility, the lack of safety checks can lead to type confusion bugs.

`static_cast<dest>(src)` casts a pointer/object of `src` type to a pointer/object of the `dst` type. Unlike `reinterpret_cast`, `static_cast` performs lightweight compile-time type checking to avoid bad casting. Specifically, the compiler will verify whether the source class and the destination class are within the same class hierarchy. However, it cannot detect an unsafe downcast and thus, can still result in type confusion bugs.

`dynamic_cast<dest>(src)` can avoid unsafe downcasts because it performs a runtime type check to make sure the allocation type of the source object is actually compatible with the destination type, using C++'s own runtime type information (RTTI). To perform a runtime type check, `dynamic_cast` first locates the RTTI of the source object from a pointer stored in its virtual function table. The RTTI contains a null-terminated byte string of the mangled type name as the type information of the current type, and one or more pointers to its base classes' RTTI. To check for type compatibility, `dynamic_cast` recursively compares the mangled name of all base classes of the source object with the mangled name of the destination type. If a match is found, the casting is valid and `dynamic_cast` returns a valid pointer; otherwise a null pointer is returned. Since the type check involves slow string comparison and possible recursive traversal of the base classes' RTTI, using `dynamic_cast` operators can be 90 times slower than using `static_cast` [26]. Due to its performance overhead, `dynamic_cast` is intentionally avoided in release builds of well-developed applications such as Chrome.

`const_cast<dest>(src)` simply drops the `const` qualifier of the source object. Since it does not actually modify the type itself, it is not of interest to us.

For backward compatibility, C++ compilers also support C-style explicit casts. When encountered, the compiler will try the following sequence of casts one after another until the program can be compiled without an error: (1) `const_cast`, (2) `static_cast`, and (3) `reinterpret_cast`.

2.2 Type Confusion Sanitizers

Type confusion sanitizers [22, 24, 26] aim to overcome two limitations of `dynamic_cast`: (1) its high runtime performance overhead and (2) its limited protection scope (i.e., it only supports classes with virtual function tables). To reduce the runtime overhead, type confusion sanitizers typically employ two optimizations. First, instead of storing the type information as a string, they store a unique hash (guaranteed at link time) of the mangled name, so that a compatibility check can be done using an integer comparison instead of a string comparison. Second, they compact all base classes' type information into a single RTTI, where the hash values are also sorted for faster binary search. To overcome the second limitation, type confusion sanitizers store the RTTI of an object in a disjoint lookup table. To perform runtime type checks, type confusion sanitizers first extract the type hierarchies during compile time and emit RTTI that records all the compatible types of a class (i.e., all ancestor classes, including itself). At runtime, after an object is allocated, type confusion sanitizers associate the object with its RTTI (e.g., by using a hash table where the key is the address of the object and the value is the RTTI entry). At the cast site, the sanitizer retrieves the RTTI and looks for a matching hash value among all compatible types with the destination type.

2.3 C++ Casting with Custom Run Time Type Information (RTTI)

Although designers of type confusion sanitizers have spent lots of effort trying to reduce the runtime overhead, these sanitizers still introduce significant overhead. For example, during our evaluation, a state-of-the-art type confusion sanitizer HexType [24] still introduces a 10.2% – 65.7% overhead on Chromium benchmarks. A large portion of this overhead comes from its lookup of RTTI. Specifically, to maintain

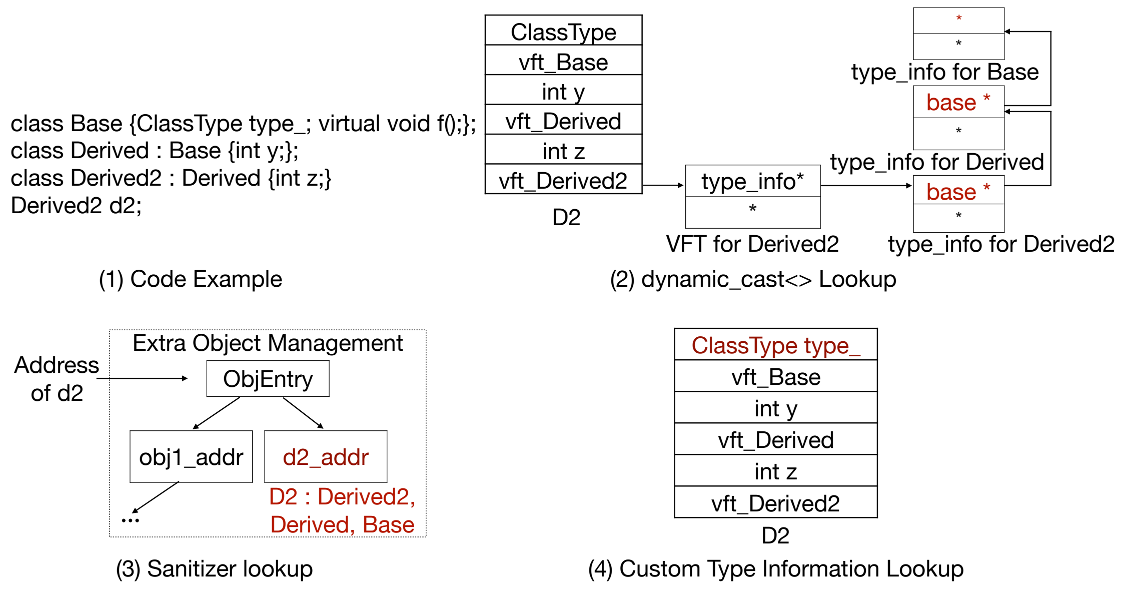


Figure 2: The code example and three different methods for verifying the safety of a type cast. (2) The `dyn_cast` method validates the safety of the cast by utilizing the type information stored in the virtual table. (3) Sanitizer approaches perform type checks by tracking and extracting the type information stored in a disjoint data set. (4) Developers perform the type checks directly through the custom RTTI defined within the class hierarchy.

binary compatibility, type confusion sanitizers use decoupled RTTI (i.e., the RTTI of an object is stored disjointedly at another memory location). As a result, they need extra steps to find the RTTI. In practice, large C++ projects like the Chromium browser and the LLVM compiler frameworks prefer developer-inserted custom RTTI and type checks to prevent type confusion vulnerabilities. Because such RTTI is tightly coupled with an object (e.g., as a member field or a special virtual function), runtime type checking is much more efficient than both `dynamic_cast` and type confusion sanitizers.

Figure 2 visualizes the differences between the three different approaches, i.e., `dynamic_cast`, type confusion sanitizers, and developer-inserted RTTI. In (1), a base class and two derived classes are defined, with one derived class inheriting from `Base` and the other from `Derived`. The `ClassType` is a predefined enumeration variable. An object `d2` of the `Derived2` class is then instantiated. In (2), the memory layout of the `d2` object is depicted. When a `dynamic_cast<dst>(d2)` operation is executed, it first locates the RTTI of the object, which is stored as the first item in the virtual function table (VFT). By following the pointer, the mangled name of the base class is extracted and compared with type name of `dst`. If a match is found, the casting is considered valid. However, this lookup process involves traversing a linked list, making it relatively slow. The sanitizers' lookup is demonstrated in (3). After obtaining the address of `d2`, it refers to an additional data structure specifically designed for efficient lookup. This data structure stores all compatible types of the runtime object, including `D2`. By utilizing caching, the lookup process becomes significantly faster compared to the `dynamic_cast` operation. Finally, in (4), the lookup process is illustrated when the class encodes custom RTTI. In this case, the field `type_` which holds the custom RTTI is directly accessed, and a value comparison is performed, resulting in the fastest lookup method.

In fact, during our investigation, we found that most of the type confusion vulnerabilities in the Chrome browser have been preemptively fixed by means of developer-implemented type checks with custom RTTI [5, 6, 8]. This motivates us to develop T-PRUNIFY. For instance, CVE-2021-30561 [12] is a type confusion vulnerability in Chrome. It allows a remote attacker to potentially exploit heap corruption via a crafted HTML page. Figure 3 shows the vulnerability and the main patch. Inside function `WasmJs::InstallConditionalFeatures()`, the variable `maybe_webassembly` is retrieved as a type of `Object`; it is then directly cast into a `JSObject` and used later. However, the object retrieved could in fact be of types other than `JSObject`, which causes a type confusion vulnerability. The patch fixed the bug by adding a custom type check `webassembly_obj->IsJSObject()` at line 15 to ensure that the type is of `JSObject` before proceeding to the subsequent type cast.

```
/* Main patch for CVE-2021-3056,
 * uninteresting code lines are ommited.
 */
void WasmJs::InstallConditionalFeatures(Isolate* isolate,
                                        Handle<Context>
                                        ↪  context) {
-  Handle<JSObject> webassembly = Handle<JSObject>::cast(
-    maybe_webassembly.ToHandleChecked());
+  Handle<Object> webassembly_obj;
+  if (!maybe_webassembly.ToHandle(&webassembly_obj)) {
+    // There is not {WebAssembly} object.
+    // We just return without adding the
+    // {Exception} constructor.
+    return;
+  }
+  if (!webassembly_obj->IsJSObject()) {
+    // The {WebAssembly} object is invalid.
+    // As we cannot add the {Exception}
+    // constructor, we just return.
+    return;
+  }
+  Handle<JSObject> webassembly =
↪  Handle<JSObject>::cast(webassembly_obj);
}
```

Figure 3: The simplified patch for CVE-2021-30561

3 Overview of T-PRUNIFY

In cases where developers have already encoded custom runtime type information (RTTI) into C++ classes and implemented their own type checks, we aim to identify and remove redundant type confusion sanitizer checks, thereby reducing sanitizer overhead. To this end, we design and implement a lightweight static analysis tool named T-PRUNIFY that can achieve this goal.

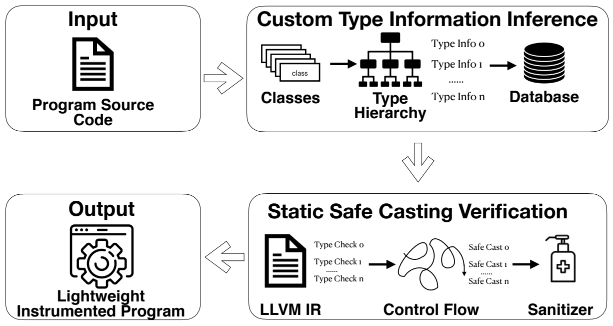


Figure 4: Work flow of T-PRUNIFY.

Figure 4 depicts the workflow of T-PRUNIFY. It consists of two high-level components, which are briefly described below:

1. Custom Type Check Inference. In this step, T-PRUNIFY takes the source code of the target program as input and attempts to recognize all the custom type checks inserted by developers. To do so, we first infer custom runtime type information (RTTI) encoded by developers by analyzing class definitions. The challenge lies in the lack of a unified standard for encoding or annotating such custom type information, which can vary across modules and class hierarchies in the program. Therefore, we need to have a comprehensive understanding of the various patterns developers can choose to encode custom types. In our solution, we perform an offline manual investigation of various class hierarchies in Chrome, and summarize them into three common patterns. With these patterns, T-PRUNIFY performs static analysis to automatically recognize and validate custom RTTI in classes, and then stores them in a database. Based on the identified custom RTTI, we then recognize developer-implemented type checks (e.g., in the form of `if` statements) as operations over the type information.

2. Static Safe Casting Verification. Based on the identified custom type checks, we then try to prove statically whether a downcast is always safe (i.e., the destination class is always compatible with the class indicated by the type check under all execution paths). The challenge is that type casts may not be performed immediately after a type check, and furthermore the type check may not always be correct. To tackle this challenge, T-PRUNIFY employs an intra-procedural flow- and field-sensitive static analysis to track the refined type suggested by developer-implemented type checks, and validate if a downcast will be safe. Once T-PRUNIFY determines a cast is safe, it will instruct a sanitizer to not insert another redundant type check at compile time. In the end, the output is a hardened program that enjoys the same level of security guarantee but with a much lower runtime overhead.

4 Custom Type Check Inference

As mentioned above, T-PRUNIFY uses existing developer-inserted custom type checks to eliminate type confusion sanitizer-induced checks. In this section, we describe how to identify developer-inserted custom type checks.

4.1 Systematic Investigation

Developers can choose to explicitly encode the runtime type information (RTTI) in a class definition directly. However, such encoding can be ad-hoc as it is entirely up to the developers to design a scheme to differentiate types. To understand how custom RTTI are commonly encoded, we conduct a manual investigation of over 100 Chrome hierarchies and identified several common categories as follows.

Category 1: Custom RTTI encoded in a Base Class Field. One way to store type information in C++ is by using a field defined in the base class, which is initialized to a different and unique value in each subclass according to its concrete type. Figure 5 shows part of the type hierarchy that class `CSSValue` belongs to. Class `CSSValue` defines a private member called `class_type_`, which is initialized in the constructor. Three subclasses `CSSImageValue`, `CSSShadowValue`, and `CSSURIValue` initialize this member field through their respec-

```
class CSSValue {
public:
  explicit CSSValue(ClassType class_type)
      : class_type_(class_type) {}
  bool IsImageValue() const {
      return class_type_ == kImageClass; }
  bool IsShadowValue() const {
      return class_type_ == kShadowClass; }
  bool IsURIValue() const {
      return class_type_ == kURIClass; }
private:
  const uint8_t class_type_; // enum ClassType
};
class CSSImageValue : public CSSValue {
public:
  CSSImageValue() : CSSValue(kImageClass) {}
};
class CSSShadowValue : public CSSValue {
public:
  CSSShadowValue() : CSSValue(kShadowClass) {}
};
class CSSURIValue : public CSSValue {
public:
  CSSURIValue() : CSSValue(kURIClass) {}
};
```

Figure 5: An example in Category 1: type information is stored in a member field, which is initialized with a different enumeration constant. Arrow indicates the inheritance relationship.

tive constructors with different values. In this example, the three values, e.g., `kImageClass` are unique enumeration constants. The base class `CSSValue` also defines three utility functions for type checking `IsImageValue()`, `IsShadowValue()`, and `IsURIValue()`.

Category 2: Custom RTTI as a Constant Without Fields. Classes in this category override a virtual method defined in the base class to return different constants to indicate the actual type of the object. Consider the class hierarchy of class `BasicShape` shown in Figure 6, it defines a virtual method `GetType()`. Each subclass overrides this method by returning a different enumeration constant that uniquely identifies its object type. For example, the subclass `BasicShapeCircle` overrides `GetType()` inherited from the base class to return `kBasicShapeCircleType`, which is unique to this subclass.

Category 3: Custom RTTI as a Type Check Function. Classes in this category do not use enumeration constants to indicate custom types; instead, they define custom type check functions. Figure 7 illustrates such an example where a base class `CanvasImageSource` defines a number of virtual methods, like `IsVideoElement()` and `IsCanvasElement()`, which returns `false` by default. The two subclasses: `HTMLVideoElement` and `HTMLCanvasElement` each override the corresponding type check method to return `true`. When the `IsVideoElement()` method is called, it returns `true` only if the object is of class `HTMLVideoElement`. This method, along with its return value `true`, serves as a way to uniquely identify the object type.

```
class BasicShape {
public:
    virtual ShapeType GetType() const = 0;
};
class BasicShapeCircle : public BasicShape {
public:
  ShapeType GetType() const override {
    return kBasicShapeCircleType;
  }
};
class BasicShapeEllipse final : public BasicShape {
public:
   ShapeType GetType() const override {
    return kBasicShapeEllipseType;
   }
};
class BasicShapePolygon final : public BasicShape {
public:
   ShapeType GetType() const override {
        return kBasicShapePolygonType;
   }
};
```

Figure 6: An example in Category 2: the type information returned by a virtual method is overridden in each subclass to return a different enumeration constant.

4.2 Custom RTTI Identification

As the foundation of our solution, we need to construct a precise database that contains custom RTTI for a class hierarchy (i.e., a group of classes that share a base class), referred as *class signatures*, that can uniquely identify a class/type in the hierarchy. In other words, we will construct a map between a class signature and the actual (allocation) type. More specifically, we scan all the source code files (including header files) and look for class hierarchies that match the aforementioned three categories. For the first two categories, we check the following conditions at the syntax level:

1. a unique enumeration constant is either assigned to a member field of each class in the hierarchy, or returned by a virtual method;
2. if a member field is assigned with the constant, the assignment should happen inside the constructor and the field should not be modified once it is initialized.

For the third category, we use the following heuristics:

1. each class in the hierarchy overrides a unique method and changes its return value.
2. each class should have at least one overridden method that returns a unique value not seen in other classes in the hierarchy.

For each class hierarchy (classes that share a base class), we store the unique constant and method (collectively considered as the class signature) and its corresponding class type

```cpp
class CanvasImageSource {
public:
    virtual bool IsVideoElement() const { return false; }
    virtual bool IsCanvasElement() const { return false; }
    virtual bool IsVideoFrame() const { return false; }
};
class HTMLVideoElement : public CanvasImageSource {
public:
    bool IsVideoElement() const override { return true; }
};
class CanvasRenderingContextHost : public
 ↪  CanvasImageSource {
};
class HTMLCanvasElement : public
 ↪  CanvasRenderingContextHost {
public:
   bool IsCanvasElement() const override { return true; }
};
class VideoFrame : public CanvasImageSource {
    bool IsVideoFrame() const override { return true;}
}
```

Figure 7: An example in Category 3: type information is encoded as the return value of a custom type checking function that is overridden to return `true` in the corresponding subclass.

```cpp
1  /* Some code snippet for use of custom type information,
2  * uninteresting code lines are ommited.
3  * */
4  std::unique_ptr<Shape> Shape::CreateShape(const
    ↪  BasicShape* basic_shape) {
5    std::unique_ptr<Shape> shape;
6    switch (basic_shape->GetType()) {
7      case BasicShape::kBasicShapeCircleType: {
8        /*To<> is implemented as a static_cast<>*/
9        const BasicShapeCircle* circle =
          ↪  To<BasicShapeCircle>(basic_shape);
10       //...
11     }
12     case BasicShape::kBasicShapeEllipseType: {
13       const BasicShapeEllipse* ellipse =
          ↪  To<BasicShapeEllipse>(basic_shape);
14       //...
15     }
16     case BasicShape::kBasicShapePolygonType: {
17       const BasicShapePolygon* polygon =
          ↪  To<BasicShapePolygon>(basic_shape);
18       //...
19     }
20   }
21 }
```

Figure 8: The simplified code for class `BasicShape`.

in our database. Note that we find that a class hierarchy can sometimes have a subset of classes with signatures while the remaining classes do not have signatures by design. In other words, there may be a "sub-class-hierarchy" within a complete hierarchy that encodes class signatures. To accommodate such cases, we effectively look for such sub-class-hierarchies. As long as the heuristics described above apply to the sub-class-hierarchy, we still infer that it has encoded class signatures.

4.3 Custom Type Check Identification

Given the database, we can perform an analysis to find the type checks in the target program. In general, any statement that uses the custom type information to control the program flow is considered a type check. More specifically, we look for statements that are of the form of `type == c` (including `switch` cases), where the left-hand side can be any expression that evaluates to a previously-recognized type-indicating member variable or type-indicating method, and the right-hand side is a constant. Given that we have mapped each unique constant to a corresponding class type, we can tell exactly which type is checked for in the statement.

Figure 8 shows an example of type checks relating to class `BasicShape`, which is described in Figure 6; the type check compares `basic_shape->GetType()` against enumeration constants that we record in the database (e.g., `kBasicShapeCircleType`), and then jump to different program branches. T-PRUNIFY can identify each switch case as a type check, and determine that the type of object `basic_shape` is `BasicShapeCircle` between line 7 and line 11. We also discuss how Chrome sometimes uses custom C++ template expressions to perform type checks in §6.

Note that we find rare cases where a type check would not look like a straight equality comparison in the form of `type == c`. Instead, it can use inequality comparisons, e.g., >= or <=. We do not currently recognize such type checks and leave them as future work.

5 Static Safe Casting Verification

After constructing the database of custom type information and identifying the custom type checks in the program, the next step is to determine whether the type casts are actually safe, i.e., sufficiently protected by those checks. At a high level, given a type check, we analyze all the type cast statements that are dominated by the type check. If the destination type in the type cast statement is compatible with the checked type, we consider it a safe type cast.

There are several considerations in performing such an analysis. First, type casts do not always happen immediately after a type check, and the pointer used for the type check may not be the same one that is used for type casts. Therefore, we perform a flow-sensitive and field-sensitive intra-procedural pointer analysis to make sure that the pointer points to the same object at the type check and the type cast. For example, the casting at line 13 in Figure 9 is safe because `case1` and `base` point to the same object.

Second, the pointer analysis needs to consider different execution paths and summarize possible types along all execution paths. Therefore, T-PRUNIFY will only determine a casting as safe if all types are compatible with the destination

```
1  class Base;
2  class Sub : Base;
3  class SubSub: Base;
4  // operator isa<>() is an identified type check
5  void foo(Base *base, Base *other) {
6    Base *case1, *case2, *case3, case4;
7    Sub *sub; SubSub *subsub;
8    case1 = case2 = case3 = case4 = base;
9    if (isa<Sub*>(base)) {
10     // case1: check is done using base
11     // but casting is done using case1
12     // need to know base alias with case1
13     sub = static_cast<Sub*>(case1); // SAFE
14     if (other != nullptr) {
15       case2 = other;
16     }
17     // case2: may point to both 'other' of Base* type
18     // and base of Sub* type (due to the check)
19     // Base* is not compatible with Sub*
20     sub = static_cast<Sub*>(case2); // possibly UNSAFE
21     // case3: the check is insffucient
22     // previous check only indicate base if of Sub* type
23     // not compatible with SubSub*
24     subsub = static_cast<SubSub*>(case3); // possibly
         ↪  UNSAFE
25     if (isa<SubSub*>(base)) {
26       // case4: continuous refinement
27       // after this check, we know base is of
28       // SubSub* type
29       subsub = static_cast<SubSub*>(case4); // SAFE
30     }
31   }
32 }
```

Figure 9: Different corner cases need to be considered when proving when a type cast is safe.

type. For example, T-PRUNIFY cannot prove that the casting at line 20 in Figure 9 is safe. This is because `case2` *may* point to both an object of `Sub` type due to the type check at line 9, and an object of `Base` type due to the pointer reassignment at line 15. As not all aliased objects are compatible with the destination type `Sub*`, the casting is potentially unsafe and needs an additional runtime check.

Third, it is possible that the type check is insufficient in protecting subsequent type casts, e.g., the target of the type cast is not compatible with the type checked. Therefore, T-PRUNIFY does not blindly trust a custom type check; it also performs a type compatibility check to ensure that the refined type suggested by the type check is indeed compatible with the destination type of cast. If T-PRUNIFY cannot determine the casting is safe, it acts conservatively and will not eliminate a sanitizer check. This is because such cases can potentially be real type confusion bugs. For example, T-PRUNIFY cannot prove the casting at line 24 in Figure 9 as safe, because the check at line 9 only indicates `case3`, which is a must-alias with `base`, is of type `Sub*`, which is not compatible with type `SubSub*`.

Fourth, the analysis should consider multiple type checks that gradually narrow down the type to a more specific type (i.e., subclass). This also means the analysis should be flow-sensitive. For example, the casting at line 29 in Figure 9 is safe, because after the check at line 25, the type of `base` is further narrowed down to `SubSub*`. However, without a flow-sensitive analysis, one cannot be sure `base`'s type must be `SubSub*`.

Finally, type casts can be performed in a separate function from the function that performed the type check. For example, the type cast may happen in a callee of a caller that performs the type check. In such cases, we cannot conclude that the type cast is safe simply because one caller has performed a safe type check. Instead, we need to analyze all callers to make sure safe type checks are always present before the type cast. In our current design, we perform only an intra-procedural analysis that makes sure the type cast happens in the same function that performs the type check. We consider this a conservative solution and will extend it to the inter-procedural case in the future.

In the end, once T-PRUNIFY finds a safe casting, it will inform an existing type confusion sanitizer not to emit redundant type checks during the compilation.

6 Implementation

In this section, we describe several key implementation details. Overall, we implement T-PRUNIFY on top of the libclang library and LLVM (v14.0.5). The implementation consists of 9,652 lines of code in total. Specifically, we implement the component "custom runtime type information inference" using libclang by analyzing the source code of the target program (as certain information like the C++ class hierarchies is preserved better at the source code level). We implement the component "static safe casting verification" using LLVM passes as LLVM is more appropriate for pointer analysis. Since our analysis spans source code and LLVM IR, we need to pass intermediate analysis results from source code level to the IR level, which we will describe in this section. In addition, we modified a state-of-the-art type confusion sanitizer, HexType [24], to facilitate the evaluation of T-PRUNIFY.

Class Hierarchy Construction. The C++ compiler front-end like Clang can accurately parse the C++ class hierarchy. However, as the LLVM IR language is generic (i.e., needs to support different source languages) and is relatively low level, the C++ class hierarchies are not explicitly stored at the IR level. Therefore, we implement a Clang plugin to store the C++ class hierarchies (i.e., inheritance relations and type compatibility) and store them for further use.

Class Signature Database Building. This part is done by analyzing the source code of a target program directly (instead of LLVM IR). Specifically, we use libclang's python binding. Besides missing the class hierarchy information at the LLVM

IR level, another reason for source code analysis is that enumeration constants at C/C++ level will be lowered to integer constants thus losing their semantic information and become indistinguishable from other integer constants. libclang allows the user to iterate the abstract syntax tree (AST) to get compilation time information (e.g., the type of the variable, the name of the variable, the type of the functions). Using libclang, we iterate through the AST to (1) extract enumeration constants used to assist identification of custom runtime type information (see §4 for details), and (2) record the use of these constants (e.g., assignment, comparison, return value).

Feeding Source Code Analysis Results to the LLVM Analysis. The class signature database stores variable names, method names, and enumeration constants to assist identification of custom type checks. However, at the LLVM IR level, names of C++ virtual methods are available only in type definitions. At method call sites, virtual methods will be lowered to indirect calls, thus losing the source code level semantics. For example, a simple method call of `basic_shape->GetType()` would look like the following in LLVM IR (simplified for reading):

```
%41 = getelementptr (%"class.blink::BasicShape"*)** %40,
%42 = load i32 (%"class.blink::BasicShape"*)** %41
%43 = call noundef i32 %42(%"class.blink::BasicShape"* %0)
  switch i32 %43, label %342 [
  ...
```

To overcome this issue, we modified the Clang++ front-end to annotate LLVM IR with method names. In the above example, our annotation would label `%42` as `GetType()`. The IR snippet is shown as:

```
@.str.2 = "blink::BasicShape::GetType"
%vfn = getelementptr (%"class.blink::BasicShape"*)** %vtable,
%10 = load i32 (%"class.blink::BasicShape"*)** %vfn
%call31 = call noundef i32 %10(%basic_shape)
%11 = call i32 @llvm.annotation(i32 %call31, (@.str.2))
 switch i32 %call31, label %sw.default [
 ...
```

The `%vtable` is the equivalent pointer as `%40` before the annotation.

Manually-Summarized Type Checks We find that some subsystems of Chrome choose to use custom C++ templates to implement type checks. For example, we have seen `IsA<T>` frequently which operates as a type check for type `T`. Behind the template, different classes can choose to implement it differently. Since our solution to recognize type checks is by analyzing the source code, we currently recognize these statements specifically through manually-curated domain knowledge. Technically, we could pre-process the source code into a version without templates and then perform our follow-up analysis. However, due to implementation issues, we were not able to succeed at this point. We leave this as our future work.

7 Evaluation

To assess the extent of custom Run-Time Type Information (RTTI) usage and the effectiveness of T-Prunify, we conducted a comprehensive examination of open-source C/C++ software. Subsequently, we carried out a systematic evaluation with the aim of addressing the following research inquiries:

- **RQ1:** How prevalent is custom RTTI?
- **RQ2:** How many casts can be classified by T-Prunify as safe casts out of all the downcast operations?
- **RQ3:** How accurate is T-Prunify in identifying classes with custom RTTI, type checks, and safe casts? In other words, we measure the false negatives and false positives of T-Prunify.
- **RQ4:** How much runtime overhead can T-Prunify improve by pruning unnecessary sanitzier checks?

Experimental Setting. All evaluations are conducted on machines equipped with Intel(R) Xeon(R) Gold 6248 CPU processors and 1024GB RAM, running on a 64-bit Ubuntu 18.04.6 LTS operating system.

Evaluation Target and Experimental Setup. To investigate the prevalence of custom RTTI, we sampled open sourced C/C++ programs with large code bases. Then, we applied T-Prunify to the following programs: the SPEC CPU2006 C++ program, Chromium and the LLVM toolchain. We chose Chromium and the LLVM toolchain as evaluation targets because they are complex, large-scale, and well-engineered pieces of software. If we show that our analysis if effective on them, we argue that it should also work on other targets, as long as the project leverages developer-inserted custom runtime type information (RTTI) to avoid unsafe castings.

In addition, unlike smaller programs that may not have complex class hierarchies (and hence, few developer-implemented type checks), according to our analysis, both Chromium and LLVM source code indeed have many complex class hierarchies and a large number of developer-implemented type checks. Therefore, we believe they represent ideal benchmarks to validate the ideas proposed in this paper. We believe that other large-scale programs would also similarly benefit from our solution.

We design experiments by compiling those programs into three versions: the original program (original) without any instrumentation, the fully HexType-instrumented program (`program-hextype`), and the program with reduced instrumentation after applying T-Prunify (`program- T-Prunify`). For `program-hexytpe` and `program- T-Prunify`, we did not instrument libc++ as the standard C++ library because (1) we consider it as safe and (2) it does not include any custom RTTI.

HexType Configurations. The original HexType [24] was implemented based on llvm-3.9.0, which is no longer compatible with the Chromium version we tested. Therefore, we ported it to llvm-14.0.5, which could be used to compile the target Chromium we evaluated. When assessing the overhead, we considered all the optimizations implemented by HexType, including the elimination of checks for safe casts that can be verified during compilation time. We also apply this HexType version when evaluating SPEC CPU2006 and the LLVM toolchain.

Benchmarks. In order to effectively showcase the performance improvements achieved through the use of T-PRUNIFY, we provide further details on the benchmarks used in our experiments. SPEC CPU is a well-established benchmark, and we use the running time to represent the performance. For Chromium, we chose three different benchmarks, namely Speedometer, JetStream2, and Motion Mark [4] to exercise different parts of a browser. Speedometer is a benchmark that measures the responsiveness of web applications by simulating user interactions with the browser (e.g., DOM manipulation). JetStream2 is a comprehensive benchmark suite that measures the performance of JavaScript and WebAssembly in advanced web applications. It consists of a variety of tests, including latency and throughput tests, that cover a wide range of web application use cases. Finally, Motion Mark is a benchmark designed to thoroughly test the graphics systems of web browsers. This benchmark includes a variety of subtests, including the CSS, image and text rendering, that assess the performance of the browser's rendering capabilities. For LLVM toolchains, we use the Linux kernel v6.5 under `allyesconfig` as the compilation target.

7.1 Prevalence of custom RTTI

We selected seven C/C++ open sourced software projects with large codebases. As shown in Table 1, we observed that 4 out of 7 projects had type confusion CVEs, implying common uses of type casts. In addition, we randomly sampled 10 class hierarchies for each project and found the use of custom RTTI in 6 of the 7 projects. In summary, all six projects demonstrated a substantial utilization of custom RTTI. This result underscores the prevalence of custom RTTI in C/C++ software. For Chromium, we find 8 out of the 10 sampled class hierarchies were found to contain custom RTTI; for Firefox and the LLVM toolchain, 6 out of 10 sampled class hierarchies include custom RTTI. In the case of JavaScriptCore, although only 3 out of 10 sampled classes contain custom RTTI, some are located in important modules, e.g., the DOM module.

7.2 Coverage

In this section, we will first show the overall results of analyzing the three projects, based on the results, we calculate the **coverage**, which measures out of all the downcasts, how many of them are identified as the safe casts.

Table 1: Prevalence of the custom RTTI in large scale C/C++ software.

Software	TypeConfusion CVE	Custom RTTI
Chromium	Y	Y (8/10)
Mozilla Firefox	Y	Y (6/10)
Hermers	Y	Y (7/10)
JavaScriptCore	Y	Y (3/10)
LLVM ToolChain	N	Y (6/10)
QT	N	Y (5/10)
Boost	N	N

The overall result is shown in Table 3, using Chromium as an example; there is a total of 54,617 classes that are part of class hierarchies, out of which 6,671 are base classes, forming class hierarchies. We observe that all downcasts occur within 1,123 of these class hierarchies and a total of 5,160 classes appear as downcast targets.

Among the 1,123 class hierarchies, we identify 719 hierarchies, and 3,585 classes within these hierarchies, that have custom RTTI. Furthermore, we find 827 classes in total that have both RTTI and appear as downcast targets. As we can see, many classes have custom RTTI but are never used as downcast targets. Upon inspecting some such cases, we find that their custom RTTI is used in scenarios like serialization and deserialization [1] or logging [7].

Finally, we find 49,364 downcast operations in Chromium, and 23,721 of them have destination types with custom RTTI. A subset of these downcasts, i.e., 6,704, are determined to be safe casts. In other words, T-PRUNIFY finds these downcasts are protected by developer-inserted type checks. Overall, this represents a significant fraction of downcasts that can be exempt from sanitizer checks.

Since 6,704 out of 49,304 downcasts in Chromium are identified as safe casts by T-PRUNIFY, the coverage is $6,704/49,304 = 13.58\%$. Similarly, the coverages for SPEC CPU xalancbmk and the LLVM toolchain are 9.82% and 9.49%, respectively. Although the static coverage appears low, the dynamic type checking reduction is far more significant, which we will show in §7.4. Since T-PRUNIFY does not find custom RTTI in the other six SPEC CPU C++ programs, we only list the results for xalancbmk here.

7.3 Accuracy

In this section, we selected Chromium as our test subject to evaluate the outcomes at each critical step. These steps include custom RTTI identification, type check identification, and safe casts identification. Then, we manually curated the ground truth of the above steps relating to the top 50 classes (that appeared as destination types of downcasts). Using this dataset, we then analyzed the accuracy of the results produced by our system, specifically looking for any false positives or false negatives.

Custom RTTI Identification We collected the top 50 downcast targets and list them in the Table 2. We found that 25 of the top 50 classes had custom RTTI encoded, and our approaches correctly identified 20 of the 25, resulting in 5 false negatives. We are unable to infer the custom type info encoded in 5 classes primarily due to the patterns that we currently do not recognize, as described in §4.1. For instance, one of them is the class `v8:Uint32` which has a member function `IsUint32()` that returns `true` if the object is of class `v8::Uint32`. However, this function examines whether the object value is within the range of [0, kMaxUInt32]. Among the 20 classes that are identified to have custom RTTI by T-PRUNIFY, we find no false positives; this is expected as our analysis is conservative.

Type Check Identification. In terms of false negatives in this step, since T-PRUNIFY failed to identify the RTTI of 5 classes, it will therefore automatically miss type checks relating to these 5 classes. To evaluate whether our safe cast identification will miss any additional cases, we sample 54 type checks from the remaining 20 classes with custom RTTI. For most classes, we sampled three checks per class (however, note that there are cases where we can find only one type check). The results show that T-PRUNIFY can identify all 54 type checks, meaning no false negatives. To evaluate false positives, we sample 50 type checks that are reported by T-PRUNIFY, and all of them are true positives.

Safe Cast Identification. We follow a similar approach described above to evaluate the false negatives and false positives of safe cast identification. The same 54 sampled type checks are also in fact safe casts, according to our manual analysis. T-PRUNIFY successfully identifies 51 to be safe casts, missing the remaining 3 because of the lack of an inter-procedural analysis, i.e., the check is performed in a caller function but the cast happened in a callee. So, we have 3 false negatives in this data set. In addition, we sampled some cases to see whether the lack of inter-procedural analysis is the only reason. In particular, we find one false negative even when the check and cast are in the same function. The example is located in the v8 submodule: a base class `BaseSpace` which has two subclasses `NormalPageSpace` and `LargePageSpace`. Before casting an object `space` to type `NormalPageSpace`, the developer performed a type check `!(space.is_large())`. This case constitutes a safe cast. However, T-PRUNIFY failed to identify it because it did not take into account the fact that there are only two possible types, `!is_large()` effectively indicates that `space` is of type `NormalPageSpace`. Finally, we also collect 3 patches that fix type confusion vulnerabilities in Chrome with developer-implemented type checks and they are all identified by T-PRUNIFY, i.e., no false negatives. To evaluate false positives, we sample 50 safe casts reported by T-PRUNIFY from six different submodules and find none of them are false positives. Overall, the results exhibit a high accuracy, with

Table 2: The top 50 downcast classes targets in the Chromium, `# of cast` is the frequency, `Type Info?` is the ground truth whether the class has encoded some form of the custom type information. The last column shows whether T-PRUNIFY captured those information into our database.

Cast to	# of cast	Type Info?	T-PRUNIFY Captured?
v8::FunctionTemplate	3117	Y	Y
v8::Object	2974	Y	Y
v8::Int32	2735	Y	N
blink::Element	1973	Y	Y
llvm::Constant	1541	N	N
blink::EventTarget	1507	N	N
v8::Uint32	1343	Y	N
llvm::Instruction	971	N	N
v8::Number	827	Y	Y
llvm::Function	806	N	N
blink::HTMLElement	716	Y	Y
blink::WebGLRenderingContextBase	476	N	N
blink::WebGL2RenderingContextBase	445	N	N
perfetto::trace_processor::TypedColumn	421	N	N
blink::LocalFrame	405	Y	Y
blink::LocalDOMWindow	401	Y	Y
blink::DOMWindow	397	N	N
v8::Boolean	395	Y	Y
blink::WebGLUniformLocation	374	N	N
v8::internal::Isolate	312	N	N
v8::Array	304	N	N
skjson::ObjectValue	294	N	N
v8::JSVisitor	286	N	N
GrGpuResource	276	N	N
blink::UniqueElementData	273	Y	Y
llvm::cl::OptionValueCopy	273	N	N
v8::String	273	Y	Y
blink::ShareableElementData	267	Y	Y
blink::JSBasedEventListener	241	Y	Y
blink::CSSPrimitiveValue	231	Y	Y
blink::SVGElement	225	Y	Y
tint::sem::Vector	213	N	N
blink::LayoutBoxModelObject	208	Y	Y
blink::LayoutBlockFlow	203	Y	N
v8::internal::compiler::HeapObjectData	196	N	N
blink::Longhand	195	Y	N
blink::LayoutBox	177	Y	N
ppapi::PPB_Graphics3D_Shared	166	N	N
blink::Node	153	N	N
content::WebContentsImpl	149	N	N
base::DictionaryValue	148	N	N
blink::Document	145	Y	Y
llvm::StructType	143	N	N
blink::HTMLInputElement	142	Y	Y
blink::NGPhysicalBoxFragment	140	Y	Y
blink::HTMLCanvasElement	135	Y	Y
blink::TransformPaintPropertyNode	134	N	N
llvm::GlobalValue	126	N	N
llvm::MDString	121	N	N
blink::JSEventHandler	118	Y	Y

Table 3: Overall statistics of the results.

# of	Chromium	LLVM	xalancbmk
class hierarchies	6, 671	934	86
classes in hierarchies	54,617	8,842	825
class hierarchies with downcasts	1,123	244	7
classes as downcast targets	5,160	2,537	59
class hierarchies w/ custom RTTI found	719	183	3
classes w/ custom RTTI found	3,585	1,404	38
classes w/ custom RTTI & as downcast targets	827	1,064	19
downcast ops	49,364	211,571	560
downcast ops where destination types w/ RTTI	23,721	161,442	192
downcast ops with type checks (safe casts)	6,704	30,027	55

no false positive and very few false negatives in each steps.

7.4 Runtime Overhead Reduction

So far, we have evaluated the results of T-PRUNIFY statically, e.g., the effectiveness of T-PRUNIFY in terms of the statically-identified safe casts. In this section, we will measure the runtime overhead achieved by T-PRUNIFY compared to the state-of-the-art solution, HexType. As mentioned, we evaluate T-PRUNIFY in three programs: xalancbmk in SPEC CPU 2006, Chromium and LLVM toolchain. When evaluating Chromium, we utilize three widely recognized web browser benchmarks. As for the LLVM toolchain, we assess its performance by compiling the Linux kernel version 6.5 with the `allyesconfig` configuration. We tested each benchmark six times back-to-back, and eliminate the first run as it would be a cold start. Then we use the box plot to visualize the result of the last five run for programs with some deviations. We use the median values as the representative numbers to represent the overhead; the results are depicted in Table 4. Further, we calculate the type checks that instrumented by HexType and T-PRUNIFY when running those benchmarks; the results are shown in Table 6.

Results for xalancbmk. We measured the performance with five runs. As seen in Figure 10a, the median of the baseline running time (without any instrumentation) is 158s. When compiled with HexType, the median running time increases to 163s, which translates into an overhead of 1.03×. When compiled with T-PRUNIFY, the median running time is 161s, resulting in an overhead of 1.02×. Interestingly, we do observe the program compiled with T-PRUNIFY experiences significantly fewer type checks, i.e., from 283 million to 80 million. Yet the end-to-end overhead reduction is not pronounced in this benchmark. This is attributed to the fact that these type checks constitute a relatively minor portion of the total execution time.

Results for Chromium. To compute the runtime overhead for Chromium, we rely on the "scores" produced by each benchmark for each Chromium, Chromium-hextype, and Chromium- T-PRUNIFY, respectively. Note that these scores from different benchmarks may consist of different metrics,

Table 4: Overhead improvement for three projects relative to their respective benchmarks, the improvement is calculated based on the HexType instrumentation.

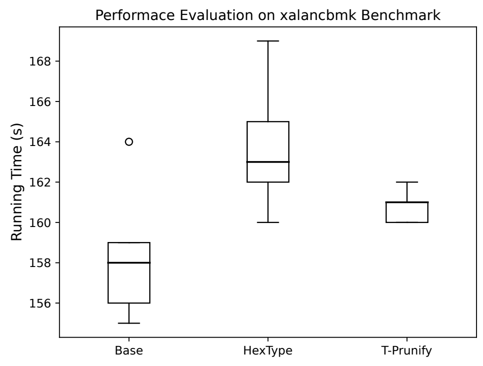
Software	Benchmark	Hextype	T-PRUNIFY
SPEC CPU	xalancbmk	1.03×	1.02×
Chromium	Speedometer	1.11×	1.08×
Chromium	JetStream2	1.22×	1.05×
Chromium	MotionMark	2.92×	1.51×
LLVM	Linux	16.7×	10.5×

e.g., throughput. Nevertheless, we assume these scores captured the most suitable metrics as intended by each benchmark, where high score shows better performance.

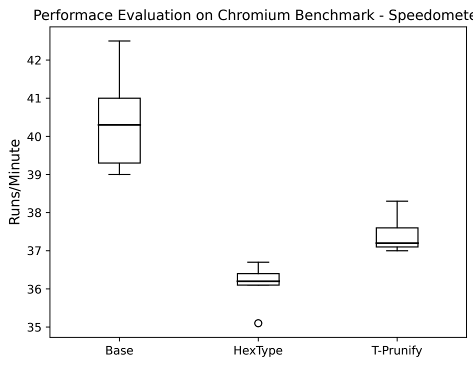
For Speedometer, we see that the original Chromium can achieve 40.3 runs/min, while chrome-hextype and chrome-reduced achieved 36.2 runs/min and 37.2 runs/min, respectively. Overall, Chromium-hextype experiences a relatively low overhead on the Speedometer benchmark to begin with, i.e., 10.2%, indicating that the exercised sanitizer-inserted checks have a relatively low proportion. Nevertheless, T-PRUNIFY manages to reduce the overhead from 10.2% ($= 1 - \frac{36.2}{40.3}$) to 7.7% ($= 1 - \frac{37.2}{40.3}$), representing a 25% relative reduction. In Table 6, we also report the total number of type checks that are executed at runtime with HexType and T-PRUNIFY. We can see that T-PRUNIFY successfully eliminated a large number of checks that would otherwise be performed by HexType. The box plot for five runs is shown in Figure 10b.

In JetStream2, the overall score of the original Chromium is 68.6, while Chromium-hextype and Chromium-reduced achieved scores of 56.3 and 65.6 respectively. The improvement is significant, from 17.9% ($= 1 - \frac{56.3}{68.6}$) to 4.4% ($= 1 - \frac{65.6}{68.6}$), representing a 75% relative reduction in overhead. In fact, the score of Chromium-reduced is very close to that of the uninstrumented Chromium. Looking at Table 6, we can see that the majority of the sanitizer checks are pruned, i.e., 2,800M out of 3,795M. We believe that the observed high ratio can be attributed to the predominant use of the V8 engine in the exercised code paths by JetStream2. The V8 engine has a history of being susceptible to numerous type confusion bugs [14–17]. Consequently, developers have inserted a significant number of type checks as a precautionary measure to mitigate potential vulnerabilities. The box plot for five runs is shown in Figure 10c.

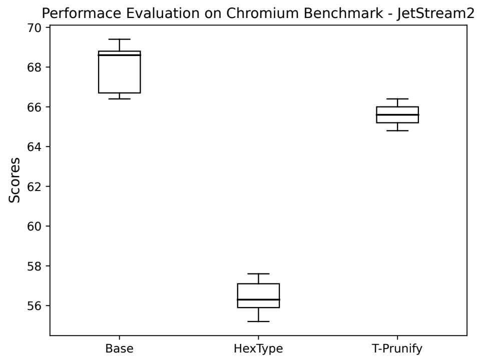
In MotionMark, the overall score of the original Chromium is 72.31, while Chromium-hextype and Chromium-reduced achieved 24.75 and 47.78, respectively. The improvement of results is the most significant out of the three benchmarks, i.e., from 65.8% ($= 1 - \frac{24.75}{72.31}$) to 33.9% ($= 1 - \frac{47.78}{72.31}$), representing a 48.5% relative reduction in overhead. The box plot for



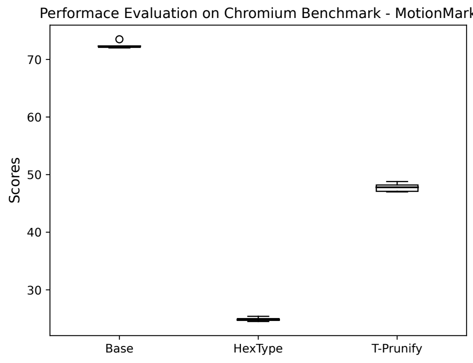
(a) The box plot for performance comparison among three configurations in `xalancbmk` benchmark. Lower running time means higher performance.



(b) The box plot for performance comparison among three configurations in `Speedometer` benchmark. Higher number means better performance.



(c) The box plot for performance comparison among three configurations in `JetStream2` benchmark. Higher score means better performance.



(d) The box plot for performance comparison among three configurations in `MotionMark` benchmark. Higher score means better performance.

Figure 10: Box plots for `xalancbmk` and Chromium benchmarks. Figure (a) is the boxplot for `xalancbmk`, lower running time means higher performance. Figures (b), (c) and (d) shows the benchmarks for Chromium, higher score means better performance.

five runs is shown in Figure 10d.

According to Table 6, T-PRUNIFY results in 327M fewer sanitizer checks at runtime. The significant number of pruned checks in this benchmark is due to the many rendering elements such as SVG node and HTML elements with CSS style. Many of these classes have custom RTTI and safe casts (as some of our examples showed in §2).

Results for LLVM Toolchain. Next, we calculate the runtime overhead for the LLVM toolchain. Specifically, we use it to compile the Linux kernel v6.5 with `allyesconfig` and measure the compilation time to represent the performance. Overall, we find that HexType takes 369 mins to finish the compilation, while T-PRUNIFY takes only 231 mins. We conducted five runs in this experiment and dropped the first run, since four runs yield identical running times measured in minutes, we do not include the box plot here. The compilation speed up is shown in Table 4, we reduced the HexType's 16.2× overhead to 10.5×, representing a 35.19% overhead improvement. This significant improvement can be attributed to the 743B (46.82%) fewer sanitizer checks at runtime.

Compilation overhead Besides the run time overhead, we also log the compile time overheads that T-PRUNIFY brings to compile the Chromium and LLVM Toolchain; the results are shown in Table 5. Compared to HexType, the compilation time is shorter, which means the analysis performed by T-PRUNIFY is relatively lightweight and saves significant instrumentation workload. The difference in compilation time between HexType and T-PRUNIFY can be attributed to two reasons. First, HexType inserts sanitizers' type check functions in the front end. Second, in the optimization phase, HexType analyzes each inserted type check function, extracting the source type and target type to perform the compilation time safe casts validation. T-Prunify can help reduce the overheads from both operations because fewer sanitizer's type check functions are inserted in the first place.

In summary, we demonstrate that in each of the benchmarks, T-PRUNIFY significantly reduces the overhead of redundant sanitizers' type checks at runtime, even though only about 10% of the type-cast checks are pruned statically as described in §7.2. This is likely due to such redundant

Table 5: Compilation time comparison.

Program	original	HexType	T-PRUNIFY
Chromium (-j32)	38min	59min	56min
LLVM Toolchain (-j32)	13min	67min	39min

Table 6: Number of dynamic cast verification performed by HexType versus TPRunify.

Benchmark	Hextype	T-PRUNIFY	Reduced
Chromium-Speedometer	1, 558 M	241 M	1, 317 M (84.53%)
Chromium-JetStream2	3, 795 M	995 M	2, 800 M (73.78%)
Chromium-MotionMark	502 M	175 M	327 M (65.24%)
xalancbmk	283 M	80 M	203 M (71.73%)
LLVM-compile-Linux	1, 587 B	844 B	743 B (46.82%)

checks being frequently executed at runtime, and these checks being expensive to execute.

8 Limitations and Extensions

While our experimental results are quite promising, we did encounter some practical limitations of T-PRUNIFY while manually assessing accuracy. In the custom RTTI identification phase, we discovered cases where a wide range of constants or even structure pointers were used to assist encoding custom RTTI information, patterns not handled by our current approach. These cases could be addressed in future work by doing deeper semantic analysis during the RTTI identification phase. It is also interesting to investigate the encoding patterns of custom RTTI across open-source projects beyond Chrome. Additionally, our safe cast analysis is currently intra-procedural, which means that it may overlook certain safe casts that occur across different function calls. In future work we plan to make this analysis inter-procedural, thereby supporting cases where a type check occurs in a caller while the cast occurs in the callee.

Beyond the aforementioned improvements, there are multiple fruitful ways to broaden and extend this work. First, we believe there are uses for automatic detection of custom RTTI schemes beyond reducing sanitizer overhead. For example, in cases where casts are not protected by a type check, one could automatically insert type checks into the code using the custom RTTI found by our technique. Also, we believe it would be worthwhile to extend our RTTI detection approach could to support `structs` in the C language—there also, custom RTTI has been used to ensure cast safety [2].

9 Related Works

In this section, we compare T-PRUNIFY with closely related work in three areas.

Type Confusion Sanitizers. As mentioned before, type confusion sanitizers aim to detect bad castings that can introduce confusion vulnerabilities, by instrumenting the target program with additional runtime checks. Undefined Behavior sanitizer (UBSan) [31] is one of the earliest available type confusion sanitizers. It relies on the standard C++ RTTI to perform type compatibility check. As a result, it does not support non-polymorphic classes and may introduce crashes [26]. Recently, Clang CFI [30] also added support for detecting type confusing bugs by leveraging the standard C++ RTTI. CaVer [26] aims to address two main issues of the standard C++ RTTI: (1) it improves the speed of type checking by using unique hashes instead of mangled names, and by including all compatible types in a single RTTI entry, instead of requiring traversing the class hierarchy; (2) it supports non-polymorphic classes by using a decoupled lookup table to find RTTI associated with a memory object. TypeSan [22] and HexType [24] further improve the performance and coverage over CaVer by using lower cost data structures, caching, and by expanding the instrumentation targets. EffectiveSan [19] can also detect type confusion vulnerabilities, besides other memory errors like out-of-bound and use-after-free. EffectiveSan uses low-fat pointer [18, 20, 25] to achieve efficient metadata access. It also supports checking casts between primitive types. Bitype [27] further compress the metadata structure into bit stream and speedup the compatibility check via `xor` operations. T-PRUNIFY is orthogonal and complementary to these type confusion sanitizers as our goal is to leverage developer-inserted custom type checks to eliminate redundant checks induced by type sanitizers. Therefore, T-PRUNIFY can be combined with any of these type sanitizers.

Optimizing Sanitizer Checks. There are also some work to reduce the sanitizers' overhead, these work could be divided into two categories. The first is using sanitizer-specific static analysis to remove only semantically redundant checks, for example, RedCard [21] is designed to use static analysis to reduce the redundant instrumentation for dynamic race conditions. Similarly, DataGuard [23] uses a set of sophisticated static analyses to prove the safety of stack objects and migrate them to safe stack, thus reducing the runtime protection overhead. Furthermore, WPBound [29] utilizes value range analysis to effectively minimize the number of out-of-bound memory checks inserted by sanitizers. Besides static analysis, SIMBER [33] incorporates statistical inferences to identify redundant bound checks. Another approach develops the framework and use general heuristics to remove costly sanitizer checks irrespective their semantics, those work includes ASAP [32] and SanRazor [34]. ASAP [32] allows developers to specify the acceptable percentage of runtime overhead based on their resource constraints. Leveraging this information, ASAP automatically instruments the program to maximize the security promise within the given budget. While SanRazor [34] combines runtime profiling and static analysis to identify and eliminate repeated and redundant checks, thereby optimizing computing resources. T-PRUNIFY falls within the first category, but with a specific focus on type confusion sanitizers.

10 Conclusion

Type confusion vulnerabilities are severe security threats to C/C++ programs. Due to the high performance overhead of the standard C++ runtime type information (RTTI), in large complex C++ projects like the Chrome browser, developers usually introduce custom RTTI and type checks to prevent type confusion bugs. Based on this observation, we implemented T-PRUNIFY, a tool that can automatically identify developer-inserted type checks and leverage these checks to validate the safety of type casts. Applying T-PRUNIFY to the Chrome browser allows us to identify a large number, i.e., 6,704, of safe casts. Leveraging this information, T-PRUNIFY can help remove redundant type casting checks induced by type confusion sanitizers like HexType and reduce the corresponding performance overhead by 25% to 75%.

Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. And many thanks to our shepherd who paid a lot of patience and effort to check our revision. This research was partially sponsored by the U.S. Army Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. It was also partially supported by NSF grant #1801534, #1652954, #1953933, #2155213 and #2046026.

References

[1] flattenable_is_valid_as_child(). https://github.com/google/skia/blob/main/src/core/SkRuntimeEffect.cpp#L371.

[2] Linux Commit f306dff7. https://github.com/torvalds/linux/commit/17cfe79a65f98abe535261856c5aef14f306dff7, 2018.

[3] CWE-843: Access of Resource Using Incompatible Type ('Type Confusion'). https://cwe.mitre.org/data/definitions/843.html, 2022.

[4] Browser Benchmarks. https://browserbench.org, 2023.

[5] [compiler] fix bug in representation-changer::getword32representationfor. https://chromium.googlesource.com/v8/v8/+/fd29e246f65a7cee130e72cd10f618f3b82af232%5E%21/#F0, 2023.

[6] [parser] fix eval tracking. https://chromium.googlesource.com/v8/v8/+/a4aece44c60ea1be4699667dbd27403574520df0%5E%21/#F1, 2023.

[7] v8/src/objects/js-objects.cc. https://source.chromium.org/chromium/chromium/src/+/main:v8/src/objects/js-objects.cc;l=2865?q=JSObject::JSObjectShortPrint, 2023.

[8] [wasm] refine installation of the webassembly.exception constructor. https://chromium.googlesource.com/v8/v8/+/c0614e9bcef7266d2e4544602d668c01b5dcaa37%5E%21/#F0, 2023.

[9] Satish Chandra and Thomas Reps. Physical type checking for c. In *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 66–75, 1999.

[10] National Vulnerability Database. CVE-2015-3077. https://nvd.nist.gov/vuln/detail/CVE-2015-3077, 2015.

[11] National Vulnerability Database. CVE-2021-21224. https://nvd.nist.gov/vuln/detail/CVE-2021-21224, 2021.

[12] National Vulnerability Database. CVE-2021-30561. https://nvd.nist.gov/vuln/detail/CVE-2021-30561, 2021.

[13] National Vulnerability Database. CVE-2022-1486. https://nvd.nist.gov/vuln/detail/CVE-2022-1486, 2022.

[14] National Vulnerability Database. CVE-2023-0473. https://nvd.nist.gov/vuln/detail/CVE-2023-0473, 2023.

[15] National Vulnerability Database. CVE-2023-0703. https://nvd.nist.gov/vuln/detail/CVE-2023-0703, 2023.

[16] National Vulnerability Database. CVE-2023-1215. https://nvd.nist.gov/vuln/detail/CVE-2023-1215, 2023.

[17] National Vulnerability Database. CVE-2023-1235. https://nvd.nist.gov/vuln/detail/CVE-2023-1235, 2023.

[18] Gregory J Duck and Roland HC Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction (CC)*, pages 132–142, 2016.

[19] Gregory J Duck and Roland HC Yap. Effectivesan: type and memory error detection using dynamically typed c/c++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–195, 2018.

[20] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. Stack bounds protection with low fat pointers. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[21] Cormac Flanagan and Stephen N Freund. Redcard: Redundant check elimination for dynamic race detectors. In *European Conference on Object-Oriented Programming*, pages 255–280. Springer, 2013.

[22] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. Typesan: Practical type confusion detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 517–528, 2016.

[23] Kaiming Huang, Yongzhe Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. The taming of the stack: Isolating stack data from memory errors. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, page 17, 2022.

[24] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. Hextype: Efficient detection of type confusion errors for c++. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2373–2387, 2017.

[25] Albert Kwon, Udit Dhawan, Jonathan M Smith, Thomas F Knight Jr, and Andre DeHon. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security (CCS)*, pages 721–732, 2013.

[26] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. Type casting verification: Stopping an emerging attack vector. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 81–96, 2015.

[27] Chengbin Pang, Yunlan Du, Bing Mao, and Shanqing Guo. Mapping to bits: Efficiently detecting type confusion errors. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 518–528, 2018.

[28] Bjarne Stroustrup. Multiple inheritance for c++. *Computing Systems*, 2(4):367–395, 1989.

[29] Yulei Sui, Ding Ye, Yu Su, and Jingling Xue. Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions. *IEEE Transactions on Reliability*, 65(4):1682–1699, 2016.

[30] The Clang Team. Clang Control Flow Integrity. https://clang.llvm.org/docs/ControlFlowIntegrity.html, 2022.

[31] The Clang Team. UndefinedBehaviorSanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2022.

[32] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High system-code security with low overhead. In *2015 IEEE Symposium on Security and Privacy*, pages 866–879. IEEE, 2015.

[33] Hongfa Xue, Yurong Chen, Fan Yao, Yongbo Li, Tian Lan, and Guru Venkataramani. Simber: Eliminating redundant memory bound checks via statistical inference. In *ICT Systems Security and Privacy Protection: 32nd IFIP TC 11 International Conference, SEC 2017, Rome, Italy, May 29-31, 2017, Proceedings 32*, pages 413–426. Springer, 2017.

[34] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. {SANRAZOR}: Reducing redundant sanitizer checks in {C/C++} programs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 479–494, 2021.