Sourcerer: channeling the `void`

Nicolas Badoux[1], Flavio Toffalini[1,2], and Mathias Payer[1]

[1] EPFL, Lausanne, Switzerland
[2] Ruhr Universität Bochum, Bochum, Germany

Abstract. Type confusion vulnerabilities occur when a program misinterprets an object as an incompatible type. Such errors result in undefined behavior and can lead to illegal memory accesses undermining security. For compatibility reasons, the C++ programming language tolerates insecure type conversions, delegating the responsibility for assuring an object's type to the developer. Sanitizers help developers detect and patch vulnerabilities during dynamic testing, *i.e.,* before they reach production environments. However, current type confusion sanitizers either incur prohibitive runtime overheads, or fail to check all casts. In particular, casts from `void*` have historically been overlooked due to challenges in recognizing the underlying object's type, thus leading to incomplete type coverage.
We introduce Sourcerer, a new sanitizer that correctly and fully traces and recognizes *all* type confusions, in particular, casts from unrelated types and `void*`. Sourcerer enriches the classes involved in a cast with runtime type information to perform precise runtime checks. When compared with the state-of-the-art, Sourcerer expands type coverage to *all* cast operations, 8,507M additional casts on the SPEC CPU2006 and CPU2017 benchmarks—a 118% increase—with reasonable average performance overhead of 5.14%. Additionally, we conduct an ablation study to understand what causes this runtime overhead and showcase a fuzzing campaign finding six bugs, highlighting the improved bug-finding capabilities when Sourcerer is deployed.

Keywords: Sanitizer, type confusion, C++.

1 Introduction

C++ offers speed and flexibility at the compromise of not enforcing strong type and memory safety guarantees. For example, the lack of type safety allows an object to be converted to any type at zero cost, trading versatility for the risk of *type confusion*, which occur when a program misinterprets an object as an incompatible type, leading to undefined behavior. Type confusion vulnerabilities have been consistently exploited in the wild (e.g., CVE-2024-30357 in Foxit, CVE-2025-24129 in iOS, CVE-2024-9859, or an $11,000 vulnerability #329781390 in Chromium) and are classified under Common Weakness Enumeration CWE-704 [22]. Identifying and fixing such violations early in the development cycle is key to prevent their active exploitation. To this end, automatic software testing (*fuzzing*) is a strong ally in identifying unexpected behavior. However, type confusions are challenging to detect as they may not trigger a crash and the compiled code no longer has any notion of the *C++ types*. Even though recognizing type confusion is complicated, their detection is crucial for overall software security. To address this, *sanitizers* are used during automatic software testing to report unintended software behavior. For C++, sanitizers exist for many memory-related vulnerabilities, however, systematically detecting *all type confusions* remains a challenge for which no sanitizer has been widely deployed so far.

The underlying difficulty of designing a robust type confusion sanitizer for C++ stems from the language type conversion mechanisms. These are divided into three categories: (i) implicit conversion, *e.g.,* `bool` to `int`, (ii) user-defined copy constructors, *e.g.,* `To(From&)` for classes `From` and `To`, or (iii) explicit casting operators, *e.g.,* `reinterpret_cast`. While implicit and developer-defined conversions are safe, explicit casts allow for arbitrary conversions, potentially compromising type safety. In practice, aside from `dynamic_cast`, for which the standard mandates runtime checks, the other explicit cast operators, `static_cast` and `reinterpret_cast`, rely on developers to guarantee the conversion's validity. Due to a lack of security awareness, outdated fear of performance overhead, or C-style programming, current codebases still contain potential incorrect type conversions. To validate a type conversion, the relationship between the source and destination types must be checked at runtime. Static checks cannot be certain of the source type as C++ allows for type aliasing either through `void` pointers used as a generic pointer type or via polymorphism (which allows referring to derived objects through their parents). While these patterns allow for flexible code, they hinder the possibility of static validation of type conversions and increase the type system complexity, therefore putting the program at risk of type confusion. As C++ does not provide type information at runtime, most objects remain glorified C-style structs. Only a subset of all objects, those with virtual functions, are equipped with a runtime type identifier as mandated by the C++ standard. Indeed, only those objects can benefit from runtime checks (*e.g.,* `dynamic_cast`) to guarantee type safety while conversions of other object types remain prone to type confusion.

Different approaches exist to prevent type confusions. First, migrating to a type- and memory-safe language like Rust [21] removes many vulnerability classes, including type confusions. However, rewriting a project is costly and risks introducing new bugs such as logic ones. While safe languages is the preferred solution for new projects, porting old code bases is not always feasible due to constrained development resources. Switching to a C++ dialect with more safety guarantees, like the security profiles in the C++ Core Guidelines [26] is less demanding but still requires a deep understanding of the dialect and significant code changes. For example, to prevent type confusions, the C++ Core Guidelines mandate the removal of all unsafe cast operators [27]. If a rewrite is out-of-scope, the alternative is to detect and/or prevent type confusions at runtime. Throughout the years, many type confusion sanitizers [10, 14] tried to trace and check object lifetimes. However, they incur high runtime costs and are prone to false positives [23]. LLVM Control Flow Integrity (CFI) [28] takes another approach by relying on Runtime Type Information (RTTI), which incurs a low runtime overhead but offers an incomplete detection as it is restricted to polymorphic objects. type++ [1], a C++ dialect, extends LLVM-CFI's detection to all derived casts with only small modifications of the codebase. After deep inspection, however, we observe that type++ only partially supports unions due to design limitations in the RTTI initialization. Thus, existing tools lack a viable method to detect all cast operations. Most importantly, current sanitizers are limited to derived casts, *e.g.,* between a parent and a child, while leaving the majority of unrelated casts, if not all, unverified (*e.g.,* from `void*`). For example, type++ leaves 7,151M casts unprotected—(45.67% of all unrelated casts across both SPEC CPU2006 and CPU2017). In contrast, EffectiveSan [7] addresses all cast operations by encoding type information as low-fat pointers but requires heavy-weight modifications of all allocators and checks the type at each pointer dereference (which, compared to casts, is a very frequent operation), causing an unnecessarily high runtime overhead of 49%. *Reliably detecting all type confusions with a low runtime cost remains, therefore, an open challenge.*

In this paper, we propose Sourcerer, a novel sanitizer capable of detecting *all* type confusions by enforcing a runtime type check at every explicit cast operation. The novelty of Sourcerer lies in embedding inline Runtime Type Information (RTTI) into all object types involved in unrelated casts, overcoming the restriction to derived casts that limits state-of-the-art solutions, *e.g.,* type++ and LLVM-CFI. To achieve this goal, Sourcerer introduces *RTTIInit*, a new C++ object initializer responsible solely of setting inline RTTI. With the introduction of RTTIInit, Sourcerer alleviates porting issues and overcomes the limitations of previous dialects, which forcibly injected default constructors and conflicted with the C++ standard. Furthermore, the introduction of RTTIInit solves the type initialization at union activation, another unsupported feature in existing dialects. As a result, Sourcerer performs runtime type validation for each cast from a non-generic types (*i.e.,* developer-defined types) and generic pointers (*i.e.,* `void*` or integral types), finally unlocking *full type testing* in C++ programs.

Sourcerer's pipeline consists of two stages. First, a static analysis identifies all classes involved in cast operations and helps the developer to adhere to the dialect. Then, our compiler generates an executable with the initializer responsible for setting the inline type information and the necessary type checks. We evaluate Sourcerer on the SPEC CPU2006 [11] and SPEC CPU2017 [3] benchmarks[3] and conduct a fuzzing case study on a leading C++ project, OpenCV [2], showcasing the ability to use Sourcerer to find illegal unrelated casts. Overall, Sourcerer detects all cast operations in the SPEC CPU benchmarks and incurs, on average, only a 5.14% performance overhead. Supporting objects involved in unrelated casts requires additional code changes. We deem the necessary adaptation of 453 out of 2,040K LoC reasonable with respect to the 4.95× more classes instrumented compared to the instrumentation of only derived casts. In

[3] When using SPEC CPU, we refer to the SPEC CPU2006 and CPU2017 benchmarks.

our evaluation, we conduct an ablation study to understand the root cause of the observed overhead. Our study concludes that the *Application Binary Interface* (ABI) change is the main cause of overhead highlighting the possibility of drastically improving performances by modernizing specific casts. In terms of security impact, Sourcerer identifies 152 type confusions in the SPEC CPU2006 and CPU2017 benchmarks. Finally, we conduct a fuzzing campaign in which we deploy our sanitizer on OpenCV [2], discovering six unrelated type confusion bugs, all missed by the state-of-the-art competitors.

Overall, the main contributions of Sourcerer can be summarized as follows:

- Quantifying *unrelated* casts so far hidden from state-of-the-art sanitizers.
- Proposing an extension to the type++ dialect to check *all* cast operations.
- A new initializer, RTTIInit, that sets object's type information, increasing the compatibility with the C++ standard and reducing the runtime cost.
- A thorough evaluation of Sourcerer against the state-of-the-art.
- A case study showcasing how Sourcerer can be used to detect vulnerabilities in real-world software through fuzzing campaigns on well-tested software.

We release the source code of Sourcerer and the documentation to replicate our experiments as open-source at `github.com/HexHive/Sourcerer`.

2 Background

In this section, we introduce the key concepts behind Sourcerer's approach, focusing on how C++ handles type conversion and distinguishes unrelated from derived casts. Additionally, we introduce the concept of type confusion vulnerabilities (CWE-704 [22]).

2.1 C++ casting

Casting—adjusting the type of objects—allows for flexible and generic code. While implicit conversions (*e.g.,* conversions defined by the C++ standard like `char` to `int`) or the developer-defined ones are safe, explicit casts pose the risk of undefined behavior. Specifically, unsafe casts have two origins: *derived or down casts* occur when the resulting type inherits from the source class while *unrelated casts* encompass conversions between two types not related by inheritance (*e.g.,* `void` or other generic pointer type). The reverse of a derived cast, *i.e.,* from a base to a derived type, is known as an *upcast* and is implicit and therefore safe.

Both unrelated and derived casts can be safe. For example, casting from `void*` is well-defined if the pointed memory was previously cast from the desired type (*e.g.,* Line 27 in Listing 1). Similarly, a derived cast is legal if the object is of the derived type or one of its descendant (*e.g.,* Line 19 in Listing 1). Outside these cases, the result of the cast is undefined. For example, in Line 24 in Listing 1 a `Sibling` object referenced as a `Base` is illegally cast through `static_cast` to a `Drvd` pointer. After such a cast, the program might try to access the `y` attribute of the `Drvd` object which is out-of-bound as `Sibling` objects are smaller. Such illegal casts are referred to as *type confusions* and might lead to memory

```
 1 class Unrelated {char w;};
 2 class Base {
 3 public:
 4   int x;
 5   virtual ~Base()=default;
 6 };
 7 class Drvd:public Base{
 8   int y[3];
 9 };
10 class Sibling:public Base{
11   double z;
12 };
13 int main() {
14   Unrelated* u = new Unrelated();
15   Drvd* d = new Drvd();
16   Sibling* s = new Sibling();
17   // Safe casts: no need for verification //
18   Base* b = static_cast<Base*>(d);
19   Drvd* d2 = dynamic_cast<Drvd*>(b);
20   if(d2 == nullptr) return 1; //if cast fails
21   Base* b2 = static_cast<Base*>(s);
22   void* v = d; // Implicit generic cast
23   // ------------ Derived cast ---------- //
24   d2 = static_cast<Drvd*>(b2); // Illegal
25   d2 = reinterpret_cast<Drvd*>(b);
26   // ----------- Unrelated cast --------- //
27   d2 = static_cast<Drvd*>(v);
28   d2 = reinterpret_cast<Drvd*>(s);// Illegal
29   Unrelated* u2 = (Unrelated*) d; // Illegal
30   return 0;
31 }
```

Listing 1: Examples of derived (lines 24 & 25) and unrelated casts (lines 27–29). We omit statistics on safe casts (upcast and `dynamic_cast`, lines 18 & 19).

corruption. To avoid such errors, C++ developers need to maintain a mental model of the actual object types. To do a type conversion, the C++ standard mandates the use of one of the following explicit cast operators if the conversion is not inherently safe—*e.g.,* neither implicit nor defined by the developer:

- `dynamic_cast` handles only derived casts. It queries, at runtime, the type information of the object to verify the conversion's safety. Therefore, the source and destination type must be polymorphic to ensure the presence of RTTI. `dynamic_cast` is the *only* safe runtime cast operator in C++.
- `static_cast` provides only compile-time checks. For derived casts, the compiler checks that the source and destination types are part of the same type hierarchy. Notably, this fails to prevent all illegal derived casts. Finally, `static_cast` can convert `void` pointers to another type, without any checks. In contrast to `dynamic_cast`, `static_cast` lacks type safety guarantees.
- `reinterpret_cast` allows developers to interpret the bytes of an object as a new type. This is inherently unsafe and breaks type safety guarantees as no runtime checks are executed. Typically, this operator is used for unrelated casts, but it also supports derived cast.

Additionally, `const_cast` allows changing the qualifiers but not the object's type. This might result in undefined behavior but is not linked to type confusions. Lastly, C++ supports C-style casts which are translated to the above compatible cast operator with the highest safety guarantees. We will, therefore, not explicitly consider C-style casts in the rest of the paper.

2.2 type++ Limitations

Research for detecting type confusions has evolved with diverse mechanisms being explored. Badoux et al. [1] propose a new C++ dialect mitigating, by design, type confusions in derived casts. By extending all derived cast objects with runtime type information (RTTI), type++ can protect each derived cast, maintaining their type safety throughout the program execution. Adding RTTI into these objects changes the *Application Binary Interface* (ABI) resulting in some (limited) porting effort. To set the RTTI, their implementation artificially injects default constructors to all classes involved in down-to casts. This solution conflicts with some C++ idioms like if it is marked as `deleted` or when constructor calls are actively avoided (§5.5).

Crucially, type++ stops short of being *completely type safe* as it only partially protects unrelated casts—slightly less than half in the case of the SPEC CPU benchmarks. Moreover, instrumenting only a subset of cast classes breaks the implied compatibility between some classes resulting in unnecessary porting effort. We hypothesize that the reliance on the constructor to set RTTI is the root cause of these porting issues.

3 Threat Model

Sourcerer is a sanitizer for detecting *all*, derived and unrelated, type confusions in C++. In particular, when a type confusion bug is triggered during testing, we expect Sourcerer to detect the type safety violation. We assume a correct implementation of our compiler and the program to be correctly ported to the type++ dialect. Specifically, Sourcerer is not designed for an adversarial scenario due to possible false negatives if an attacker can leak and set type information. We refer to §9 for a thorough discussion. In summary, our threat model aligns with the ones from previous type confusion sanitizers [14].

4 Challenges

Upon careful evaluation of related works, we identified several unsolved challenges for type confusion sanitizers. First, the state-of-the-art, *e.g.,* type++ and HexType, offer incomplete type safety as they miss most unrelated casts. Only EffectiveSan offers theoretically full type safety but at an unnecessary high runtime cost, creating the second challenge we aim to address. Lastly, we identified only partial support for unions in type++ and EffectiveSan while Sourcerer provides complete support.

- **Unrelated casts**: To detect all type safety violation, a sanitizer needs to cover *all* cast operations, including unrelated casts.
- **Performance overhead**: For a sanitizer to be widely adopted, it should have a minimal performance overhead.
- **Union support**: C++ unions allow different types to refer to the same memory. This is a problem for type confusion sanitizers as they need to track the union type in memory.

```
mov     $0x1,%edi
call    47340 <malloc@plt>
```

```
mov     $0x10,%edi
call    47340 <malloc@plt>
call    46f80 <_ZN9RTTIInitUnrelated>
...
<_ZN9RTTIInitUnrelated>:
lea     0x1ad9(%rip),%rax #vtable address
mov     %rax,(%rdi)
ret
```

Listing 2: Assembly code of the `malloc`-ation of an `Unrelated` object from Listing 1 without and with RTTIInit.

5 Sourcerer's Design

In this section, we introduce the core concepts allowing Sourcerer to address these challenges. First, we lay out which classes need type information to truly check *all* casts. Then, we describe *RTTIInit*, an optimized inline type information initializer reducing type++ dialect divergence and lowering the performance overhead by 3× in comparison with the other complete type confusion sanitizer, EffectiveSan. Additionally, we explain the key properties of RTTIInit allowing the support of unions. Finally, we list the idioms unsupported by earlier work that Sourcerer handles, like templates for EffectiveSan.

5.1 Classes to Instrument

Sourcerer's core contribution is to check all casts. Specifically, Sourcerer instruments all the classes involved in any derived or unrelated casts, thus extending Property 2 laid out in the type++ paper [1]. Formally and in line with the properties defined in the type++ paper, we refer to this specialization as *Explicit Runtime Types For All Casts*:

Property 1 (Explicit Runtime Types For All Casts.) *Given all classes CS of a program P, Sourcerer associates a unique type T to each class $A \in CS$ if A is either the destination or the type of the source object of an explicit cast.*

This new property allows for the verification of all type casts while keeping the number of classes to instrument at a minimum. Due to the ubiquity of casts from `void*`, the sanitizer needs to instrument an increased number of classes—1,043 additional ones in the SPEC CPU benchmarks, a 5× increase compared to previous works [1] targeting only derived casts. The key insight to implement this property lays in RTTIInit, that allows transparent type information initialization in objects without interfering with the C++ constructors.

5.2 RTTIInit

The ability to type check an object at runtime depends on the presence of type information. Typically, approaches using external metadata to track object types

```
 1 using namespace std;
 2 union Union {
 3     int i;
 4     char c[2];
 5 };
 6 struct X { Union u;};
 7 union BiggerUnion {
 8     X x;
 9     OtherClass c;
10 };
```

```
12 int main() {
13     BiggerUnion bu; // No member active
14     bu.x.u.i = 65; // bu.x and bu.x.u.i active
15     // u should only be accessed as integer.
16     cout << bu.x.u.c[0] <<endl; // UB
17     bu.x.u.c[0] = 0x42; // u is active as char[]
18     bu.x.u.c[1] = 0x41;
19     bu.c = OtherClass(); // Direct assignment
20     return 0;
21 }
```

Listing 3: Abbreviated snippet showing valid and invalid union member accesses. The assignment at line 14 activates `bu.x` and `u.i`. Access to `u` through a non-active member leads to undefined behavior (*e.g.,* line 16) [5]. Sourcerer's RTTI-Init is called at each access while type++ misses instrumenting line 17 as calling the constructor would overwrite `bu.x` and `u.c`.

struggle to follow all object lifetime events (*e.g.,* copy). Inline metadata, as pioneered by LLVM-CFI and type++, is more robust as the type information is stored in the object and not disjoint from the object such as for TypeSan [10] and HexType [14]. The type information will be carried in the different lifetime events. Therefore, only the object creation requires careful handling to ensure the type information is correctly initialized. Typically in C++, object creation happens through `new` or direct assignment, which both call the object constructor which also sets RTTI when required. The type++ implementation followed this approach, by forcing constructor calls for object creation not relying on any initialization but only allocation, *e.g.,* `malloc`. This approach breaks different idioms in the C++ standard like explicitly `deleted` constructors or `const` objects where initialization should occur only once. To avoid these issues, Sourcerer introduces a new initializer, *RTTIInit*, uniquely focused on setting RTTI as exemplified in Listing 2. By interacting only with the RTTI field, Sourcerer avoids incompatibilities with `const` qualifiers and minimizes the performance cost of setting the RTTI. Additionally, RTTIInit does not interfere with the remaining object content, allowing for complete support of unions as detailed in §5.3. Lastly, our initializer, as a new language feature, does not conflict with existing constructors or the lack thereof which was a limitation of type++ that complicated its deployment.

5.3 Support for Unions

Unions, in C++, use the same memory to store objects of different types, allowing, however, only a single type to be active at a time. The union switches type when a member is activated, either through a direct assignment (Line 19 in Listing 3) or by setting a field of a union member. As the core property of Sourcerer is to maintain inline type information throughout the program execution, Sourcerer needs to ensure that, upon activation, the type information

```
1 template <class _Tp>
2 struct __list_node : public __list_node_base<_Tp> {
3   // Starting the lifetime of nodes without initializing in order to be
↪   allocator-aware.
4 private:
5   _ALIGNAS_TYPE(_Tp) char buf[sizeof(_Tp)];
6 
7 public:
8   _Tp& __get_value() {
9     return *__launder(reinterpret_cast<_Tp*>(&buf));
10  }
```

Listing 4: Challenging idioms in the libc++ 19.0.0 `list` implementation. A node is allocated through a `char` array later cast to the desired type. The constructor is later called via `construct_at`. type++ calls the constructor at line 5 thereby unfortunately disabling the constructor homing optimization [17]. Sourcerer, on the other hand, can either allow-list the cast at line 9 or call the RTTI initializer at line 5 without breaking the optimization.

is correctly updated in memory. Direct assignments do not require further handling as the incoming object's RTTI is already set. As a field assignment can occur when the object is already activated, calling a constructor would overwrite the stored object content (*e.g.,* Line 17 in Listing 3). Sourcerer, however, calls RTTIInit at each field assignment, which sets the RTTI without modifying the remaining object content.

5.4 Dialect Simplification

Sourcerer relaxes two dialect requirements introduced by type++. First, the type++ compiler requires a default constructor to be defined for each instrumented class and has to relax the `deleted` attribute in case the default constructor is defined as such. Sourcerer's instrumentation, on the other hand, does not use constructors and retains the intention of the developer. Sourcerer, additionally, remove type++ changes for `const` variables initialization. Relying on a constructor to set RTTI imposes a second initialization step, breaking the single initialization requirement of `const` variables. Conversely, as RTTIInit is not counted as an actual initialization step, it averts any limitation for `const`.

5.5 Unsupported Idioms in Earlier Work

While deploying Sourcerer, we encountered a C++ idiom that type++ could not support. Specifically, libc++, the LLVM C++ standard library, explicitly avoids calling the object constructor when allocating a tree node to allow for constructor homing [17], an optimization reducing the amount of emitted debug information. Instead, they allocate a `char` array and then cast it to the desired type as shown in Listing 4. type++ either breaks the optimization or cannot set the RTTI, disregarding type safety. Sourcerer, on the other hand, can instrument this peculiar allocation pattern without breaking the optimization.

When evaluating EffectiveSan, we identified two unsupported idioms. First, Custom Memory Allocators (CMA) need to be replaced, incurring many LoC modifications. In comparison, Sourcerer only requires a list of CMAs and handles their instrumentation automatically. Secondly, similarly to the type++ authors, we encountered a false positive due to incomplete handling of C++ templates, an issue not faced by Sourcerer.

6 Implementation

Sourcerer needs to add inline metadata to the necessary classes and instrumentation to verify for all casts. We implement the Sourcerer prototype on top of the modular compiler toolchain, LLVM 19.0.0 [16]. Specifically, we port the class collection, custom allocator logic, as well as the warning analysis from type++ to LLVM 19.0.0. For RTTIInit, we copy the logic of the default constructor, but trim it down to set only type information *i.e.,* the vtable and the RTTI. The resulting ABI change is similar to the one in type++—any function interacting directly with the object size might be problematic. For verification, we rely on LLVM optimized type checks. Moreover, compared to type++, we add support for multiple allocator edge cases such as zero-size allocation and frees through `realloc`. Overall, our implementation totals 9K LoC and consists of two compilation passes, first to gather the types and then to instrument them. We open-source Sourcerer and the evaluation at github.com/HexHive/Sourcerer.

7 Evaluation

Sourcerer's evaluation targets the following research questions:

– **RQ1:** What extra efforts are required to check all unrelated casts?
– **RQ2:** What is Sourcerer runtime overhead compared to the state-of-the-art?
– **RQ3:** Which source causes the performance overhead introduced by Sourcerer?
– **RQ4:** How effective is Sourcerer at detecting type confusion vulnerabilities?
– **RQ5:** How does Sourcerer perform in a real-world bug-hunting scenario?

Experimental setup. Our evaluation runs in Ubuntu 20.04 Docker containers on a server with two Xeon E5-2680v4 @ 2.4GHz and 256GB of RAM.

Evaluation targets. Sourcerer's evaluation is twofold: first, we compare it to the state-of-the-art, then we demonstrate Sourcerer effectiveness on current large-scale projects. As such, we evaluate Sourcerer on the SPEC CPU2006 [11] and SPEC CPU2017 [3] benchmarks as they are the common benchmarks across the state-of-the-art. For both, we select all the C++ programs and compile them with `-O2` optimization level. Overall, we evaluate Sourcerer on 2,040K LoC lines of code across the 16 programs of the SPEC CPU suites. Additionally, we conduct a fuzzing campaign against OpenCV [2] (commit `796adf`), the state-of-the-art computer vision library, showcasing the ability to use Sourcerer to find illegal unrelated casts. We choose OpenCV as it is a large, 1.3M LoC, and popular C++ project with a fuzzing setup readily available from OSS-Fuzz [25].

State-of-the-art competitors. We compare Sourcerer against a representative set of the state-of-the-art type confusion sanitizers. Specifically, we choose type++ [1] as it is the most recent type confusion protection and the cast checker of LLVM-CFI [28] due to its use in industry. Finally, we report numbers from EffectiveSan [7] as it is the only other tool claiming to protect unrelated and derived casts. Despite our efforts, we were unable to run EffectiveSan as the cast checking configuration is neither present in the source code nor in the documentation. Lastly, EffectiveSan checks types at pointer access, and, therefore, does not report the number of cast operations protected. This discrepancy makes a quantitative security comparison of Sourcerer and EffectiveSan meaningless.

For each tool, we report the performance overhead, averaged across five runs, and compare it to the corresponding LLVM vanilla version—13.0.1 for type++, and 19.0.0 for LLVM-CFI and Sourcerer. To assess the effectiveness of Sourcerer, we report the number of runtime cast operations checked, similarly to type++. Every configuration is run with Link-Time Optimization (LTO) enabled as required by LLVM-CFI cast checking.

7.1 Porting Effort

Sourcerer follows the type++ dialect specification, and, therefore, requires similar porting efforts to translate C++ code into its dialect. Starting from type++'s open-source patches, we address the additional warnings caused by the classes involved in unrelated casts. As a first metric, we report the number of extra classes that need to be instrumented to check unrelated casts. For these extra classes, we break down the kind of unrelated cast causing their instrumentation in the column *Unrelated* in Table 1. On average, Sourcerer instruments and monitors 4.95× more classes than type++ as our checks stretch beyond derived casts. Some programs like POV-Ray have few classes, *e.g.,* 12, involved in derived casts but extensively use unrelated casts with 161 classes cast. SoPlex experiences a less dramatic increase, *e.g.,* 3× more classes, with most classes cast as part of libc++ data structures headers (*e.g.,* vector). This reduces the overall effort as porting the library is amortized across the different programs. In Table 1, we report the classes requiring explicit instrumentation. The total number of types instrumented is a superset as some classes inherit the instrumentation and templates are counted once and not per specialization.

The actual porting effort caused by the new classes is small. For the SPEC-CPU benchmarks, we only modify 120 LoC on top of the type++ patches, for a total of 453 LoC patched. The new changes are relatively minor, for example, an initialization procedure (*e.g.,* `placement_new`) in NAMD 2017. Around 20% of the LoC changed are fixes for type confusions. They are necessary because the ABI changes do not always allow Sourcerer to recover from the subsequent memory corruption. For example, in Povray 2017, a parent object is created by `malloc`-ing enough memory but storing the returned pointer in a variable of a

Table 1: Breakdown of the number of classes instrumented by Sourcerer as well as the number of RTTI initialization. *Total* shows the number of classes that Sourcerer instruments which is broken down into the cause of the instrumentation. *Derived* indicates classes involved in derived cast and, therefore, already instrumented by type++. Then, we report the additional classes involved in unrelated casts either from a specific type (*i.e., class*) or from generic pointers (*e.g.,* `void*`). The last two columns indicate the number of instrumented objects and the percentage which are initialized through RTTIInit.

	Program	LoC	Instrumented classes				# RTTI Init.	% Through RTTIInit
			Total	Derived	Unrelated			
					class	`void*`		
SPEC CPU2006	NAMD	4K	27	9	4	14	460K	100.0
	deal.II	95K	63	12	4	47	15,875M	98.32
	SoPlex	28K	39	13	6	20	726M	5.91
	POV-Ray	79K	161	12	23	126	6,243M	99.97
	OMNeT++	27K	53	13	4	36	2M	65.82
	Astar	4K	31	9	4	18	6,024M	91.97
	Xalan-C++	264K	149	46	5	98	1,407M	99.86
SPEC CPU2017	cactuBSSN	63K	31	11	5	15	334K	81.19
	NAMD	6K	26	9	4	13	0	-
	Parest	359K	120	26	4	90	20,384M	98.53
	POV-Ray	80K	161	12	23	126	25,179M	100.0
	Blender	616K	57	12	20	25	70M	74.3
	OMNeT++	86K	125	14	5	106	3M	38.93
	Xalan-C++	291K	203	46	7	150	35,276M	99.61
	Deep Sjeng	7K	27	9	4	14	15M	0.0
	Leela	31K	34	11	4	19	4,491M	99.53
	Total	2,040K	1307	264	126	917	-	-

larger child type as exemplified in Listing 5. As Sourcerer identifies the returned memory as a child object, it calls RTTIInit which set information out of the allocated bounds, leading to memory corruption. The biggest changes were in Blender, which interacts heavily with C code resulting in two challenges. First, as some structs are defined in headers included in both C and C++ code, we had to mimic the presence of RTTI information in the C code to ensure a compatible ABI. The second issue arises when, in C code, an instrumented C++ object is cast to a pure C struct. As Sourcerer only instruments C++ code, the cast is not checked nor does the destination type expect the RTTI field. We modified the C struct to be aware of the presence of type information. Moreover, instrumenting more classes showed some unexpected benefits as we could remove a patch from type++ for SoPlex as layout similarity is restored between two types involved in an unrelated type confusion.

Table 2: Performance and security evaluation of Sourcerer compared to the state-of-the-art. Under "*%*" we report the performance overhead compared to the tool's baseline. Then, in the *Casts* column, we report how many unrelated casts were checked at runtime. The two *Δ* columns show the extra operations verified by Sourcerer on top of the competitors. The last two columns report the average and peak memory overhead of Sourcerer in terms of the working set size.

	Program	LLVM-CFI		type++		Sourcerer		*Δ* Casts		Memory	
		%	Casts	%	Casts	%	Casts	LLVM-CFI	type++	Avg.	Max.
SPEC CPU2006	NAMD	0.41	1	0.27	0	0.53	1	0	1	0.45	0.45
	deal.II	-1.15	649K	1.95	122M	4.33	128M	127M	5M	-2.17	1.95
	SoPlex	-0.01	206K	0.22	27M	12.64	30M	30M	4M	66.79	66.83
	POV-Ray	0.92	1M	1.60	1,342M	7.93	1,345M	1,344M	4M	10.39	10.39
	OMNeT++	3.42	3	0.67	270K	3.04	284M	284M	283M	1.43	1.43
	Astar	0.85	0	0.90	0	16.99	4M	4M	4M	104.89	82.03
	Xalan-C++	0.63	5K	-0.54	5K	-1.29	161K	156K	156K	9.28	2.85
SPEC CPU2017	cactuBSSN	0.47	0	-0.71	100	-0.21	21K	21K	21K	0.30	0.08
	NAMD	0.52	5	-0.68	0	0.71	0	-5	0	0.23	0.22
	Parest	0.36	24K	0.07	85M	2.46	106M	106M	21M	2.11	1.99
	POV-Ray	0.08	39K	1.73	5,370M	9.59	5,381M	5,381M	11M	10.98	10.96
	Blender	0.69	0	2.92	7M	9.62	7,643M	7,643M	7,636M	2.78	3.01
	OMNeT++	1.31	6M	0.34	6M	4.11	501M	495M	495M	2.21	2.12
	Xalan-C++	-0.10	4K	-0.24	198M	15.38	243M	243M	45M	4.94	4.43
	Deep Sjeng	-0.14	0	-1.09	0	0.45	1	1	1	16.37	16.37
	Leela	-0.53	0	-0.27	1K	-0.19	416K	416K	414K	15.69	5.69
	Avg./Total	0.26	8M	0.43	7,158M	5.14	15,666M	15,658M	8,507M	-	-

EffectiveSan preserves the C++ ABI but struggles with custom allocators. In their artifact, the changes necessary for SPEC CPU2006 totaled a non-trivial 297 LoC. In contrast, Sourcerer needs to modify 453 LoC while incurring, in the worst case, only a third of EffectiveSan's average runtime overhead.

Overall, we conclude that the porting efforts for Sourcerer are reasonable and in line with the efforts necessary for similar tools.

7.2 Performance Overhead

Since speed is key for automatic testing, sanitizers should incur a limited performance overhead (§4). Below, we quantify the performance cost of deploying Sourcerer on the SPEC CPU benchmarks and compare it to the state-of-the-art.

Table 2 details the performance overhead of the different tools in the columns marked "%". Each value is the average performance overhead compared to a binary compiled with vanilla Clang—version 13.0.1 for type++, 19.0.0 for both LLVM-CFI and Sourcerer. We observe a negligible standard deviation across the five runs. As expected, the additional classes instrumented increase the overhead of Sourcerer compared to LLVM-CFI and type++. While LLVM-CFI and type++ are mitigations, Sourcerer's higher overhead is outstanding for a sanitizer deployed in a testing environment. For example, UBSan [4] manifests slowdowns of up to ∼1.7x. More precisely, Sourcerer incurs a 5.14% performance penalty but covers all casts in a program. When looking only at SPEC CPU2006, the target set of EffectiveSan, Sourcerer overhead is limited to 6.47% compared to the 49% overhead reported by EffectiveSan's authors. Consequently, Sourcerer reduces the runtime overhead for a complete sanitizer by a factor of seven. In fact, Sourcerer shows a similar overhead to HexType [14] but additionally checks unrelated casts and avoids false positives by design.

Looking at individual programs, we observe that the overhead is not uniform. As shown in Table 2, programs such as cactuBSSN, NAMD and Deep Sjeng experience virtually no overhead but also check the fewest casts. On the other hand, SoPlex and Xalan-C++ 2017 suffer from an overhead of around 15% due, in part, to caching being undermined by bigger objects on hot paths. To conclude, a complete type confusion sanitizer is practical if the checks occurs at cast time and not at the frequent dereference sites, as implemented in EffectiveSan.

7.3 Source of the Performance Overhead

In this section, we study the performance cost of Sourcerer instrumentation. In particular, we conduct an ablation study on the three following elements: the type checks, the RTTI initializer, and the ABI changes. First, we disable the type checks but leave the instrumentation intact. Then, we also remove the call to RTTIInit, leaving only the changes to the ABI in place. Each experiment is compared to the same vanilla Clang baseline. We present the results in Table 3.

Comparing the columns *Full* and *W/o type checks* shows that the verification cost can be important, *e.g.,* Blender. Upon closer inspection, Blender exhibits a high ratio of failing type checks which is a slow path. Indeed, in a testing environment, we expect the program to be terminated upon encountering a type confusion while in this evaluation we continue to assess the program performance. Fixing these type confusions would reduce the cost of the checks to a similar level as in Xalan-C++ 2006 which proceeds to more, but successful, type checks. Overall, the numerous optimizations implemented in LLVM type checks [20] allow for this limited performance cost.

Inspecting the column *ABI change only*, we observe that altering the object size negatively affects the caching behavior. Indeed, in Astar, we observe that the class `pointt` is instrumented by Sourcerer. As a single byte object, adding RTTI double its size, negatively impacting caching in the tight loops of the `flexarray::add` function. Reducing this overhead could be achieved by removing all casts of `pointt`, and, therefore, Sourcerer instrumentation. This would return Astar to the original caching performance and a minimal overhead. A similar issue is observed in SoPlex. The cost of the ABI change varies a lot across programs and use-cases. Compared to type++, this effect is magnified in Sourcerer by the additional classes instrumented.

Lastly, Table 3 allows us to estimate the overhead of Sourcerer's RTTIInit as it is the only change between the columns *W/o type checks* and *ABI change only.* Disabling RTTIInit does not lead to a significant change in performance, highlighting the effectiveness of Sourcerer's RTTI initialization design.

```
 1 class X {}; // Instrumented
 2
 3 class A {
 4   int x; // A is instrumented
 5 };
 6
 7 class B : public A {
 8   int y;
 9   X x; // Need to set x RTTI
10 };
11
12 int main() {
13   A* a;
14   a=(B*)(malloc(sizeof(A)));
15   free(obj);
16   return 0;
17 }
```

Listing 5: Simplified excerpt of a type confusion in POV-Ray 2017. Sourcerer calls RTTIInit at line 14 right after the call to `malloc`. Sourcerer assumes, from the cast, that the underlying object is of type `B` while the allocated size is only sufficient for an `A` object. The call to `A`'s RTTIInit will try to set `x`'s RTTI information (line 9), resulting in an out-of-bounds write.

Table 3: Ablation study of Sourcerer performance overhead. The column `Full` lists the overhead of Sourcerer compared to the baseline. The third column shows the overhead once the type checks are disabled. For the last column, we additionally disable calls to RTTIInit, highlighting the cost of the ABI change. Comparing `Full` and `W/o type checks` shows the overhead of the cast validation. Finally, comparing the last two columns allows for an estimation of the overhead induced by RTTIInit.

	Program	Full	W/o type check	ABI change only
SPEC CPU2006	NAMD	0.09	0.15	-0.15
	deal.II	3.89	5.36	3.53
	SoPlex	12.88	12.87	8.80
	POV-Ray	8.58	5.20	5.41
	OMNeT++	6.28	1.37	0.05
	Astar	16.51	16.86	15.53
	Xalan-C++	-1.83	1.42	0.48
SPEC CPU2017	cactuBSSN	0.18	-1.01	-0.30
	NAMD	-0.15	-0.06	0.75
	Parest	1.77	2.32	2.07
	POV-Ray	9.73	5.68	5.95
	Blender	9.16	1.00	1.26
	OMNeT++	5.96	2.29	2.90
	Xalan-C++	14.64	13.97	15.34
	Deep Sjeng	0.15	0.06	0.16
	Leela	-0.50	-0.82	-0.51

Overall, this study highlights the cost of the ABI change, which is strongly dependent on the program and its use-cases. Nonetheless, the average overhead of Sourcerer is lower than other sanitizers while detecting all type safety violations.

7.4 Security Effectiveness

Sourcerer is a sanitizer deigned to detect *all* type confusions. In this section, we compare the number of cast checks against similar tools and describe the type confusions that Sourcerer identified which were missed by previous works. From a theoretical point of view, both EffectiveSan and Sourcerer provide complete coverage of all cast operations. We do not provide statistics about EffectiveSan checks as they do not happen at cast time but at every object dereference. However, similarly to the type++ authors, we observed false positives in EffectiveSan due to incompatibilities with templates.

Table 2 reports the number of casts checked by LLVM-CFI, type++, and Sourcerer. For each program, we list the number of unrelated casts verified during the benchmark execution. The difference between Sourcerer and both HexType and LLVM-CFI is listed in the two columns headed by *Δ Casts*, respectively. LLVM-CFI checks a mere 8M unrelated casts, less than 1% of all unrelated cast operations, while type++ already verifies slightly less than 50% due to classes being involved in both derived and unrelated casts. Sourcerer covers the remaining 50% of casts, reaching all 15.7B unrelated casts, highlighting the wide attack surface left unchecked by previous research. The increase in verified casts is dominated by Blender, but almost all programs benefit from the additional type confusion detection capabilities offered by Sourcerer. NAMD 2017 is a special case where type++ and Sourcerer report fewer casts due the changes necessary for the dialect which removed the only cast triggered in the execution.

In regards with the security impact, Sourcerer discovered 152 type confusions in the SPEC CPU benchmarks—30 more than the state-of-the-art. Most of these type confusions expect classes to have identical layouts. Despite the lack of guarantee from the C++ standard, C++ compilers rarely optimize object layouts thus, reducing the risk associated with these type confusions. Nonetheless, relying on such undefined behaviors is unsafe despite its widespread use.

7.5 Sourcerer as a Sanitizer for Fuzzing Campaigns

In this section, we showcase the effectiveness of Sourcerer as a sanitizer by using our prototype in combination with AFL++ [9]. We conduct the first fuzzing campaigns targeting specifically type confusions and compare its performance with a state-of-the-art memory sanitizer, AddressSanitizer (ASan) [24].

As target, we select OpenCV [2], the leading computer vision library, due to its ubiquity and the availability of fuzzing drivers as part of the OSS-Fuzz project [25]. We unleash AFL++, a state-of-the-art fuzzer, on the seven drivers and conduct five 24-hour fuzzing campaigns for both ASan and Sourcerer. We replay the inputs on a binary instrumented with SanCov [19] to have collision-free coverage and avoid different numbers of edges due to the instrumentation.

Thanks to Sourcerer's increased compatibility with the C++ standard, few changes are necessary to support the 1.34M C++ LoC of OpenCV. A peculiar issue is how the characteristics of matrices elements (*e.g.,* depth, size, and number of channels) are stored and later used to compute the offset between elements, shunning `offsetof`. As Sourcerer modifies this offset, matrices traversal results in RTTI being interpreted as matrix elements. Indeed, the type++ dialect is incompatible with hard-coded object sizes. More importantly, this is also less portable and needs support through `#ifdef` and header files. We leave out this code modernization as it involves modifications of the core components of the library. Instead, we use Sourcerer flexibility to disable the instrumentation of the five matrix element types—out of 668 classes involved in casts—trading a slight reduction of the findable type confusions for easier deployment of Sourcerer.

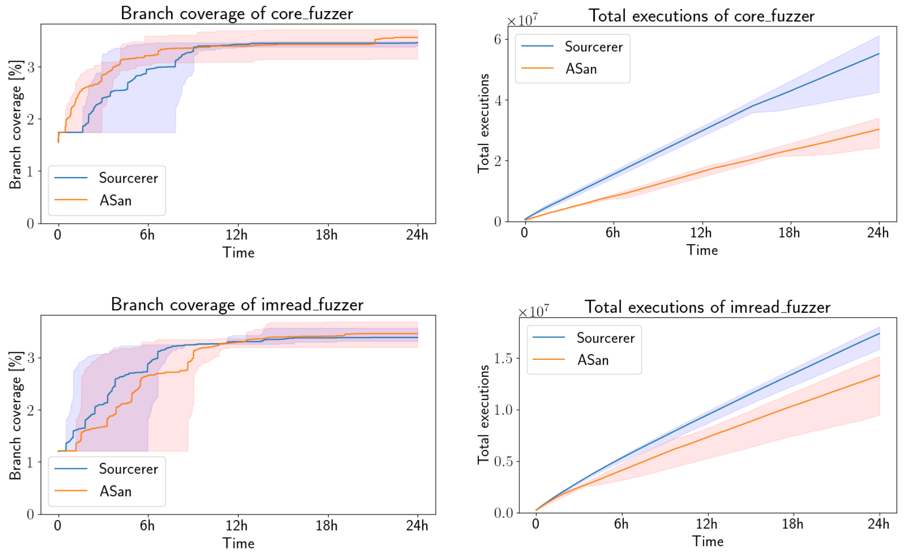

Fig. 1: Fuzzing campaign results. The left figures show the branch coverage throughout the 24h fuzzing campaign. On the right, Sourcerer allows for more executions due to reduced overhead compared to ASan.

Figure 1 shows the performance of the three fuzz drivers. In shaded colors are the minimum and maximum values achieved across the five repetitions. The left graphs highlight that Sourcerer achieves a similar branch coverage to ASan, despite being hindered by type confusion crashes. On the right, the total number of executions of the fuzz drivers shows that Sourcerer instrumentation is more lightweight than ASan, allowing to test more inputs.

In terms of crashes, the two drivers, `imread_fuzzer` and `core_fuzzer`, highlighted in Figure 1, triggered three type confusion bugs. The crashes are caused by similar unrelated casts to a `PaletteEntry` object at three different code locations. Our testing found three additional type confusions in code assuming indistinguishability between an array of `type`, *e.g.,* `float`, and a `Vec<type>` objects, representing a vector. However, as the array is oblivious to the `Vec` RTTI field, the `reinterpret_cast` results in shifted values and incorrect RTTI. Sourcerer's instrumentation makes this type confusion apparent but blocks the program execution as the object is corrupted, hindering fuzzing progress. To highlight the capabilities of Sourcerer, we investigate if type++ can identify the errors. Since type++ does not instrument `PaletteEntry` and `Vec`, type++ was unable to detect these errors, further showcasing the effectiveness of Sourcerer.

Previous type confusion sanitizers never conducted fuzzing campaigns likely due to false positives (*e.g.,* HexType) or the expensive porting effort (*e.g.,* CMAs in EffectiveSan). Sourcerer, therefore, is the first to demonstrate the feasibility and effectiveness of fuzzing campaigns with a complete type confusion sanitizer.

8 Related Works

In the next paragraphs, we discuss relevant works for type confusion sanitizers.
Type confusion defenses. TypeSan [10] and HexType [14] check derived cast by tracking object types throughout their lifetime in an external data structure. The complexity of C++ lifetimes leads to prohibitive cost and a high rate of false positives [23]. EffectiveSan [7] encodes, through fat pointers, the type and bounds of an object. At each pointer dereference, they perform a bound check and a type check causing a high runtime cost. Multiple dialects exist for C++ to prevent by design certain classes of vulnerabilities. Ironclad C++ [6] banned unions and added type information to every object requiring large changes to the source code. More recently, type++ [1] proposed limited code changes to add RTTI to each object involved in a derived cast. Finally, Uncontained [15] identifies derived type confusions in C containers, particularly in the Linux kernel.
Type check pruning. To avoid type confusions, developers implement their own type identifiers and checks. Recent works investigated disabling such checks when a sanitizer is deployed. In particular, Zhai & al. [29] automatically remove type checks redundant with HexType to improve performance. Orthogonally, HTADE [8] removes derived casts that are never dereferenced.

9 Discussion

In the following, we detail Sourcerer's limitations and possible extensions.
Custom allocator identification. Similarly to previous tools, Sourcerer tracks object lifetime and, therefore, their allocation. While `new` and direct initialization sets RTTI automatically, C style allocations (*e.g.,* `malloc`, `calloc`, and `realloc`) require to explicitly initialize the RTTI through RTTIInit. Additionally, *Custom Memory Allocators* (CMA) wrap the standard allocation functions to provide extra features like memory pool or allocation metadata (*e.g.,* ASan). To initialize RTTI, Sourcerer as other sanitizers, must be aware of these allocators and is, therefore, configurable through an allow-list. Orthogonally, CMAsan [12] recently automated CMAs identification in C++ projects.
Identification of Allocation Type. To correctly set type information after an explicit allocation, Sourcerer needs to know the allocated type. Assuming the presence of a cast to the desired type right after returning from the allocation has proven sufficient in our evaluation. A sounder static analysis would be able to follow allocation until their first actual assignment.
Reliance on Link-time optimization. Sourcerer relies on the type checks implemented by LLVM-CFI which leverages the LLVM `type.test` function [20]. To allow optimized checks, it leverages Link-Time Optimization (LTO) to optimize inheritance hierarchies. During libc++ instrumentation, we discovered that some casts were unchecked by LLVM-CFI as their LTO-visibility attribute is set to expose symbols to external compilation units, preventing LLVM from emitting type checks. Clang provides an option, `-fsanitize-cfi-cross-dso`, to check externally visible objects across Dynamic Shared Objects (DSO), at the

cost of lower performance and extra compilation requirements (*e.g.,* position-independent code) [18]. We leave evaluating this option as future work.

Future Work. Accessing a union through a non-active member is undefined in C++ [13]. Practically, the illegal access is identical to a `reinterpret_cast`. To solve this issue, Ironclad C++ [6] banned unions in their dialect. Adding type information to the types used in unions would allow Sourcerer to check the validity of union access at the cost of additional porting effort and performance overhead. We leave verifying union access correctness as future work.

Sourcerer is a sanitizer and, therefore, is not suited for an adversarial scenario. Multiple improvements are necessary to mitigate type confusions. First, all source objects need to be typed, as otherwise, an attacker controlling the object content could forge RTTI values, resulting in false negatives. As casts to integral types are widespread (*e.g.,* a pointer passed as a `void*` argument), it would result in more instrumentation. Static analysis might reduce the number of classes to instrument by inferring if a `void` pointer is ever cast in its scope. Adoption would also require Sourcerer's performance overhead to be reduced through, for example, type check pruning [29] or type check removal [8] (§8). Code modernization, *e.g.,* removing casts on hot paths or the source of instrumentation of classes heavily cached, has the highest improvement potential §7.3.

10 Conclusion

We introduce Sourcerer, a novel type confusion sanitizer that checks *all* casts at runtime. By combining inlined type information with our optimized RTTI initializer *RTTIInit*, Sourcerer checks all casts explicitly while reducing the divergence of the type++ dialect with the C++ standard. Additionally, RTTIInit supports unions and other idioms which were missing from existing tools.

We evaluate Sourcerer on the SPEC CPU benchmarks. Our tool checks twice as many unrelated casts compared to type++. On average, Sourcerer incurs 5.14% overhead, six times lower than the other complete type confusion sanitizer, EffectiveSan. Our ablation study identifies the cause of the overhead to be the required ABI changes. Lastly, during our fuzzing case study targeting specifically type confusions, we identify six new type confusion bugs and many code locations that would benefit from code modernization efforts.

Acknowledgments

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, and the German Research Foundation (DFG) under Germany's Excellence Strategy – EXC 2092 CASA – 390781972.

References

1. Badoux, N., Toffalini, F., Jeon, Y., Payer, M.: type++: Prohibiting Type Confusion With Inline Type Information **34**(4), 1–17 (2025)
2. Bradski, G., et al.: Opencv. Dr. Dobb's journal of software tools **3**(2) (2000)
3. Bucek, J., Lange, K.D., v. Kistowski, J.: SPEC CPU2017: Next-generation compute benchmark. In: Companion of the 2018 ACM/SPEC International Conference on Performance Engineering. pp. 41–42 (2018)
4. Clang: UBSan. `clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`
5. cppreference.com: Union. `en.cppreference.com/w/cpp/language/union`
6. DeLozier, C., Eisenberg, R., Nagarakatte, S., Osera, P.M., Martin, M.M., Zdancewic, S.: Ironclad C++ a library-augmented type-safe subset of C++. ACM SIGPLAN Notices (2013)
7. Duck, G.J., Yap, R.H.: 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)
8. Fan, X., Long, S., Huang, C., Yang, C., Li, F.: Accelerating Type Confusion Detection by Identifying Harmless Type Castings. In: Proceedings of the 20th ACM International Conference on Computing Frontiers. p. 91–100 (2023)
9. Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: AFL++ : Combining incremental steps of fuzzing research. In: 14th USENIX Workshop on Offensive Technologies (WOOT 20). USENIX Association (Aug 2020)
10. Haller, I., Jeon, Y., Peng, H., Payer, M., Giuffrida, C., Bos, H., van der Kouwe, E.: TypeSan: Practical type confusion detection. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 517–528
11. Henning, J.L.: SPEC CPU2006 benchmark descriptions. ACM SIGARCH Computer Architecture News (2006)
12. Hong, J., Jang, W., Kim, M., Yu, L., Kwon, Y., Jeon, Y.: CMASan: Custom Memory Allocator-aware Address Sanitizer. In: 2025 IEEE Symposium on Security and Privacy (SP). pp. 74–74. IEEE Computer Society (2024)
13. ISO C++ Standards Committee and others: Standard for Programming Language C++. Working Draft N4950. Tech. rep., Tech. rep. ISO IEC JTC1/SC22 (2023)
14. Jeon, Y., Biswas, P., Carr, S., Lee, B., Payer, M.: HexType: Efficient Detection of Type Confusion Errors for C++. In: CCS (2017)
15. Koschel, J., Borrello, P., D'Elia, D.C., Bos, H., Giuffrida, C.: Uncontained: Uncovering Container Confusion in the Linux Kernel. In: 32nd USENIX Security Symposium. USENIX Association (Aug 2023)
16. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: International symposium on code generation and optimization, 2004. CGO 2004. pp. 75–86. IEEE (2004)
17. LLVM: Constructor type homing. `blog.llvm.org/posts/2021-04-05-constructor-homing-for-debug-info/`
18. LLVM: LLVM Control Flow Integrity: Shared library support. `clang.llvm.org/docs/ControlFlowIntegrity.html#cfi-cross-dso`
19. LLVM: SanitizerCoverage. `clang.llvm.org/docs/SanitizerCoverage.html`
20. LLVM: Type Metadata. `llvm.org/docs/TypeMetadata.html`
21. Matsakis, N.D., Klock, F.S.: The Rust language. ACM SIG Ada Letters (2014)
22. Mitre Corporation: Common Weakness Enumeration (CWE) 704: Incorrect Type Conversion or Cast. `cwe.mitre.org/data/definitions/704.html`

23. Payer, M.: Type Confusion: Discovery, Abuse, Protection. `hexhive.epfl.ch/publications/files/18SyScan360-presentation.pdf` (2017)
24. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: AddressSanitizer: A fast address sanity checker. In: 2012 USENIX Annual Technical Conference (2012)
25. Serebryany, K.: OSS-Fuzz - google's continuous fuzzing service for open source software. USENIX Association, Vancouver, BC (Aug 2017)
26. Stroustrup, B., Sutter, H., et al.: C++ Core Guidelines (2018)
27. Sutter, H., Stroustrup, B., other contributors: C++ Core Guidelines. `github.com/isocpp/CppCoreGuidelines/` (2015)
28. Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., Pike, G.: Enforcing Forward-Edge Control-Flow integrity in GCC & LLVM. In: 23rd USENIX Security Symposium (2014)
29. Zhai, Y., Qian, Z., Song, C., Sridharan, M., Jaeger, T., Yu, P., Krishnamurthy, S.V.: Don't Waste My Efforts: Pruning Redundant Sanitizer Checks by Developer-Implemented Type Checks. In: 33rd USENIX Security Symposium (2024)