type++: Prohibiting Type Confusion With Inline Type Information

Nicolas Badoux
EPFL
nicolas.badoux@epfl.ch

Flavio Toffalini[†]
Ruhr-Universität Bochum; EPFL
flavio.toffalini@rub.de

Yuseok Jeon
UNIST
ysjeon@unist.ac.kr

Mathias Payer
EPFL
mathias.payer@nebelwelt.net

***Abstract*—Type confusion, or bad casting, is a common C++ attack vector. Such vulnerabilities cause a program to interpret an object as belonging to a different type, enabling powerful attacks, like control-flow hijacking. C++ restricts runtime checks to polymorphic classes because only those have inline type information. The lack of runtime type information throughout an object's lifetime makes it challenging to enforce continuous checks and thereby prevent type confusion during downcasting. Current solutions either record type information for all objects disjointly, incurring prohibitive runtime overhead, or restrict protection to a fraction of all objects.**

Our C++ dialect, type++, enforces the paradigm that each allocated object involved in downcasting carries type information throughout its lifetime, ensuring correctness by enabling type checks wherever and whenever necessary. As not just polymorphic objects but all objects are typed, all down-to casts can now be dynamically verified. Compared to existing solutions, our strategy greatly reduces runtime cost and enables type++ usage both during testing and as mitigation. Targeting SPEC CPU2006 and CPU2017, we compile and run 2,040 kLoC, while changing only 314 LoC. To help developers, our static analysis warns where code changes in target programs may be necessary. Running the compiled benchmarks results in negligible performance overhead (1.19% on SPEC CPU2006 and 0.82% on SPEC CPU2017) verifying a total of 90B casts (compared to 3.8B for the state-of-the-art, a 23× improvement). type++ discovers 122 type confusion issues in the SPEC CPU benchmarks among which 14 are new. Targeting Chromium, we change 229 LoC to protect 94.6% of the classes that could be involved in downcasting vulnerabilities, while incurring only 0.98% runtime overhead over the baseline.

I. Introduction

The C++ language provides high-performance and object-oriented abstraction capabilities. C++ organizes objects in parent-child relationships, in which child classes inherit (and extend) attributes and functions from their parents. Ideally, any new class represents a new type in the program, thus designing flexible software architectures. Casting operations enable more modular software development as objects can be passed to functions defined for another type. Unfortunately, C++ does not enforce type safety and cannot check the correctness of all cast operations due to limited type information available at runtime, resulting in the risk of type confusion. Type confusion happens when the program interprets an object as belonging to a different type. In these attacks, fields may be interpreted as different types (*e.g.,* an `int` misinterpreted as a pointer) or invoking unexpected virtual functions. Type confusion vulnerabilities in C++ applications are a building block to mount code-reuse [1] and data-only attacks [2] in a wide range of software products, such as Telegram (CVE-2021-31318), Firefox (CVE-2023-25736, CVE-2023-25737), or Google Chromium (CVE-2019-5757, CVE-2020-6464, CVE-2022-3315, CVE-2023-6348). Across all these CVEs, developers tried to detect these vulnerabilities (*e.g.,* using `static_cast` or custom runtime type information) but ultimately failed due to mismatches between the expected static types in the source code and the dynamic type of the objects at runtime. Type confusions are reported under the Common Weakness Enumeration *CWE-704: Incorrect Type Conversion or Cast* and caused by *CWE-843: Access of Resource Using Incompatible Type*.

Considering the impact of type confusion attacks, researchers designed new languages that are type-safe by design, such as Java or Rust [3]. These languages prohibit erroneous typecasts, mitigating the attack surface. However, due to the compatibility between C++ and C, enforcing type safety and thereby mitigating type confusion is challenging. For interoperability, C++ provides an almost duality between classes and C-style structs. Once an object has been allocated, there is no inherent way of retrieving its type from the memory encoding. In C, casts merely determine the fields' offset. With polymorphic objects (those with virtual functions), C++ introduces vtable pointers (*vptr*), *i.e.,* runtime type information (RTTI) to uniquely identify the type of the object. Without vtable pointer, the type of a memory area remains opaque at runtime. C++ provides different cast operations offering varying guarantees. Only dynamic casts execute a runtime check, but can only be used for polymorphic objects since they rely on the RTTI of vtable pointers. Conversely, static casts leverage the declared types in the source code to statically check class relationships while C-style casts carry neither a dynamic nor static check. To enforce type safety, only `dynamic_cast` should be used [4], an option that core contributors of the standard envision to ease even at the cost of breaking the ABI [5].

[†]Work done while at EPFL.

Network and Distributed System Security (NDSS) Symposium 2025
24–28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.23053
www.ndss-symposium.org

LLVM-CFI [6] leverages RTTI to protect casts involving the fraction of polymorphic objects, *i.e.,* only around 3% of all the casts in our benchmarks. Alternatively, the academic community proposed several sanitizers [7], [8], [9], [10], [11], [12], [13] that track all objects involved in down-to casts and check their types against disjoint metadata. However, they incur high overhead and are imprecise (*e.g.,* if object copy is incorrectly tracked), resulting in both false positives and false negatives. Therefore, C++ remains prone to type confusion, a common attack vector used in exploits.

By introducing type++, a C++ dialect, we tackle type confusion vulnerabilities from a language perspective. By relinquishing some compatibility (with negligible impact in practice), type++ checks down-to casts at runtime, guaranteeing that objects are only used under the respective types specified at initialization. Our C++ dialect builds on the property that *each object is typable throughout its lifetime*. type++ associates a unique type (*i.e.,* a hidden field) to each object. This inline type field is key for fast and effective type checks for all casts, similar to other type-safe languages (*e.g.,* Java or Rust). At the same time, type++ is highly compatible with existing C++ programs. The few incompatible code patterns are easy to detect, and therefore amenable to straightforward patching. We analyze and classify the code adaptations that type++ requires. For each, we study solutions and measure the impact of such modifications on real cases. Where the code patterns are incompatible with type++, we devise compile-time warnings and errors that aid developers in porting code to the type++ dialect.

To demonstrate the efficacy of our dialect, we implement type++ by extending Clang and LLVM. Our compiler takes C++ programs as input, produces warnings in case of incompatibilities, and then emits executables protected against downcasting type confusion.[1] We deploy type++ across the two compiler benchmarks SPEC CPU2006 [14] and SPEC CPU2017 [15]. Additionally, we develop a benchmark to compare type++ to external metadata approaches. Moreover, we apply our type++ compiler to Chromium to show compatibility with, likely, today's most complex C++ code base. We choose the SPEC CPU benchmarks because their use cases range from simple examples (*e.g.,* NAMD ∼3 kLoC) to large, complex programs (*e.g.,* Blender ∼600 kLoC) and because it allows comparison with previous works. We successfully compile around 2,040 kLoC, encompassing both SPEC CPU2006 and CPU2017. Chromium is a massive, fast-moving project, and only setting up the environment to compile it with a different LLVM version may take months. We manage to protect 3,030 classes (out of 3,201 — 94.6%) in roughly twelve man months, while we deem the remaining classes easy to address given developers familiar with the codebase. For all our use cases, we measure the performance overhead, the number of lines of code modified during the porting, and compare it with existing runtime type confusion mitigations.

Our results show that type++ only incurs on average a 0.94% overhead when considering both SPEC CPU2006 and CPU2017 together, which is two orders of magnitude faster than HexType, a state-of-the-art type confusion sanitizer. Additionally, type++ validates 23 times more casts than LLVM-CFI. Compared to previous approaches, type++ does not suffer from false-positive type confusion detection since it is free from incorrect object lifetime tracking that plagued the previous type confusion detectors. Finally, type++ remains highly compatible with the current C++ ISO Standard. When looking at the SPEC CPU benchmark suite, inspecting the warnings resulted in modifying 125 LoC and 131 LoC for SPEC CPU2006 and CPU2017, respectively. For Chromium, our porting modifies only 229 LoC out of the 35 MLoCthe project contains.

To sum up, our contributions are:

- Specification of type++, a C++ dialect that enforces safe down-to cast by design through our baseline property that all objects remain typable throughout their lifetime.
- Study the adaptation required to port standard C++ programs over to type++ and propose an analysis/mechanism to help developers with this task.
- Evaluation of type++ against both SPEC CPU2006 and SPEC CPU2017, demonstrating its high compatibility and low performance overhead.
- Characterization of the porting for Chromium on type++ through a twelve months long study, along with a discussion of this effort.

The full source code, the documentation to replicate our experiments, and our technical reports are open-source: `https://github.com/HexHive/typepp/`.

II. Background

C++ is an object-oriented programming language with diverse features and constraints due to the tight relationship with the systems programming language C. Here, we detail the concept of class hierarchies (§II-A), C++ casting operations (§II-B), and the type confusion problem (§II-C).

A. Classes hierarchies and polymorphism

C++ relies on classes to implement polymorphism, which allows a developer to abstract generic concepts and reduce code duplication in a program. In the simplest schema, classes are organized hierarchically in parent-child relationships. This approach allows child classes to extend or override some functionality of the parent. In particular, C++ implements function override using a vtable, which is a table of function pointers to so-called virtual functions [16]. For instance, the classes `Greeter` and `Execute` in Listing 1 are both children of the class `Base`. Conversely, the class `UnrelatedGreeter` is disconnected from the inheritance tree. Defining a new class means introducing a new type in the program according to the language's type system. Therefore, an object of type `Greeter` cannot be used where an `Execute` or `UnrelatedGreeter` object is expected.

[1] Unless differently specified, we use the term type confusion to refer to downcasting type confusion throughout the paper.

B. Casting in C++

Casting allows the programmer to reinterpret the runtime type of an object. In addition to implicit conversion, C++ features five distinct casting operators with distinct rules and properties. The developer is responsible for choosing the appropriate casting operator.

1) `const_cast` strips the `const` and `volatile` property of an object. This operation has its caveats but is unrelated to type confusion. They are thus considered out of scope.

2) `dynamic_cast` changes the type of an object with runtime verification validating the compatibility of the object's type. The validation routine mandated by the language specification generally requires the presence of metadata in the form of runtime type information (RTTI). As RTTI is only available for polymorphic types, `dynamic_cast` cannot be used for non-polymorphic objects. The C++ Core Guidelines recommends `dynamic_cast` in almost all use cases as the performance impact is, despite rumors, negligible [4].

3) `static_cast` changes the type of an object similarly to `dynamic_cast`. However, no runtime check is performed nor is RTTI required. Limited casting validation is performed at compile time, namely the check searches for a path in the type hierarchy from the source to the target type. Lacking any runtime verification, wrong casts (*e.g.*, an object of type `Parent` could be cast into `Child`) lead to type confusion vulnerabilities. Casting a pointer from outside the class hierarchy (*e.g.*, `char*` or `void*`) into an object of a specific class is permissible as well. Incorrect usage of `static_cast` is the main culprit for type confusion in C++ applications.

4) The `reinterpret_cast` operator reinterprets the underlying memory area of the object as the target type. Verification happens neither at compile time nor at runtime, and the correctness is delegated to the programmer, thus raising similar security issues to `static_cast`.

5) In C++, C-style casts are automatically replaced at compilation time by following a priority schema: a `const_cast` is preferred over a `static_cast`, which in turn is favored over a `reinterpret_cast`. The first operator that satisfies compilation is selected. Due to their definition in terms of the other C++ casting operators, C-style casts inherit the underlying potential for type confusion vulnerabilities.

C. Type confusion

Type confusion vulnerabilities fall into two categories, namely *down-to-cast* and *unrelated type* (or `void*`) casting. *Down-to-cast* are portrayed in the functions `downToCast()` (line 28) while an *unrelated type* cast occurs in the `unrelatedType()` function (line 34) in Listing 1.

Down-to-cast vulnerabilities happen when a base class is cast into a subclass. In our example, the program treats an instance of a class `Execute` as if it is an instance of the `Base` class (line 42). Then, the base class is passed to the function `downToCast()` (line 43) which finally casts it into a `Greeter` object (line 29). At this point, the program erroneously considers the pointer `g` as a `Greeter`, however, the entry in the vtable of `g` points to the function `exec` that activates the payload (line 31) instead of the harmless `sayHi` function.

Listing 1 Example of a C++ class hierarchy. The functions `downToCast` and `unrelatedType` are examples of *down-to-cast* and *unrelated cast* type confusion vulnerabilities, respectively.

```
1  #include <iostream>
2
3  class Base { // Parent class
4    /* Other fields and functions */
5  };
6
7  class Execute : public Base { // Child of Base
8  public:
9    virtual void exec(const char *program) {
10     system(program);
11   }
12 };
13
14 class Greeter : public Base { // 2nd Child of Base
15 public:
16   virtual void sayHi(const char *str) {
17     std::cout << str << std::endl;
18   }
19 };
20
21 class UnrelatedGreeter { // Unrelated class
22 public:
23   virtual void sayHello(const char *msg) {
24     std::cout << "Hello: " << msg << std::endl;
25   }
26 };
27
28 void downToCast(Base* b, const char *msg) {
29   Greeter *g = static_cast<Greeter*>(b);
30   // exec() is invoked instead!
31   g->sayHi(msg);
32 };
33
34 void unrelatedType(void* p, const char *msg){
35   UnrelatedGreeter *g =
       ↪  reinterpret_cast<UnrelatedGreeter*>(p);
36   // exec() is invoked instead!
37   g->sayHello(msg);
38 };
39
40 int main(int argc, char *argv[]) {
41   const char *payload = "/bin/bash";
42   Base *b = new Execute();
43   downToCast(b, payload);
44   unrelatedType((void*)b, payload);
45   delete b;
46   return 0;
47 }
```

Unrelated cast vulnerabilities, instead, exploit backward compatibility for C programs that allows pointers to be considered as a generic type `void*`. In our example, the pointer `b` is an instance of class `Execute` (line 42). When `unrelatedType()` takes `b` as input (line 44), the compiler loses any type information of `b`. Therefore, the pointer can be cast as `UnrelatedGreeter` (line 35). The wrong cast allows an adversary to invoke `exec()` (line 37).

In the literature, there are two main approaches to mitigate these issues. First, approaches based on disjoint metadata structures trace and check the type of objects at runtime for down-to-cast [8], [9], [17], [10]. However, these approaches introduce considerable overhead and low precision. For instance, incorrect tracking of the object lifecycle (*e.g.,* due to not propagating metadata during object copies) causes stale metadata which might result in both false positives and negatives. False positives arise when the allocator reuses the space of a previously deallocated object without updating the metadata, thus leading to a misalignment between the actual object type and the stale metadata. False negatives, instead, appear when an object's metadata is unavailable and the system cannot decide whether a type violation occurred. Second, approaches may use existing (partial) type information to implement (partial) runtime type checks, *e.g.,* based on pre-existing RTTI metadata and `dynamic_cast` in C++. Unfortunately, C++ allocates RTTI metadata only for polymorphic objects, thus limiting the type check coverage. type++ overcomes the limitations of both approaches by defining a new C++ dialect that allocates inline metadata for all objects involved in downcasting, enforcing precise type checks, thus prohibiting type confusion by design.

III. Threat Model

As type++ can be deployed both as a sanitizer and as a mitigation, our threat model encompasses both scenarios.

We assume an adversary with knowledge of an illegal downcast, *i.e.,* the adversary knows where in the source code the downcast is and can construct an input to trigger a type confusion through this downcast. Our work prevents such type violations. More precisely, and in line with previous work [9], [8], arbitrary writes are out of scope for our mitigation. We assume that the attacker cannot modify the type information embedded in objects. This assumption is challenging to guarantee in practice but deploying mitigations like DEP, stack canaries, ASLR, CFI, and safe allocators [18], [19] approximates it. However, even with such mitigations, all out-of-bounds writes may not be prevented. For example, given a `strcpy` missing a length check and an array of objects whose layout ends with a `char*` object, an out-of-bound write may overwrite the next object vtable pointer.

Additionally, we assume a correct implementation of the type++ compiler and of the underlying operating system. Finally, as type++ does not rely on secrets, arbitrary reads do not generally compromise type++'s guarantees [19]. Leaking code pointers may allow an attacker to circumvent ASLR. While there are already many code pointers (*e.g.,* vtable pointers) available as target, type++ will increase the number of code pointers and, thereby, augment this attack surface.

IV. TYPE++ SPECIFICATION

C++ cannot enforce type safety due to the compatibility requirements of C++ objects with C structs. Also, historically, C++ compilers ran into performance and optimization constraints for type checks [16]. Yet, type violations are frequently used as the initial bug in exploit chains. Enforcing type checks as part of a safe C++ dialect mitigates these security risks. Essentially, C++ strictly associates a type only to polymorphic classes, all other structures and classes remain vulnerable to type confusion bugs. For simplicity's sake, we refer to classes and structures interchangeably. Our proposal, type++, is a novel C++ dialect that mitigates downcasting type confusion by design. In particular, type++ assigns a type to all objects of a program and enforces runtime checks for all down-to casts. We call this concept *Explicit Runtime Types*, formally described as:

Property 1 (Explicit Runtime Types.) *Given all classes CS of a program P, type++ associates a unique type T to each class $A \in CS$ at compile time. Embedding the type T into each object of class A enables explicit type checks before each downcast, ensuring their correctness.*

In practical terms, Property 1 ensures that each instance of a class A embeds a field that uniquely identifies its type. In classic C++, instances of polymorphic classes embed a vtable pointer for this purpose and a pointer in the RTTI section encodes the type information. Our property generalizes this requirement from polymorphic objects to all objects. We enforce this property at compile time. The introduction of this extra pointer allows a program to implement strict downcasting protections, thus mitigating completely these type confusion attacks. In our evaluation (§VI), we show that adopting type++ provides strong protection at low performance overhead. Our system can, therefore, be used both as an effective sanitizer (helping developers discover and fix bugs) and as a powerful mitigation (preventing exploitation of type confusion vulnerabilities through early program termination).

Enforcing Property 1 protects all objects, thus achieving strong type safety mitigation by design. In practice, however, type confusion attacks require the execution of a cast. Thus, adversaries target only classes involved in these casts. Therefore, without loss of generality or security, adding type information only to *classes involved in downcasting operations* achieves the same security guarantees as protecting all classes. Furthermore, blindly embedding type information into all classes might change assumptions in the code about a class, thus possibly reducing the compatibility between type++ and the C++ standard. In type++, we consider that a class is involved in downcasting operations if anywhere in the program an object of that class is either cast to a different type in the same hierarchy or if a cast returns an object of this type. Upon these considerations, we introduce a additional property, called *Explicit Runtime Types for Cast Objects*, that specializes Property 1 and assigns a runtime type only to those classes at risk of down-to type confusion attacks. We formalize this property as follows:

Property 2 (Explicit Runtime Types for Cast Objects.) *Given the classes CS of a program P, type++ associates a unique type T to each class $A \in CS$ if A is involved in a downcasting operation.*

To enforce Property 2, we infer, during a compile-time analysis phase, which classes are involved in downcasting. Then, we associate a unique type T only to those classes involved in casting operations. In §VI, we measure the impact of this optimization and show that enforcing Property 2 on SPEC CPU required us to change only 314 LoC out of 2,040 kLoC (around 0.04%).

To further improve the compatibility between standard C++ and type++, a developer can restrict Property 2 only to a subset of classes, manually annotated in the source code. This allows developers to apply type++ only to some program components, thus reducing the number of incompatibilities to address. This feature is implemented as an additional property called *Explicit Runtime Types for Annotated Objects*, formally described as:

Property 3 (Explicit Runtime Types for Ann. Objects.) *Given the classes CS of a program P, and a set of annotated classes $AC \subseteq CS$, type++ associates a unique type T to each class $A \in AC$.*

The enforcement of Property 3 enables us to deploy type++ on complex programs, such as Chromium, by using limited resources (i.e., graduate students unfamiliar with the Chromium code base) and protecting 94.6% of its classes. Further information about this case study is provided in §VI-D.

In the rest of this section, we discuss the differences between type++ and standard C++. Without loss of generality, we mainly consider Property 1 as it has stricter assumptions, while we switch to Property 2 or Property 3 only in clearly stated specific cases.

A. Affected programming patterns

type++ relies on Property 1 to ensure strong typing. However, this protection also introduces new assumptions not considered in the C++ standard and that lead to different programming patterns. Here we discuss the new patterns introduced by type++ and how to adapt legacy C++ code to our dialect. Since porting consumes precious developer time, we measure the impact of such efforts in §VI. Our evaluation shows that the trade-off between adopting type++ and the new security guarantees is acceptable.

Phantom casting. Listing 2 shows an example of phantom casting. A phantom class is a parent-child relationship where the child's data layout is equivalent to the parent's data layout [9] (lines 3 and 5). Even though this corresponds to an illegal downcast in principle, in practice, current C++ implementations ignore phantom casting to maintain interoperability between C and C++ code.

Solution and Impact: Even though this practice may be exploited for de-facto type confusion [20], type++ allows their usage for backward compatibility. However, since this may cause issues in later development, we implement a static analysis to find active cases. In our evaluation, we report no such type confusion when applying Property 2 (§VI-A).

Listing 2 Example of *phantom* casting.

```
1  #include <stdio.h>
2
3  class BaseType { /* other fields */ };
4
5  class PhantomType : BaseType {};
6
7  void checkCast() {
8    BaseType *ptr = new BaseType();
9    // The following cast results in Undefined
10   // Behavior in standard C++ but is commonly
11   // tolerated as both classes have the same layout.
12   if (dynamic_cast<PhantomType*>(ptr) == NULL)
13     printf("error!\n");
14 }
```

`sizeof()/offsetof()` usage. C++ offers the `sizeof()` and `offsetof()` operators. The former checks the size of classes or structures. The latter, instead, returns the offset of a field with respect to a class or structure. In Listing 3, we show an example as part of a `Substitution Failure Is Not An Error` (SFINAE) expression [21]. Adopting type++ introduces an extra RTTI pointer in the classes `yes` and `no` (lines 8 and 12), altering their expected size and the results of `val` (line 28).

Solution and Impact: `sizeof` and `offsetof` return the correct values when taking the extra type field into consideration, which is the expected behavior for most use cases (*e.g.,* `malloc(sizeof(T))`). However, issues may arise when comparing the results with a scalar or with another value returned by these operators. Comparison with a scalar (*e.g.,* `sizeof(no)` == 2) is already discouraged in standard C++ as the type size might be implementation-dependent. With type++'s extra type field, the expected scalar will differ. To mitigate the misuse of these features, we design two strategies. First, one can enforce Property 2 to reduce the number of instrumented classes, thus improving the cross-compatibility between standard C++ and type++. Second, we designed a static analysis to emit a warning whenever an instrumented class is used as a parameter of `sizeof()` or `offsetof()`. Our evaluation shows high usage of this pattern, specifically 957 occurrences for Property 1 and 129 occurrences for the relaxed Property 2 in SPEC CPU benchmarks. In practical terms, a single pattern (*e.g.,* a SFINAE template) required a trivial source code adaptation. This high compatibility is due to the small number of scenarios where `sizeof()` misbehave (*i.e.,* when comparing to a scalar or when padding is involved as in Listing 3). The code adaptations are also usually simple (*e.g.,* increasing the size of `no` to over 16 bytes, line 12). We discuss these cases in more detail in §VI-A.

Implicit `placement new`. Listing 4 shows an example of *implicit* `placement new`. The class Y is used to allocate memory for X in a similar fashion to the `placement new` C++ operator. Without type++, the objects instantiated by classes X and Y have the same size (line 11) and may be used interchangeably—as long as the developer accepts the underlying type confusion. However, in type++, due to the Property 1, class X and class Y contain an RTTI pointer which increases the class size. Therefore, the resulting size of class Y exceeds the one of class X as it includes space for two vtable pointers, its own mandated by RTTI and the one

Listing 3 Code example of `sizeof()` in SFINAE. The issue arises because `sizeof(yes)` and `sizeof(no)` now have the same size due to extra padding when using type++.

```
#include <iostream>

struct foo {
  typedef float X;
};

struct yes {  // type++: sizeof(yes) == 16
  char c[1]; // due to struct padding matching
};           // the alignment of the RTTI pointer.

struct no {  // type++: sizeof(no) == 16
  char c[2]; // due to struct padding matching
};           // the alignment of the RTTI pointer.

template <typename T>
struct has_typedef_X {

  template <typename C>
  static yes& test(typename C::X*);
  template <typename>
  static no& test(...);

  static const bool val =
  ↪    sizeof(test<T>(nullptr)) == sizeof(yes);
};

int main(int argc, char *argv[]) {
  std::cout << std::boolalpha;
  std::cout << has_typedef_X<int>::val << std::endl;
  ↪    // standard C++: false, type++: true
  std::cout << has_typedef_X<foo>::val << std::endl;
  ↪    // true for both standard C++ and type++
  return 0;
}
```

contained in `__blob_` (line 4) for the vtable pointer of X. type++ reports the type confusion between class X and class Y but might produce an error at runtime as the cast at line 9 leads to out-of-bound accesses.

Solution and Impact: As classes X and Y are not related, type++ alerts the developer of the type confusion. The valid fix would be to replace this pattern with the C++ `placement new` operator. In type++, this pattern cannot be handled automatically and requires a fix for valid execution. While current C++ compilers accept this pattern, it relies on Undefined Behavior as the translation between X and Y is not well-defined. In our evaluation, we found 131 (for Property 1) and 3 (for Property 2) instances of this pattern in the 2,040 kLoC of the SPEC CPU benchmarks. However, none of them resulted in a runtime error nor produced unexpected behaviors (*i.e.,* the benchmarks' output was correct). In addition to reporting the type confusion, type++'s analysis identifies this code pattern and warns the developer that a fix is necessary.

Listing 4 Code example of implicit `placement new`.

```
1  class X { /* other fields */ };
2
3  class Y {
4    char __blob_[sizeof(X)];
5  };
6
7  int main(int argc, char *argv[]) {
8    X* x;
9    Y* y = reinterpret_cast<Y*>(x);
10   X* z = reinterpret_cast<X*>(y);
11   static_assert(sizeof(X) == sizeof(Y), "error");
12   // The above assert is true in standard C++ but
13   // false in type++.
14   return 0;
15 }
```

V. TYPE++ TECHNICAL DETAILS

A. Default constructors

To enforce the above properties, type++ requires a default constructor for each instrumented class to set the vtable pointer of an object correctly. We use the default constructors for handling heap allocators (§V-C) and union (§V-D) initialization. However, naively synthesizing default constructors might break either the C++ semantic or the original program logic. For instance, a class might be purposely designed without a default constructor to be a POD (Plain Old Data) type or a non-clonable object (*i.e.,* classes without copy constructor [16]).

To automatically inject dummy default constructors without breaking either the C++ standard nor the developer intention, type++ uses a two-step compilation approach. The first step verifies that the program is C/C++ compliant without considering the type++ specifications, *i.e.,* ensuring that the program adheres to the C/C++ semantic and is free of compilation errors. In the second step, we override the C++ semantics and forcibly inject the default constructors. Note that, if the first step fails, the compilation does not proceed. This allows us to keep the original C/C++ language semantic along with helpful compiler warnings or error messages while synthesizing default constructors for the vtable pointer initialization.

B. Uninitialized variables

Uninitialized variables lead to undefined behavior [22]. An uninitialized object does not call a constructor and therefore does not initialize the vtable pointer. This might cause crashes during `dynamic_cast` as well as crashes during type++ checks. We detect the use of uninitialized variables by enabling the Clang flag `-Wuninitialized` during the initial compilation and report the warnings to the developer.

C. Allocation through C-style allocators

When an object is created using an allocator from the C-style `malloc` family (*e.g.,* `malloc`, `realloc`, or `calloc`), the system only reserves space for the object without initializing it, thus not setting the vtable pointer. To enforce Property 1, our static analysis automatically identifies the use of C-style allocators for instrumented classes. Then, it explicitly sets the vtable pointer right after the allocation using a default constructor. Spurious use of `calloc` instead of `malloc` may allocate more memory than required for a single object. We mitigate this behavior by retrieving the size of the block of memory effectively allocated via `malloc_usable_size()` and loop through the allocated memory, calling the constructor for as many objects that fit in the block. This transformation allows us to correctly handle vtable pointer initialization through `realloc` by only setting the vtable pointer on newly allocated memory. Indeed, calling the constructor on previously allocated memory could result in data being overwritten. Therefore, when faced with an allocator from the `realloc` family, type++ first retrieves the size of the previously allocated memory. This allows us to compute the range of newly allocated memory and only initialize the vtable pointer there. We process `std::allocator_traits` [16] likewise: For each class implementing the trait, we identify the memory allocation and inject the vtable pointer initialization, accordingly. While our prototype implementation relies on `libc` to retrieve the actual allocated size, extending support to other systems is straightforward (e.g., via `_msize` on Windows [23] or `malloc_size` on Mac OS X [24]).

Besides the standard system allocators, we encountered many custom allocators built on top of the `malloc` primitives. Many of them injected a custom header in front of the allocated object to store metadata or debug information. Having an extra header obscures the object's location, thus hindering the pointer arithmetic and causing unpredictable crashes. To automatically handle these cases, type++ contains an allow-list with the custom allocator functions and their respective header size. The compiler takes this number into account to locate the future objects in memory and then invokes the constructor accordingly. The allow-list is externally configurable. We use this technique to correctly handle POV-Ray (SPEC CPU2006 and CPU2017), Xalan-C++, and Blender, which otherwise would generate non-protected objects (*i.e.,* missing vtable pointer during type checks). While our technique handles the benchmarks correctly, we recommend a rewrite of these allocator patterns, following a modern programming style [25].

D. Polymorphic union members

In type++, a union, like any object, must have a type attached according to Property 1. By design, it is unknown which type a union will hold at compilation time. type++ cannot, therefore, know which constructor to call when the union is first declared. This issue also prevented C++, before the C++11 standard, from having polymorphic objects as union members. To highlight the issue, let us suppose we have a union with two polymorphic classes and an `int` variable. When we create an object, we cannot predict which member's constructor to call as we do not know which member of the union will be activated later. Explicit constructors (*e.g.,* `placement_new`) are generally needed when the type is changed through a union [26]. To automatically address this, our compiler inserts a constructor call every time a union switches from or to an instrumented type.

E. Initialization of `const` variable

`const` variables have to be completely initialized during their declaration [27]. In type++, the object initialization is done in two steps: first, we set the vtable pointer, and second, we invoke the actual object constructor. Therefore, naively applying this transformation would break the `const` constraint. In our compiler, we solve this case by relaxing the `const` property for classes instrumented and allowing exactly two initialization steps. First, forcibly inserting an additional constructor exclusively sets the vtable pointer. Then relying on the native constructor to instantiate the object. The language semantics are preserved as program correctness is verified in the first compilation step described in §V-A. As an alternative, we could predict the initialization of the `const` variable and initialize it in a single step, however, this requires a complex error-prone analysis.

F. Interaction with other C++ libraries

Programs sharing classes with external libraries must agree on a common per-object memory layout. For instance, a program P might use a class C to communicate with a library L. Compiling P for type++ adds an extra vtable pointer to C and modifies its memory layout (due to Property 1). Therefore, if L is not aware of the new layout of C, the program will execute incorrectly.

To solve this issue, we simultaneously build P and L with our type++ compiler and impose Property 1 to both. This approach ensures P and L follow the same data layout. For Property 2, we instrument the set of classes involved in casts in either P or L. However, this approach requires each library L to be specialized for each program P. In case L is closed-source, one can tune Property 3 and select only those classes that do not interact between P and L. This reduces the security guarantees as all the objects of the excluded classes will not be checked at runtime but improves the compatibility with closed-source libraries. When possible, we highly recommend specializing the library to avoid any risk of type confusion.

G. Interaction with the kernel and non-C++ code

Similar to uninstrumented C++ libraries, interacting with the operating system and other native code via modified objects may create compatibility issues. Considering the burden of rebuilding the kernel, we propose a wrapper function for system calls in `libc` to remove the vtable pointer before sending modified objects to the kernel. Once the system call returns, the wrapper then adds back the vtable pointer for further handling by the compiled program.

Due to Property 2, we did not encounter any such case in our evaluation. All type++ programs interacted with `libc` without requiring any wrapper. While this is not a guarantee, the large amount of benchmarks we executed indicates that this is not a problem in practice. Without Property 2, each C struct, including those passed to the kernel, would be instrumented, requiring rebuilding the kernel or inserting multiple wrappers to adjust the objects' layout and remove vtable pointers. This highlights the benefits of our optimization over the coarse-grained approach of Property 1.

In general, if such translated structs are passed to non-type++ code, a compiler pass can detect it and warn the programmer that a translation function may be required.

H. Prototype implementation

Our type++ compiler prototype is based on the LLVM infrastructure (version 13.0.0, the latest release at project instantiation). The implementation consists of 3.1 kLoC added to Clang and LLVM passes. In particular, we modify Clang/LLVM to (i) detect and warn in case of incompatible code patterns, and (ii) change the data layout in non-polymorphic objects. We implement typecasting verification by extending the security properties of LLVM-CFI [6]. Natively, LLVM-CFI performs type checks only over polymorphic objects, thus leaving a wide attack surface for all non-polymorphic object types. In type++, we augmented all classes with RTTI, thus extending the coverage of LLVM-CFI.

As part of the evaluation, we create a microbenchmark to compare the overhead of different typecasting verification approaches. As described in §VI-C, the workload tries to mimic the typecasting behavior of OMNeT++ which is part of the SPEC CPU2006 benchmark.

VI. Evaluation

Our evaluation of type++ explores four research questions:

(**RQ1**) What is the compatibility between C++ programs and the type++ dialect (§VI-A)?

(**RQ2**) What are the new type++ security guarantees (§VI-B)?

(**RQ3**) What is the overhead introduced by type++ (§VI-C)?

(**RQ4**) What is the porting effort of type++ for a major C++ project (§VI-D)?

Experimental setup. Our experiments are performed in a Docker container based on Ubuntu 20.04 running on a server with two Intel Xeon E5-2680 v4 @ 2.4GHz CPUs with 256GB of RAM in total.

Target programs. We evaluate type++ on two popular benchmarks: SPEC CPU2006 [14] and CPU2017 [15]. For each benchmark, we select all included C++ programs, resulting in seven targets from SPEC CPU2006 and nine from SPEC CPU2017. We additionally target Chromium 90, the most recent version compatible with LLVM 13 that is available on Debian. We measure Chromium runtime performance through JetStream2 [28], an aggregation of JavaScript and WebAssembly benchmark.

State-of-the-art. We compare type++ against four recent state-of-the-art projects: TypeSan [8], HexType [9], EffectiveSan [13], and LLVM-CFI [6]. Note that LLVM-CFI provides typecasting verification but limits it to polymorphic objects only. Since TypeSan, HexType, and EffectiveSan are seven and six years old, respectively, we managed to compile them only against the seven targets from SPEC CPU2006, while we built all the 16 targets with LLVM-CFI. We run all the related work experiments with the experimental setting of type++. As a common baseline, we use LLVM 13 and optimization level `O2` with Link Time Optimization enabled as it is required by LLVM-CFI. LLVM-CFI and type++ are both built on top of LLVM 13. The relative numbers of both of them refer to native execution on LLVM 13. Likewise, we used LLVM 3.9 for TypeSan and HexType and LLVM 4.0 for EffectiveSan since they were developed for these platforms, respectively.

A. Compatibility analysis

We perform a compatibility analysis and count lines of code (LoC) that match the programming patterns described in §IV-A. For detecting *implicit* `placement new` and `sizeof()/offsetof()`, we employ a conservative static analysis over the AST to avoid true negatives, as done in previous works [29]. For phantom casts, we extract the class hierarchy and infer the classes' data layout through LLVM. Since we rely on static analysis, the final numbers are a generous over-approximation. We implement these analyses as a plugin for Clang. Additionally, we count the number of uninitialized objects that are undefined behavior in standard C++. These issues are orthogonal to the type++ porting efforts and may result in miscompilation even in standard C++. To identify uninitialized objects, we enable the flag `-Wuninitialized` during the vanilla compilation [30]. We measure the number of affected code patterns for Property 1 and Property 2, respectively.

Table I shows the result of our analysis. Overall, compiling the SPEC benchmarks with Property 2 produces 179 warnings compared to 1480 warnings for Property 1, *i.e.,* the optimization of only instrumenting cast-related classes reduce the number of warnings by almost 90%. When applying Property 1, we encounter a few incompatibilities with the `sizeof()` operator as part of SFINAE expressions in the `Boost` library (similar to the example in Listing 3). As the classes involved are never cast, the issues disappear when using Property 2. Generally, most of the programs work without any modifications and only a fraction of them require manual source code adaptations. Well-behaved programs written in modern C++ generally do not require any modifications. More specifically, we modify deal.II, Blender, and POV-Ray (in both SPEC CPU2006 and CPU2017). They require changing 314 LoC in total (around 0.04%). Besides deal.II, whose modification was trivial, we discuss Blender porting efforts below and the ones of POV-Ray and Xalan-C++ in Appendices §A-B and §A-C.

Focusing on the detected warnings, *implicit* `placement new` warnings require to inspect at maximum 131 LoC when considering Property 1 and only 3 LoC in the case of Property 2. For *phantom casting*, our analysis highlights 129 LoC for Property 1 but none when Property 2 is used. `sizeof()/offsetof()` warnings concern at most 957 LoC with Property 1 and only 129 LoC otherwise. In this case, the only real incompatibility observed with `sizeof()` resides in SFINAE expressions in the `Boost` library. The other warnings for `sizeof()` are tied to memory allocation routines, thus not causing any trouble as the size is correctly adjusted during compilation. We do not observe any runtime errors/misbehavior related to warnings from `offsetof()`. We also encounter a very limited number of uninitialized objects, 21 for Property 1 and 6 for Property 2, respectively. Upon further investigation, we conclude that the warnings were false positives due to the over-approximation of Clang's default analysis and do not introduce runtime issues, such as object initialization behind opaque conditions.

Table I: Number of warnings in SPEC CPU2006 and CPU2017. For each cell, we indicate the number of possibly incompatible LoC in the format (# LoC w/ Property 1) / (# LoC w/ Property 2). The numbers show that most of the programs do not require modification, while the smallest patch modifies only 2 LoC (deal.II) and the biggest only 131 LoC (POV-Ray). This evaluation confirms the low impact of porting C++ projects to type++ dialect. The last three columns indicate if the program uses a custom allocator (*i.e.,* C), the number of unique program locations that introduce type confusions, and the compilation duration overhead, respectively.

	Use Case	Total LoC	Implicit `plac. new`	Phantom Casting	`sizeof()` `offsetof()`	Uninit. Objects	LoC Changed add (+)	del (-)	Custom Allocator	T. C. Errors	Compilation Overhead
SPEC CPU2006	NAMD	3,887	0 / 0	0 / 0	1 / 0	0 / 0	-	-	-	-	106%
	deal.II	94,832	1 / 0	3 / 0	51 / 0	0 / 0	2	-	C	-	94%
	SoPlex	28,277	15 / 12	0 / 0	3 / 0	0 / 0	-	-	-	-	103%
	POV-Ray	78,679	0 / 0	0 / 0	223 / 61	9 / 3	79	44	C	56	106%
	OMNeT++	26,647	0 / 0	14 / 1	7 / 1	0 / 0	-	-	C	-	93%
	Astar	4,280	0 / 0	0 / 0	9 / 0	0 / 0	-	-	-	-	112%
	Xalan-C++	264,389	5 / 0	129 / 0	13 / 0	1 / 0	-	-	C	4	107%
SPEC CPU2017	CactuBSSN	63,307	0 / 0	0 / 0	1 / 0	0 / 0	-	-	-	-	108%
	NAMD	6,396	0 / 0	0 / 0	6 / 0	0 / 0	-	-	-	-	110%
	Parest	359,012	72 / 20	7 / 0	101 / 3	0 / 0	-	-	-	1	108%
	POV-Ray	80,079	0 / 0	0 / 0	223 / 61	9 / 3	87	44	C	53	109%
	Blender	615,895	14 / 0	0 / 0	15 / 9	0 / 0	47	11	C	1	103%
	OMNeT++	85,732	0 / 0	102 / 0	35 / 1	2 / 0	-	-	C	2	111%
	Xalan-C++	291,160	24 / 3	125 / 0	259 / 1	0 / 0	-	-	C	5	109%
	Deep Sjeng	7,284	0 / 0	0 / 0	7 / 0	0 / 0	-	-	-	-	117%
	Leela	30,524	0 / 0	0 / 0	3 / 0	0 / 0	-	-	-	-	84%
	Total	2,040,380	131 / 35	380 / 1	957 / 137	21 / 6	215	99	-	122	106%

Finally, regarding custom allocators, we address them by using an allow-list of functions. During our evaluation, we find 16 custom allocators across 6 programs (deal.II, Blender, SoPlex, Xalan-C++, and both versions of POV-Ray). type++ gracefully handles allocators with and without custom headers within the same program, for example, `MEM_mallocN` and `BLI_memarena_alloc` in Blender.

Use case: Blender. We encounter three different sources of incompatibility when porting Blender. The first problem involves shared structures (defined in `.h` files) between C and C++ code (§V-G). Due to Property 1, the `structs` in C++ now contain an additional `vptr` field, that remains absent in the C code. This results in having two different memory layouts for the `structs` in C and type++ code, leading to unpredictable crashes. We fix this issue by adding a field of the same size as the `vptr` pointer at the beginning of the structure in the C code, the field is activated only when compiling as C source code, thus adjusting the memory layout between the two languages. Our second issue is that Blender relies on fat pointers to store metadata information (*e.g.,* using the LSB as object type reference). The Blender developers employ bit-mask operations before pointer dereferencing. type++ flags these locations as type confusion as it cannot locate the vtable pointer due to the address offset. Upon closer inspection, we observe the vtable pointer is correctly set but the memory address is not pointing to the beginning of the object. We manually patch the code by adding a bit-mask before cast operations to adjust the pointer operations. If fat pointer casting is used at multiple locations, applying the unmasking operation automatically at cast verification time could be implemented as an optional feature. As overloading pointers is not common, we argue in favor of manually fixing a few cases. Blender casts fat pointers only at two locations, we address this issue with 20 LoC modified. Finally, Blender uses custom heap allocators that wrap the standard `malloc` family functions. In this case, we add the `malloc`-like functions in type++'s custom allocator list without source code modification (§V-C). Overall, we modify only 58 LoC out of more than 600K (less than 0.1%) while the program logic stays untouched.

Takeaway. This evaluation shows the high compatibility between C++ and type++. In particular, applying Property 2 requires limited effort to port C++ projects, *i.e.,* we modified 123 and 131 LoC for both POV-Ray (0.16%) and 58 LoC for Blender (< 0.1%) while it protects every cast operation. Refer to §VII for specifics that could limit type++ adoption.

B. *Security evaluation*

To evaluate the security guarantees of type++, we measure the number of downcasts checked at runtime. Specifically, we run the 16 use cases of the SPEC benchmarks against type++, TypeSan, HexType, and LLVM-CFI whose results are in Table II (LLVM-CFI) and Table III (TypeSan and HexType).

Table II shows the result of our experiments in regards to coverage of type casts with type++ compared to LLVM-CFI. Due to the introduction of Property 1, type++ allows all previously undetected objects to emerge and be properly verified. This is particularly evident for SoPlex, POV-Ray, and Leela, in which type++ alters normal classes into polymorphic ones. type++ can check the integrity of all downcasts, resulting in an additional 13B and 51B runtime cast for POV-Ray 2006 and 2017, respectively. Similarly for Leela, where type++ can now monitor the integrity of 21M down-to-casts that otherwise would remain unchecked.

Table II: Overhead and coverage evaluation of type++ and LLVM-CFI deployed over SPEC CPU2006 and CPU2017. For each program, we indicate the average overhead measured (*i.e.,* %), the number of down-to casts, and unrelated casts observed in our experiments. In the right-most part of the table, we compare LLVM-CFI and type++: the delta columns (Δ) show the difference in terms of casts protected, while the last two columns indicate type++ memory overhead.

		LLVM-CFI			type++			Δ		memory (%)	
	Use Case	(%)	down-to	unrelated	(%)	down-to	unrelated	down-to	unrelated	avg.	max.
SPEC CPU2006	NAMD	-0.52	0	0	-0.82	0	0	0	0	0.58	0.59
	deal.II	1.50	0	0	2.02	17,462M	122M	17,462M	122M	-0.18	0.20
	SoPlex	-0.22	0	0	1.15	209K	28M	209K	28M	1.94	2.98
	POV-Ray	0.55	0	1K	4.11	11,477M	1,342M	11,477M	1,342M	0.44	0.43
	OMNeT++	3.43	1,897M	3	4.07	2,521M	270K	624M	270K	0.35	0.16
	Astar	-1.16	0	0	-0.15	0	0	0	0	0.27	-0.09
	Xalan-C++	-0.12	282M	4K	0.09	284M	5K	2M	612	0.32	-0.01
SPEC CPU2017	CactuBSSN	-2.54	30	0	-0.87	80K	100	80K	100	0.12	0.11
	NAMD	-0.47	0	0	-0.42	0	0	0	0	0.17	0.17
	Parest	-0.26	11M	18K	-0.11	2,462M	85M	2,451M	85M	-0.83	-0.58
	POV-Ray	0.66	0	573	3.04	45,861M	5,370M	45,861M	5,370M	3.82	3.82
	Blender	0.25	0	0	4.58	0	7M	0	7M	1.39	1.37
	OMNeT++	3.22	1,428M	6M	2.59	2,786M	6M	1,358M	829K	1.21	1.10
	Xalan-C++	-0.28	227M	4K	-0.81	227M	198M	0	198M	1.52	1.20
	Deep Sjeng	0.12	0	0	-0.41	0	0	0	0	0.00	0.00
	Leela	0.57	0	0	0.43	21M	1K	21M	1K	0.12	0.16

Table III: Performance comparison of type++ against TypeSan, HexType, and EffectiveSan. Since TypeSan and HexType only mitigate down-to-casts, we do not report unrelated casts for type++. EffectiveSan has a different definition of cast checking which does not allow a direct comparison. We, therefore, omit the numbers and include a discussion in §VI-C.

	Use Case	TypeSan		HexType		Eff.San	type++	
		(%)	# cast	(%)	# cast	(%)	(%)	# cast
SPEC CPU2006	NAMD	0.00	0	0.17	0	588	-0.82	0
	deal.II	74.25	3,379M	6.13	3,380M	1,212	2.02	17,462M
	SoPlex	0.00	0	0.00	209K	497	1.15	209K
	POV-Ray	23.52	0	-2.39	0	667	4.11	11,477M
	OMNeT++	44.32	2,014M	29.21	2,014M	229	4.07	2,521M
	Astar	1.70	0	1.05	0	310	-0.15	0
	Xalan-C++	35.46	283M	17.96	283M	1,593	0.09	284M

Table IV: Type confusions found by HexType, EffectiveSan, and type++ in SPEC CPU. In SPEC CPU2006, type++ finds a superset of the errors found by previous works while incurring lower overhead.

	Use Case	HexType	Eff.San	type++
2006	POV-Ray	0	56^*	56
	Xalan-C++	2	2^*	4
2017	Parest	-	-	1
	POV-Ray	-	-	53
	Blender	-	-	1
	OMNeT++	-	-	2
	Xalan-C++	-	-	5

* Numbers from the paper that we could not reproduce.

Using inline cast information, combined with a complete list of custom allocators (§V-C), allows type++ to overcome the coverage issues affecting disjoint metadata approaches [9]. As shown in Table III, type++ does not miss any cast and indeed protects *every* object passing through a downcast. Similarly to LLVM-CFI, type++ has an option to additionally protect unrelated casts (*i.e.,* cast over `void*`) whose classes are polymorphic or instrumented. This feature allows type++ to stretch its protection beyond the disjointed approaches without any further porting cost. As a consequence, type++ protects a vast number of runtime casts that, so far, were unprotected, *e.g.,* for SoPlex and POV-Ray 2017 (28M and 5B casts respectively).

There are limited actions to bypass type++, *e.g.,* a possible issue could emerge if custom allocators are not properly allow-listed by the developer. Another possibility is to use a flawed type check logic. We assume the logic is sound (§III), moreover, our prototype relies on the standard type checks in LLVM, which has been widely tested and optimized by the community. Finally, another cause of type confusion could come from undefined behavior or other memory safety violations, which we considered out of scope (§III).

From our evaluation, we observe a total of 122 type confusions in SPEC CPU2006 and CPU2017, among them, 14 bugs are newly discovered by type++. Table IV compares the type confusions found by type++, HexType, and EffectiveSan. type++ can mitigate all the bugs discovered by either EffectiveSan or HexHype, demonstrating that our dialect protects an attack surface covering both state-of-the-art tools. We discuss how we tackle type confusions in POV-Ray (Appendix §A-B) and Xalan-C++ (Appendix §A-C). We attach detailed technical reports describing the type confusions in our open-source documentation. These type confusions have been fixed in more recent versions of the benchmark code. The patches are in line with type++'s properties (*e.g.,* using a proper class hierarchy and dynamic cast).

Takeaway. Our evaluation shows that type++ validates *every* downcast: it covers up to 10^9 more casts compared to the previous state-of-the-art, *e.g.,* deal.II and POV-Ray have 17B and 45B more runtime checks than with LLVM-CFI.

C. Performance overhead

We assess the performance overhead of type++ over the two benchmark sets previously introduced, SPEC CPU2006 and CPU2017. For the two SPEC benchmarks, we recompiled each use case in vanilla (*i.e.,* using the unmodified Clang as the compiler) and with alternative protection/mitigation techniques, specifically, TypeSan, HexType, EffectiveSan, and LLVM-CFI. We repeated each run five times and considered the average execution time (we observed a negligible standard deviation). For type++, we rely on Property 2 as it offers the same security guarantee as Property 1 while instrumenting fewer objects, reducing performance overhead. We do not evaluate Property 1 overhead as it would require analyzing a vastly superior number of warnings as highlighted in §VI-A to obtain a worse runtime performance and no additional security guarantees. We summarize the evaluation of type++ against LLVM-CFI in Table II, while the results against TypeSan and HexType are in Table III. In both tables, we show the runtime overhead against their baseline (*i.e., %*) and count the number of casts protected at runtime. The latter is further split between the down-to cast in the scope of type++ and the unrelated cast that LLVM-CFI and type++ additionally support. We consider only down-to-cast for TypeSan and HexType since they do not cover unrelated casts. For EffectiveSan, we omit the number of casts since their definition is incompatible with ours. Additionally, we compare the memory overhead of type++ against LLVM-CFI in the last two columns of Table II. We also investigated the impact of patches on the program's performance. For this, we compile both original and patched programs against vanilla Clang and measure the overhead. Finally, we break down the cost of each operation of the different type-checking approaches. This shows that disjoint metadata approaches suffer from heavy lookup costs and cannot achieve performance on par with inlined approaches.

The overhead introduced by type++ (Table II) ranges from -0.87% (CactuBSSN) to 4.58% (Blender), while the LLVM-CFI overhead stays between -1.16% (Astar) and 3.22% (OMNeT++ 2017). The type++ overhead loosely correlates with the number of additional casts protected. For instance, OMNeT++ 2006 shows a performance overhead of 4.07% while protecting around 800M more casts than LLVM-CFI when considering *down-to-cast* and *unrelated casts* together (around 2.5B casts in total for type++ against 1.9B casts for LLVM-CFI, *i.e.,* 30% more). Similarly, for POV-Ray 2006, LLVM-CFI introduces an overhead of 0.55% for protecting only 1K casts, while type++ introduces a 4.11% overhead by covering more than 12B casts in total (down-to and unrelated casts together). This is due to our core property that ensures that every object possesses type information queryable at runtime, allowing type++ to stretch inline protections already tested and optimized to every cast, including many that remained unchecked by previous works.

In comparison with disjoint metadata structure approaches, such as HexType and TypeSan, the benefits of type++ are even more noticeable (Table III). While HexType exhibits a maximum overhead ranging up to 29.21% (OMNeT++) and TypeSan 74% (deal.II), type++'s overhead is limited to 4.58% and 4.11% for Blender and POV-Ray 2006, respectively. Nonetheless, type++ is capable of protecting more casts than previous approaches, *i.e.,* type++ covers 25% more casts than HexType for OMNeT++. Furthermore, Property 1 allows us to protect every cast, resulting in 14B casts from deal.II 2006 protected, that are covered by neither HexType nor TypeSan. The performance improvement is mainly driven by type++ not requiring heavy cast tracking nor disjoint metadata structures operations that introduce a notable overhead.

EffectiveSan uses Low-fat pointers [31] to store type information. This work shares a different threat model compared to type++ since they trace any type of cast regardless of security implications. The result is an impactful overhead that ranges from $\sim$230% to $\sim$1600%. This exemplifies how Property 2 limits the overhead by focusing on real cast operations. Additionally, we observe a few false positives when deploying EffectiveSan over deal.II. Specifically, EffectiveSan wrongly reports as type confusion some template variables that it infers belong to different types. Conversely, type++ did not show any false positives in our evaluation.

In terms of memory, type++ introduces a negligible overhead compared to LLVM-CFI that stays below 1.20% on average and 1.52% at maximum. The only exception is POV-Ray 2017 with an overhead of 3.82% (average and maximum). We deem this (limited) discrepancy to be caused by the additional casts protected compared to LLVM-CFI. Since we protect more objects, we introduce more RTTI in memory. However, we consider the observed overhead acceptable in practice.

The impact introduced by our patches is less than 3% in the worst case—1.46% (deal.II), 1.43% (POV-Ray 2006), 2.83% (POV-Ray 2017), 2.11% (Blender). Therefore, we argue that the patches, while modernizing the program code, do not harm performance nor reduce type++ overhead.

Finally, Astar, Deep Sjeng, and NAMD are examples of programs without any casting operations. In these cases, type++ did not meaningfully affect their performance since we do not introduce any cast-checking nor modify any object.

Performance on a microbenchmark. To more precisely understand the causes of the performance overhead for each type confusion protection, we design a microbenchmark that separates the cost of each operation. In particular, we analyze HexType as an example of the disjoint metadata approach. HexType can be decomposed into four major operations: metadata insertion, metadata deletion, metadata lookup, and type checking. We compare these operations with the ones of both LLVM-CFI and type++ as they are identical. As the type information is directly stored in the object the lookup cost is minimal, we, therefore, concentrate our effort on evaluating the cost of the LLVM-CFI type check. Despite investing large efforts, we were unable to isolate the different costs of the EffectiveSan prototype implementation. The prototype injects extra optimization flags into the compilation pipeline which activates vector optimizations as part of its LLVM integration. These extra optimizations disturb the results compared to the baseline. We were unable to disable these extra optimizations.

Table V: Breakdown of the cost [ns] of each operation for disjoint and inlined metadata type checking. For HexType, verification (Verif.) is the sum of a Type check and a Lookup operation. A cast verification is 7.7× faster in type++.

HexType					LLVM-CFI/type++
Insert	Delete	Lookup	Type check	Verif.	Verification
19.17	3.21	2.25	2.88	5.13	0.66

Our microbenchmark mimics the OMNeT++ workload from SPEC CPU2006: it creates 480M objects, 45% of which are involved in cast operations, and executes 2.5B cast operations in total. Per our analysis, 99% of the casts in OMNeT++ are caused by five unique code locations that we replicate in our microbenchmark. We also approximate the cast distribution and the class hierarchies. The most complex cast has a five-level hierarchy between the instantiated object and its base class. We measure the execution order of the 900M cast operations by looping through the five cast operations. The code is written to minimize caching effects and assess the actual cost of each operation. The benchmark is compiled with `O2` as optimization level and evaluated on the same machine as the rest of type++ evaluation.

For LLVM-CFI, we measure the duration of the validation of a static cast that type++/LLVM-CFI instrument. For HexType, instead, we isolate and measure the four main operations of disjointed metadata approaches: insert, lookup, delete, and type check. Table V summarizes the results. HexType has additional costs to handle the metadata lifetime (Insert and Delete), which is a single write for type++. Cast verification is a lookup and a type check for disjoint metadata approaches, while it is only a RTTI verification for type++. The figures highlight the heavy cost of the lookup operation of disjoint metadata approaches, which is more than three times slower than the type check itself of inlined metadata. This result shows that the key limitation of disjoint metadata approaches cannot be resolved by faster type checks alone. In disjointed metadata approaches, the bottleneck is caused by the query time to retrieve the data structures containing the type information necessary for the check. It has already been optimized across the different previous works. Currently, type checks are seven times slower in HexType than in type++. In addition, current disjoint data structure implementations are not thread-safe, enforcing thread-safety would likely introduce higher overhead. This leads us to conclude that type++ is the only reasonable approach to mitigate type confusion bugs while maintaining reasonable performances, as also shown by our experiments in Chromium (§VI-D). Further improvements on the performance of `dynamic_cast` are possible as shown in [32], [33], increasing, even more, the performance advantage of type++ over disjoint metadata approaches.

Takeaway. type++ largely outperforms state-of-the-art approaches in terms of overhead, while it extends inline-optimized protections to cast locations that would not be covered otherwise.

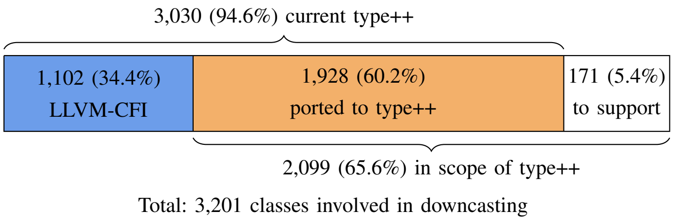

Figure 1: Chromium contains 3,201 classes involved in downcasting, among which 2,099 are in necessary for type++ to achieve Property 2. LLVM-CFI already protects 1,102 classes (34.4%). type++ currently supports 1,928 additional classes in Property 3 (60.2%, 91.8% in respect to “in scope for type++”). 171 classes (5.4%) are yet to be ported. Figure not to scale.

D. Use case: Chromium

In this section, we showcase the porting of Chromium to type++. We choose this project as an example of an established legacy codebase, analyze the challenges, and compare our findings with state-of-the-art solutions. Chromium is the open-source project underlying Google Chrome, the most popular browser. Chromium frequently faces type confusion vulnerabilities (*e.g.,* CVE-2019-5757, CVE-2020-6464, CVE-2022-3315). With over 35 MLoC written in C++, it is one of the biggest active open-source C++ projects and, therefore, an ideal target for type++. We choose Chromium version 90, which is distributed with Debian 10. For performance reasons, Chromium developers never use `dynamic_cast`, resorting to the unsafe `static_cast` in release builds. However, this performance trade-off comes at a security cost. Applying type++ to Chromium improves the browser security and serves as a benchmark for type++'s real-world applicability.

In the rest of this section, we detail the result of our compatibility analysis and discuss our deployment strategy. Then, we describe the patches applied, measure the overhead, and compare our approach to other mitigations.

Compatibility analysis. To assess the compatibility of Chromium with type++, we execute our analysis with Property 2, which reports 3,339 warnings (§VI-A). 54.1% of these warnings were linked to the implicit `placement new` issue, but none required a code change. The warnings, however, correctly point to code segments where improvements were sensible. While our analysis did not report any phantom class, it highlights 1,530 locations where `sizeof` was used.

Another source of potential incompatibilities comes from Protobuf [34], Google's data format for serialized structured data. `protoc`, the Protobuf compiler, generates C++ code from specifications. type++ analysis reported code patterns in generated code which required a minor adaptation to `protoc` to ensure that no infringing code is generated.

type++ deployment strategy. Figure 1 illustrates the Chromium classes involved in downcasting operations and which fractions type++ and LLVM-CFI each protect. Enforcing Property 2 over Chromium requires instrumenting 2,099 classes and manually addressing 3,339 warnings generated by our compatibility analysis (§VI-A). 1,102 classes already have RTTI and do not need modification since already protected by LLVM-CFI. Coping with all the issues at once is impracticable. Therefore, we adopt Property 3 to incrementally include compatible classes. For example, the team in charge of V8 [35] could apply type++ to Chromium instrumenting classes used in V8 only. This leaves the remaining classes unmodified by type++ and allows for a gradual deployment of type++ while reducing the risk of type confusion.

We develop a semi-automatic "delta-debugging" system that iteratively includes classes to Property 3, compiles, and validates Chromium. We additionally include a caching system to speed up Chromium compilation from 18 down to 6 hours on average, which remains the main bottleneck. Despite a lack of familiarity with the codebase, in roughly eight months, we manage to compile and run 1,928 of the 2,099 classes (91.8%) required by Property 2. These classes are linked to 2,428 warnings, leaving 911 warnings of Property 2 (27.3%) to further manual analysis. By considering all the classes already equipped with RTTI (34.4%) and those included with type++ (60.2%), we observe that type++ protects 1,928 classes (94.6%) involved in downcast operations in Chromium.

Chromium patching. From the compatibility analysis, we only modify 229 LoC out of 35 MLoC of Chromium C++ code base to support the above-mentioned 1,928 classes. Most of the changes involved assertions violated at compile time, *e.g.,* strict `sizeof` comparison with a scalar. Our patches disable these assertions. We plan to extend type++ with a special built-in macro which would make these assertions correct for standard C++ and type++ concomitantly. The modifications regarding `protoc` were succinct and involved only three files for 28 insertions and 19 deletions in total.

Chromium performance evaluation. Our evaluation uses the average of 10 runs of JetStream2 [28], testing three Chromium configurations: baseline, LLVM-CFI, and type++ with support of 1,928 classes (94.6% of the total). Table VI shows the final results. JetStream2 reported a score of 96.14 for baseline Chromium while LLVM-CFI suffers a 0.43% score reduction (95.73). type++ shows a slightly worse score of 94.80, indicating a 1.42% reduction of performance compared to the baseline. These results suggest that no major degradation of performance is caused by type++. These results are in line with the ones reported by the Chromium developers who observed that LLVM-CFI incurs less than 1% overhead when compared to the baseline [36]. The measured overhead does not affect the final user experience. type++ performance is on par with LLVM-CFI, and the additional 0.98% score reduction is caused by type++ protecting more than four times as many cast operations compared to LLVM-CFI (Table VI). Considering the severity of past type confusion vulnerabilities in Chromium, we argue that this extra overhead is reasonable. We are also confident that instrumenting the rest of the classes will not result in a vastly different performance since type++ already pays the cost by performing a failing type check for non-instrumented classes.

Comparison with other mitigations. The deployment of different type confusion protections has variable impacts in terms of compatibility and security guarantees. Here, we study the differences when applying type++, HexType, and LLVM-CFI—the current state-of-the-art. For HexType, we study its deployment over Firefox, which approximates the complexity of Chromium, thus comparing its reported results. LLVM-CFI is deployable on Chromium as part of the build system.

After porting 1,928 classes, type++ handles 741M down-to casts (out of 830M–89.7%). From code inspection, we conclude that the high coverage is caused by a narrow set of classes responsible for the majority of casts. HexType on Firefox protects a maximum of only 60% of the down-to casts. Similar to what was observed with SPEC CPU benchmarks, by protecting downcasting operations, we also stretch the protection of type++ over 69M unrelated casts, a form of cast that HexType cannot support. While the two browsers are not directly comparable, HexType in general protects fewer down-to casts and misses all unrelated casts compared to type++. The HexType authors mention that increasing their ratio of protected casts is impracticable as it would require complex code modification to Firefox or non-trivial adaptations of allocator tracking in HexType ([9]–Sec 5.1). In comparison, type++ already handles similar pool allocators natively (*e.g.,* Blender in SPEC CPU2017). On Chromium, type++ only misses casts of some classes due to the required engineering efforts to handle a project of this size. We further observe that HexType's implementation suffers from false positives, *i.e.,* type confusion that do not link to real bugs [37]. We reached out to the authors, who confirmed our observation. In certain cases, HexType suffers from stale metadata when the heap object lifecycle is not correctly handled (see §II-C for details), thus leading to false positive type confusions. The authors confirm that they do not delete old metadata to reduce runtime overhead. Conversely, for type++, every object embeds RTTI inline, thus carrying correct metadata throughout the object's lifecycle, and eradicating false positives by design. When evaluating LLVM-CFI on Chromium, we observe it detects only 347M (41.8% of the total) down-to casts. These numbers are in line with HexType when deployed on Firefox, and show the large attack surface left by previous works.

Attack surface. Figure 1 describes the attack surface when considering baseline, LLVM-CFI, and type++. From our analysis, there are 3,201 classes involved in down-to casts, 1,102 of them are already polymorphic, and 2,099 are non-polymorphic (POD). In the baseline, no classes are protected. Using LLVM-CFI cast protection, all the polymorphic classes become protected, *i.e.,* 34.4% of the total. The introduction of type++ pushes the tally to 94.6% of the total, almost three times more than LLVM-CFI.

Takeaway. Compared to LLVM-CFI, type++ manages to protect almost twice the amount of casts. Moreover, type++ reports no false positives and covers 69M unrelated casts, which would not be verified by HexType. After porting 1,928 classes, type++ already provides higher security guarantees than LLVM-CFI and HexType with minimal overhead.

Table VI: We compare Chromium baseline against LLVM-CFI and type++ with support of 94.6% of the necessary classes for Property 2, using the JetStream2 benchmark. A higher score is better. The third column is the number of casts protected. The last column is the percentage of all classes with RTTI, *i.e.,* including the polymorphic classes from the baseline.

Chromium	Perf. Score	(%)	Cast	Classes
baseline	96.14	-	-	34.4%
LLVM-CFI	95.73	0.43%	427M	34.4%
type++	94.80	1.42%	811M	94.6%

VII. Discussion

We discuss some potential challenges when porting source code to the type++ dialect.

Threats to type++ adoption. Previous dialects like CCured [38] and Ironclad [29], while pushing C and C++ in the right direction, failed to gain traction as a drop-in replacement. type++ proposes less drastic changes that are easier to adopt. Nonetheless, changing the language/dialect of a project may encounter resistance. We hope and will push for the inclusion of type++ into the main C++ compilers, initially as an alternative dialect, to ease accessibility to type++ and ensure future support beyond the effort of its original authors. Another likely source of opposition is the necessary integration into a wider software ecosystem, in particular with shared libraries. We advise developers to favor Property 3 to ease adoption. However, this implies maintenance of the allow-list whenever dependencies change.

Following our experiments, we identified three patterns whose occurrence in a project make the adoption of type++ challenging. First, if a large subset of classes is shared between C and C++ code, the effort to synchronize the data layout is substantial (§VII). Second, instability might arise if the project relies on undefined behavior (*e.g.,* fiddling with the LSB of pointers). Finally, the type field in objects should be protected against arbitrary writes, as laid out in our threat model (§III). This requirement is not trivial and likely comes with additional performance and memory overhead. Nonetheless, we consider these additional security guarantees a worthy trade-off between performance and security.

type++ aims to remain close to C++. New features are unlikely to cause new incompatibilities as the language moves towards type and memory safety as advocated by type++. From our experience, the effort of porting a codebase to type++ correlates negatively with how modern the codebase is. Codebases that expanded from C projects with little modernization take longer to port. As time passes, we argue that new projects will naturally follow type++ properties.

Interaction with C code. Projects may combine C and C++ code (*e.g.,* Blender, OMNeT++, Chromium). When compiling C code, the compiler must consider that some structures contain RTTI metadata, otherwise, the C code might misuse objects coming from C++ functions. To solve this problem, we have to synchronize the data layout across C and C++ modules. We envision three approaches: (i) generate new headers containing structures with an additional `vptr` only for C modules, (ii) modify the structures' data layout definition in LLVM, or (iii) add a Clang plugin that manipulates the C AST generation and adds a `vptr` to the instrumented classes. As cross-language interactions are rare, we opted to modify the structures' data layouts, *i.e.,* option (i). Note that the compiler can either change these types automatically or emit a warning for the developer.

Legacy code libraries. When interacting with legacy precompiled libraries, we cannot assume that their code is aware of the class modification. Here, we have two options: (i) we automatically infer the shared structures between legacy and type++ code and apply Property 3, ensuring equal data layouts for them in both code sections; (ii) we rewrite the legacy binary to adjust the field offsets. Neither option is perfect. As a fallback, we allow the programmer to manually specify classes shared with legacy code for Property 3.

Esoteric memory allocations. Some programs allocate contiguous memory regions that contain many objects in sequence. Our compiler automatically infers which objects the program is allocating and adds their constructors accordingly (see §V-C). To automatically locate unconventional allocators, an analysis can identify allocators that (i) cast to a specific known class, and (ii) the allocated size does not match the class size; *e.g.,* `A* a != (A*)malloc(sizeof(A) + len)`. As a sanity check, type++ separately reports any cast checks where the vtable pointer was uninitialized, highlighting the presence of unhandled allocators.

Functionality guarantee. It is key to maintain functionality when porting code from C++ to type++. If the changes mandated by type++ are not enforced, under specific conditions, the execution may deviate from the intended one. For example, in Listing 3, the comparison at Line 28 switches from `false` to `true`, changing the program's control flow. From our experience, patterns resulting in functionality difference remain rare; they are caused by unhandled C/C++ interactions as detailed above. To prevent such issues, the developer should investigate warnings reported by type++. Having a comprehensive test suite adds extra confidence but does not replace a proper investigation of the conflicting patterns.

Usage as a mitigation. Due to its low runtime overhead, type++ can be deployed as a mitigation in production to detect and protect against type confusion attacks. To this end, developers should ensure that the latest mitigations against memory safety vulnerabilities (*e.g.,* CFI) are deployed. This reduces the risk of attacks outside our threat model (§III) from modifying the type expected in the check or from skipping the type check completely. Additionally, type++ must be configured so that the program aborts when type++ detects a type confusion.

Supporting unrelated casts. Since Property 1 requires all classes to contain RTTI, type++ could enable type checks on all unrelated casts. This would guarantee full type safety in regards to all possible type confusions. This has a higher impact on performance as more type checks are performed due to the ubiquity of unrelated casts. Additionally, the porting effort increases as more classes need to be instrumented for Property 2. Finally, these classes are more likely to exhibit code smell since they already rely on the ill-advised `reinterpret_cast` operator [4]. We leave supporting unrelated casts for classes not involved in any downcast as future work.

Type confusions in non-C++ code. With Property 1, type++ guarantees the absence of downcasting type confusion in C++ code but cannot protect code in another language. Type confusion errors may still occur in non-C++ code, particularly C. Additionally, in the case of JavaScript engines, multiple type confusion vulnerabilities were reported in JavaScript JIT-generated code due to incorrect type tracking. We consider these issues orthogonal to type++ efforts.

VIII. Related work

type++ is a dialect that mitigates type confusion during typecasting operations. The literature already has C/C++ dialects addressing such problems, *e.g.,* Cyclone [39] is a type-safe dialect for C extending standard C with a set of protections that inspired other programming languages such as Rust [40] and Project Verona [41]. Likewise, Necula & al. introduced CCured [38], another C type-safe dialect that is similarly incompatible with the C++ specification. On the contrary, type++ is explicitly designed to overcome C++ typecasting limitations. DeLozier et al. introduced Ironclad C++ [29], which enforces type safety in C++ programs. However, Ironclad C++ relies on the developers to manually adapt all classes to make them compatible with `dynamic_cast`. More recently, UNCONTAINED [42] looked for incorrect casts of C structure embeddings in the Linux kernel. Another approach for type safety relies on checkers to validate RTTI information at runtime. For instance, LLVM-CFI [6] and UBSan [43] perform type checks at runtime for polymorphic objects involved in unsafe casts. However, both tools only deal with polymorphic classes while ignoring Plain Old Data objects. Conversely, type++ validates both non-polymorphic and polymorphic classes. Moreover, UBSan requires heavy code modifications hindering deployment. Alternative approaches against type confusion use checks based on disjoint metadata structures [8], [9], [17], [10]. This introduces a large overhead and suffers from a high rate of false positive. On the contrary, type++ blocks unsafe casting efficiently and effectively by design, as evaluated in §VI-B. Concurrent work [44] has looked at reducing the overhead by not adding type checks if the developer already implemented their own. This approach is orthogonal to type++ and hints at further possible performance improvements. Orthogonal efforts have looked at reducing the cost of the `dynamic_cast` type verification [32]. As type++ reuse this type verification implementation, these improvements would show an even greater performance benefit once deployed on top of type++.

IX. Conclusion

We introduced type++, a new C++ dialect that explicitly assigns RTTI to all classes in a program. type++ allows fast runtime type checks thus overcoming runtime overhead, low coverage, and imprecision of previous works.

Our study on the effort to port standard C++ programs to type++ shows that our dialect requires changing only 0.16% of a program LoC in the worst case. Over SPEC CPU2006 and CPU2017, type++ incurs a negligible performance overhead (*i.e.,* 0.94%—two orders of magnitude faster than HexType) while protecting all the down-to cast operations (unlike previous works that are limited to only a subset). We find 122 type confusion errors in SPEC CPU2006 and CPU2017, 14 of them otherwise unobserved. Finally, evaluating type++ on Chromium results in an acceptable overhead (1.42%). All our findings, code, the material to replicate our experiments, and technical reports describing the type confusions found are publicly released.

Acknowledgement

We thank the anonymous paper and artifact reviewers, and our shepherd for their feedback as well as Bhargav Kommireddy for his help during his internship. This work was supported, in part, by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), SNSF PCEGP2_186974, Fondation Botnar, and a gift from Intel corporation.

References

[1] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 745–762.

[2] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block Oriented Programming: Automating Data-Only Attacks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1868–1882. [Online]. Available: https://doi.org/10.1145/3243734.3243739

[3] N. D. Matsakis and F. S. Klock, "The rust language," *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103–104, 2014.

[4] H. Sutter, B. Stroustrup, and other contributors, "C++ Core Guidelines," https://github.com/isocpp/CppCoreGuidelines/commit/6156e957827599f2fcaa5401ebb1668ae9edcdc8, 2015.

[5] H. Sutter, "C++ safety, in context," https://herbsutter.com/2024/03/11/safety-in-context/, 2024.

[6] P. Muntean, M. Neumayer, Z. Lin, G. Tan, J. Grossklags, and C. Eckert, "Analyzing control flow integrity with LLVM-CFI," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ser. ACSAC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 584–597. [Online]. Available: https://doi.org/10.1145/3359789.3359806

[7] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, "Preventing Use-after-free with Dangling Pointers Nullification," in *22th Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8 - 11, 2015*, 2015.

[8] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe, "TypeSan: Practical type confusion detection," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 517–528.

[9] Y. Jeon, P. Biswas, S. Carr, B. Lee, and M. Payer, "HexType: Efficient Detection of Type Confusion Errors for C++," in *CCS*, 2017.

[10] C. Pang, Y. Du, B. Mao, and S. Guo, “Mapping to bits: Efficiently detecting type confusion errors,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 518–528.

[11] A. Loginov, S. H. Yong, S. Horwitz, and T. Reps, “Debugging via run-time type checking,” in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2001, pp. 217–232.

[12] LLVM Developers, “TySan: A type sanitizer,” https://lists.llvm.org/pipermail/llvm-dev/2017-April/111766.html.

[13] G. J. Duck and R. H. Yap, “EffectiveSan: type and memory error detection using dynamically typed C/C++,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[14] J. L. Henning, “SPEC CPU2006 benchmark descriptions,” *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[15] J. Bucek, K.-D. Lange, and J. v. Kistowski, “SPEC CPU2017: Next-generation compute benchmark,” in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 41–42.

[16] M. Stevanovic, *Advanced C and C++ Compiling*, 1st ed. USA: Apress, 2014.

[17] B. Lee, C. Song, T. Kim, and W. Lee, “Type casting verification: Stopping an emerging attack vector,” in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 81–96.

[18] D. J. Bernstein and F. Denis, “Libsodium secure memory,” https://libsodium.gitbook.io/doc/memory_management, 2014.

[19] P. Mao, E. V. Boschung, M. Busch, and M. Payer, “Exploiting Android’s Hardened Memory Allocator,” in *Proceeding of the 18th USENIX WOOT Conference on Offensive Technologies*, 2024.

[20] LLVM Developers, “LLVM-CFI: cast checking strictness,” https://clang.llvm.org/docs/ControlFlowIntegrity.html#cfi-strictness.

[21] J. Järvi, J. Willcock, and A. Lumsdaine, “Concept-controlled polymorphism,” in *Generative Programming and Component Engineering: Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003. Proceedings 2*. Springer, 2003, pp. 228–244.

[22] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama, “Towards optimization-safe systems: Analyzing the impact of undefined behavior,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 260–275.

[23] Windows Developers, “_msize,” https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/msize?view=msvc-170, 2023.

[24] Apple Developers, “MALLOC_SIZE,” https://developer.apple.com/library/archive/documentation/System/Conceptual/ManPages_iPhoneOS/man3/malloc_size.3.html, 2006.

[25] T. Mens and T. Tourwé, “A survey of software refactoring,” *IEEE Transactions on software engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[26] cppreference.com, “C++ Union,” https://en.cppreference.com/w/cpp/language/union.

[27] B. Stroustrup, “How do I define an in-class constant,” https://www.stroustrup.com/bs_faq2.html#in-class.

[28] S. Barati and M. Saboff, “Introducing the Jetstream 2 benchmark suite,” https://webkit.org/blog/8685/introducing-the-jetstream-2-benchmark-suite, 2019.

[29] C. DeLozier, R. Eisenberg, S. Nagarakatte, P.-M. Osera, M. M. Martin, and S. Zdancewic, “Ironclad C++ a library-augmented type-safe subset of C++,” *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 287–304, 2013.

[30] Clang Developers, “Clang 19 documentation,” https://clang.llvm.org/docs/DiagnosticsReference.html#wuninitialized.

[31] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight Jr, and A. 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 & communications security*, 2013, pp. 721–732.

[32] M. Gibbs and B. Stroustrup, “Fast dynamic casting,” *Software: Practice and Experience*, vol. 36, no. 2, pp. 139–156, 2006.

[33] S. Macintyre-Randall, “Enforcing C++ type integrity with fast dynamic casting, member function protections and an exploration of C++ beneath the surface,” Ph.D. dissertation, University of Kent, 2023.

[34] Google, “Protocol Buffers,” https://developers.google.com/protocol-buffers/.

[35] ——, “What is V8?” https://v8.dev/.

[36] Chromium Developers, “Control Flow Integrity - The Chromium Projects,” https://www.chromium.org/developers/testing/control-flow-integrity/.

[37] M. Payer, “Type Confusion: Discovery, Abuse, Protection,” https://hexhive.epfl.ch/publications/files/18SyScan360-presentation.pdf, Singapore, 2017.

[38] 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)*, vol. 27, no. 3, pp. 477–526, 2005.

[39] 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, General Track*, 2002, pp. 275–288.

[40] S. Klabnik and C. Nichols, *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[41] Microsoft, “Project Verona,” https://github.com/microsoft/verona.

[42] J. Koschel, P. Borrello, D. C. D’Elia, H. Bos, and C. Giuffrida, “Uncontained: Uncovering Container Confusion in the Linux Kernel,” in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 5055–5072. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/koschel

[43] Clang Developers, “UBSan,” https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html#available-checks.

[44] Y. Zhai, Z. Qian, C. Song, M. Sridharan, T. Jaeger, P. Yu, and S. V. Krishnamurthy, “Don’t Waste My Efforts: Pruning Redundant Sanitizer Checks of Developer-Implemented Type Checks,” in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[45] S. Groß, S. Koch, L. Bernhard, T. Holz, and M. Johns, “FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities,” in *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/fuzzilli-fuzzing-for-javascript-jit-compiler-vulnerabilities/

[46] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “Addresssanitizer: A fast address sanity checker,” 2012.

[47] T. Plachetka, “POV Ray: persistence of vision parallel raytracer,” in *Proc. of Spring Conf. on Computer Graphics, Budmerice, Slovakia*, vol. 123, 1998, p. 129.

[48] Apache Software Foundation, “Xalan-C++ version 1.10,” https://xml.apache.org/xalan-c/.

Appendix A
Manuscript Appendix

We present interesting use cases and an experiment demonstrating the capabilities of type++ as a sanitizer during fuzzing.

A. *type++ as sanitizer*

We conduct a preliminary analysis to evaluate type++ as a lightweight sanitizer in a fuzzing campaign. For this experiment, we test V8 [35] version 9.0.257.29, the Chromium JavaScript engine. We compile V8 in two configurations: one applying Property 3 and protecting a random set of 32 classes (named *typepp*), and a second one with the default V8 fuzzing configuration (named *vanilla*). Fuzzilli [45], a mature JavaScript fuzzing tool compatible with V8 (version 0.9.3) serves as our fuzzer. The experiments were executed on an Intel machine with an i7-8700 @ 3.20GHz CPU and 64GB of RAM. The fuzzing campaign lasted for 12h, starting with an empty corpus. The *vanilla* configuration reaches 15% of branches as coverage, while the *typepp* one 8% of the branches. The difference in coverage is caused by new type confusion bugs detected by type++ that block the exploration of some V8 components. These bugs were not found by ASan [46] (the default sanitizer in the *vanilla* configuration), which cannot identify type confusion anomalies. ASan would only crash if a type confusion results in a later memory safety violation. We are triaging these issues together with the

Listing 5 A (simplified) type confusion example in Xalan-C++ from SPEC CPU2006.

```
class DOMTextImpl: public DOMNode {
  DOMNodeImpl fNode;
  ...
};

class DOMElementImpl: public DOMNode {
public:
  DOMNodeImpl fNode;
  ...
};

// DOMTextImpl at runtime
DOMNodeImpl *castToNodeImpl(const DOMNode *p) {
  DOMElementImpl *pE = (DOMElementImpl *)p;
  // works because the first
  // element is fNode
  return &(pE->fNode);
}
```

Chromium team. This experiment demonstrates that type++ can be used in a fuzzing campaign as an alternative sanitizer in combination with other tools such as ASan.

B. *Use case: POV-Ray*

POV-Ray [47] is a ray-tracing program that generates images from a text-based scene description. For POV-Ray, in both SPEC CPU2006 and CPU2017, we found improper use of C-style `structs` to simulate parent-child relationships. Specifically, POV-Ray contained a set of `structs` implemented as unrelated types (*i.e.,* they did not declare a parent class) but that were used as parent-child classes. The program casts those classes in a C-style fashion assuming that the memory layout of the two classes overlaps. Blindly casting two structures is undefined behavior since the compiler might apply an optimization over one `struct` layout. Therefore, we consider these C-style casts as bugs. When running, type++ immediately identified and reported the type confusion occurrences, we thus identified 113 program locations for SPEC CPU2006 and 99 program locations for SPEC CPU2017 that triggered a type confusion bug. We then used the information from type++ to infer the correct class hierarchy and modify the source code accordingly. In POV-Ray, we edited 123 LoC in SPEC CPU2006 and 131 LoC in SPEC CPU2017, corresponding to around of their codebase. All the modifications concerned the correction of 19 classes in which we included the proper parent class in the header files, while the program logic remained unchanged. These changes were necessary as the LLVM cast checks were crashing due to non-existent RTTI, highlighting the type confusion. Modifications to allow the cast check from LLVM to recover are left as future work.

C. *Use case: Xalan-C++*

Xalan-C++ is an XSLT processor originally developed by IBM [48]. The program is written in complex C++ and heavily overloads the class hierarchy to represent the DOM of an XML document. Moreover, Xalan-C++ implements a sophisticated memory allocation strategy based on `new()` primitives and `operator::new()` overloads to optimize the allocated memory. This section focuses on the SPEC CPU2006 version, but the same reasoning applies to SPEC CPU2017. We managed to compile Xalan-C++ with type++ and run the benchmark, type++ (as well as LLVM-CFI) reported around $9,000$ type confusion errors at seven unique locations. The type confusions were generated because Xalan-C++ was attempting to cast unrelated objects that shared part of their layout (similar to Blender and POV-Ray). Listing 5 contains an example of type confusion detected: `castToNodeImpl()` gets a `DINNode` object as an input and tries to cast it into a `DOMElementImpl` type. At runtime, the function receives a `DOMTextImpl` object, which is semantically correct since it is a child of `DOMNode`. However, even though the classes share the same parent, they belong to different branches in the class hierarchy, thus creating type confusion. Nonetheless, the code works because the two objects have `fNode` as a common first field.

Our proposed solution would use multiple inheritances, in particular, `castToNodeImpl()` should accept a new type, *e.g.,* `DOMConverted`, that is added as an additional parent to `DOMTextImpl` and `DOMElementImpl`. Moreover, `DOMConverted` should implement a virtual function, *e.g.,* `getNode()`, that is implemented in the subclasses (*i.e.,* `DOMTextImpl` and `DOMElementImpl`) and returns the field `fNode` without relying on brittle memory layout assumptions. Adopting this technique would allow Xalan-C++ to avoid type confusions. Since the modification requires deep code modification (due to the intertwined relationship with the custom allocators), we leave patching these issues for future work. In this case, if the code base is too old, it is possible to have a trade-off between security and usability by leaving only 7 locations as possible attacker-surface and exempting those from runtime checking.

Appendix B
Artifact Appendix

In this appendix, we provide the requirements, instructions, and further details necessary to reproduce the experiments from our paper (DOI: 10.14722/ndss.2025.23053).

A. *Description and requirements*

The artifact contains the material to reproduce the results: Compatibility analysis (§VI-A – Table I), Performance Overhead & Security evaluation (§VI-B & §VI-C – Table II & Table V), Use case: Chromium (§VI-D – Table VI). This material is released under the Apache License 2.0, in line with the LLVM project type++ builds upon.

1) *Accessing the artifact*: We release the artifact on a public GitHub repository `https://github.com/HexHive/typepp`. While the typepp branch contains the latest version of the code, the `ndss-25-artifacts` tag contains the exact version of the code that was submitted for review in the artifact evaluation. Additionally, the code is available on Zenodo with the DOI:10.5281/zenodo.13687049.

2) *Hardware dependencies*: The artifact requires a machine with at least 128GB of RAM and a 1TB disk. 16GB of RAM is sufficient to run the SPEC CPU evaluations.
3) *Software dependencies*: The artifact was tested on Ubuntu 20.04 and require the ability to run Docker containers. An active internet connection is also necessary for the Chromium evaluation. Additionally, `curl, git, docker`, and `pip` should also be installed.
4) *Benchmarks*: The artifact requires a copy of the SPEC CPU 2006 & 2017 benchmarks.[2]

B. Artifact installation

The initial step is to clone the repository and build the Docker image. As the artifact is provided on top of a fork of the LLVM project, we recommend to only proceed to a shallow clone of the last 100 commits. This can be achieved by running the following bash command: `git clone $REPO --single-branch --branch typepp --depth 100 LLVM-typepp`. Additionally, please run `pip install -r requirements.txt` from inside the repo to install dependencies.

Each experiment is encapsulated in one or multiple Docker container. The Dockerfile is available at the root of the artifact repository. We do not provide support for running the experiments locally.

C. Experiment workflow

Our artifact aims at reproducing the results from three experiments presented in the paper. The first aims at evaluating the compatibility of type++ with existing software and quantifies the number of LoC changes necessary to port a C++ project. The second experiment evaluates the performance overhead of type++ over standard C++ with the help of the SPEC CPU 2006 and 2017 benchmarks as well as quantifies the added security guarantees provided by type++. Lastly, the third experiment demonstrate the performance and security benefits of type++ after a partial support of the Chromium browser.

We propose to run this experiments sequentially as they are presented in the paper. The artifact provides scripts to run the experiments and collect the results. The scripts will also generate tables similar to the ones presented in the paper.

More complete and detailed instructions, as well a minimal example, are available in the README file of the repository. We highly recommend following the instructions there as copying and pasting the commands from this document might introduce errors.

D. Major claims

- (C1) *Compatibility*: type++ is compatible with existing C++ codebases with a few minor changes. This is showcased in experiment E1 which runs our compatibility analysis on the SPEC CPU benchmarks. These results are presented in Table I in the paper.
- (C2) *Performance Overhead & Security Guarantees*: type++ incurs a negligible performance while protecting a vast amount of additional casts. Our experiment E2 highlights this trend on the SPEC CPU benchmarks. In the paper, the results are available in Table II. The overhead numbers can slightly differ from the ones in the paper due to the different hardware configurations, but we expect the global trend across the benchmark to remain consistent.
  Due to time constraints, we do not provide automatic scripts to run and extract the data from our competitors (i.e., HexType, TypeSan, and EffectiveSan) that would be necessary for Table III. We, however, provide instructions on how to run these experiments.[3]
- (C3) *Type checking cost breakdown*: To better assess the cost of the different type checking methods, we designed a micro-benchmark. The results are presented in Table V and can be reproduced through the experiment E3.
- (C4) *Use case: Chromium*: type++ can be used to protect large code bases. As a proof of concept, we partially protect the Chromium browser. The results are presented in Table VI and can be reproduced through the experiment E4.

E. Evaluation

In this section, we provide the detailed steps to run the experiments and process the results to get the tables presented in the paper. Overall, this process requires around two to three days of computation time on a powerful server. These instructions are also available in the README file of the artifact repository.

Experiment 1 (E1) - Claim (C1): Compatibility Analysis: [2 humans minutes + 2 compute-hours] The experiment evaluates the porting effort necessary for SPEC CPU 2006 and 2017 benchmarks and quantify the benefits of Property 2 over Property 1. The experiment consists of compiling the benchmarks with type++ to first collect the classes to instrument for Property 2 and then run the analysis to collect the warnings emitted for both properties.

[Preparation] Ensure that the two SPEC CPU benchmarks `.iso` are available at the root of the cloned repository.

[Execution] Run the commands:

```
# Build the docker images and then run two
↪  analysis (Property 1 and Property 2) on
↪  the SPEC CPU benchmarks (around 2hours).
# Expected output: Different logs highlighting
↪  first the Docker builds and then the
↪  benchmarks compilation and analysis.
↪  Finally, a table similar to Table I will
↪  be printed.
./table1.sh
```

[Results] Upon completion, the script will generate a table identical to Table I. The file `analysis_result_test.tex` contains the results.

Experiment 2 (E2) - Claim (C2): Performance Overhead & Security Guarantees: [2 humans minutes + 15 compute-hours] This experiment runs the SPEC CPU benchmarks

[2] https://www.spec.org/cpu2006/ and https://www.spec.org/cpu2017/

[3] https://github.com/HexHive/typepp/blob/typepp/COMPETITORS.md

with different flavors of cast checking. First, a baseline is established by running the benchmarks with no cast checking. Then, the benchmarks are run with type++ and LLVM-CFI to measure the overhead. Lastly, an extra run for both type++ and LLVM-CFI is performed to measure the number of cast protected.

[Preparation] Again, ensure that the two SPEC CPU benchmarks '.iso' are available at the root of the cloned repository.

[Execution] In a shell, run the following command:

```
# Build the docker images and then run all the
↪  benchmarks variation (around 15hours).
# Expected output: Different logs highlighting
↪  first the Docker builds and then the
↪  benchmarks compilation and execution.
↪  Finally, a table similar to Table II will
↪  be printed.
./table2.sh
```

[Results] Upon completion, the script will generate a table similar to Table I. There might be variations up to 10% in the performance and memory overhead numbers due to the different underlying machine. In particular, in a shared environment, the overhead might show inconsistencies. The number of casts protected should remain identical to the one presented in the paper.

Experiment 3 (E3) - Claim (C3): Type checking cost breakdown: [2 humans minutes + 20 compute-minutes] This experiment evaluates the cost of the different operations in the type checking process of HexType and type++.

[Preparation] No specific preparation is required.

[Execution] In a shell, run the following command:

```
# Build the docker images and then run the
↪  micro-benchmark (around 20minutes).
# Expected output: Different logs highlighting
↪  first the Docker build and then the cost
↪  of the different operations.
./table5.sh
```

[Results] Upon completion, the script will generate a table similar to Table V. The numbers presented should exhibit similar ratios to the ones in the paper.

Experiment 4 (E4) - Claim (C4): Use case: Chromium: [2 humans minutes + 36 compute-hours] This experiment evaluates the performance of Chromium with partial protection by type++ and LLVM-CFI. It also outputs the number of casts protected by both tools. This experiment will build Chromium first with no protection, then with LLVM-CFI, and finally with type++. Then, it will run the JetStream benchmark and, therefore, require a working internet connection. As building and linking Chromium is a time-consuming and resource-intensive task, we recommend running this experiment on a machine with plenty of RAM and in `tmux` or `screen` session to avoid interruptions.

[Preparation] Please run the following script:

```
# Fetch the Chromium source code and the
↪  modifications we applied to it.
./table6_requirements.sh
```

[Execution] In a shell, run the following command:

```
# Build the docker images, inlcuding fetching
↪  the dependencies of Chromium, and then
↪  build Chromium itself. Lastly, run the
↪  JetStream benchmark (around 36hours).
# Expected output: Different logs highlighting
↪  first the Docker builds, the fetching of
↪  depencies and then the Chromium
↪  compilation and execution. Finally, a
↪  table similar to Table VI will be printed.
./table6.sh
```

[Results] Upon completion, the script will generate a table similar to Table VI. The benchmark scores presented should exhibit similar ratios to the ones in the paper. The number of casts protected is expected to be different as the Chromium execution is not deterministic but should remain in the same magnitude.