TCD: Statically Detecting Type Confusion Errors in C++ Programs

Changwei Zou*,Yulei Sui†, Jingling Xue*,
*School of Computer Science and Engineering, UNSW, Sydney, Australia
†Faculty of Engineering and Information Technology, University of Technology Sydney, Australia

***Abstract*—The C++ programming language plays an important role in constructing software infrastructures. In large C++ projects, downcasting (converting a base-class pointer to a derived-class pointer) is error-prone, as programmers may not have a global vision about which objects a pointer might point to. If the actual type of the object pointed is not the derived class or its descendant, a type confusion vulnerability ensues, which attackers might exploit to hijack the control flow. Such attacks show an apparently increasing trend in recent years. Correspondingly, researchers have proposed several effective runtime bad-casting detection tools, which perform instrumentation at compile time and verify type casting dynamically. However, the inherent low coverage of these dynamic detectors means that each execution of these tools can only touch a small part of the whole program space.**

This paper presents a static Type Confusion Detector (TCD), which uses pointer analysis to locate potential type confusion errors, including those that are hard to reach for existing dynamic tools. Although pointer analysis has been studied for decades, type information of abstract objects is usually not put into consideration. By considering compiler instrumentation and pointer analysis together, we solve the challenges in assigning type information to abstract objects when handling C++ characteristics, such as placement new and multiple inheritance. Our experiment demonstrates that TCD is able to expose six type confusion errors in libstdc++ and QT. Moreover, on average, it finds that 14.22% of the casts are safe in one group of our test cases. This figure could even soar to 71.78% in a group of machine learning applications. In other words, TCD can also be used as a static optimizer to further reduce the runtime overhead of existing dynamic solutions.

***Index Terms*—Type Confusion; Downcasting; Pointer Analysis; Static Program Analysis;**

I. Introduction

Large software projects such as Linux kernel, compilers, browsers, and Java Virtual Machine are the cornerstone for modern software industry. To seek for higher performance and low-level access to memory, almost all these fundamental products are implemented in C or C++, both of which lack memory safety, as well as type safety. As a result, these software products are often plagued by buffer overflow, type confusion, and use-after-free vulnerabilities, which impose a tremendous threat to our society, especially considering that human beings are nowadays increasingly relying on software. Although much effort has already been paid in the past decades to tackle this important issue, there is still a long way to go.

With the rapid increase in both complexity and scale of software, teamwork is necessary to any large project in this industry. Given a base-class pointer in a large project, it is harder than ever before for C++ programmers to figure out which objects this pointer might point to. If a base pointer is cast to a derived one, it is their duty to make sure that type casting is correct at runtime. If this fails, a type confusion error occurs, allowing attackers to corrupt out-of-bound data and to hijack control flow by tampering with code pointers.

As CVE reports [1] related to type confusion errors see an apparent upward trend, several effective solutions have been presented in recent years.

By replacing all `static_cast` expressions with `dynamic_cast` ones, Undefined Behavior Sanitizer [2] could utilize the default RunTime Type Identification (RTTI) mechanism in C++ to check type casting dynamically. However, non-polymorphic classes are not included in this protection, as they do not contain any virtual table. Furthermore, the overhead of RTTI is considerably high.

To solve these problems, CaVer [3] customizes its own heap allocator to align objects in the heap such that they can be mapped to their metadata at `O(1)` complexity. In this way, both polymorphic and non-polymorphic classes can be supported. As red-black trees are used to store the metadata for local and global objects, their solution still incurs high overhead when most of the allocated objects are on stack.

CaVer has been further improved by TypeSan [4]. TypeSan relies on a compact memory shadowing mechanism to trace all objects (including stack, global and heap objects) in a uniform way, such that the overhead of tracing stack objects is greatly reduced. One limitation of TypeSan is that it might conflict with Address Space Layout Randomization (ASLR) [5], which randomly arranges the base address of the code and data sections.

To achieve lower runtime overhead, HexType [6] employs a hash table as a fast path and a red-black tree as a slow path to trace objects. EffectiveSan [7] enforces not only type safety, but also spacial and temporal memory safety, at the cost of much higher runtime overhead. In order to further decrease the performance overhead spent on dynamic checking, Bitype [8] designs a Safe Encoding Scheme to verify type casting in an efficient `xor` operation. By reordering virtual tables, CastSan [9] employs virtual pointer range checking to reduce runtime overhead. However, it is mainly designed for polymorphic classes.

According to the development of the above detection tools,

it is clear that how to reduce their performance overhead is always one of the key challenges of these systems. To detect potential type confusion errors, these dynamic tools can be driven by a fuzzing framework like AFL [10]. Even so, it is still difficult for them to touch every possible path in the program space. By contrast, static program analysis is able to make an over-approximation and to provide an alternative solution in this scenario. The example in Figure 1 illustrates the two motivations for TCD introduced in this paper.

One motivation is to detect type confusion errors statically, including those that are hard to reach in current dynamic tools. The check in line 19 represents some hard-to-satisfy conditions for a fuzzer, like checking on magic values or executing a special path in 2^N candidates. If the control flow could go through line 20, `bptr` in line 22 will point to the `B` object in line 11 and a type confusion error will be triggered. Then the `delete` operation in line 23 will attempt to invoke virtual destructor of class `C`, leading to illegal access of the orange fields incorrectly. Worse still, if these out-of-bound fields are under control of some attackers, they can easily hijack control flow by corrupting the virtual table pointer.

To test how much time existing dynamic tools (driven by AFL) might need to reach the statement in line 20, we read an 8-byte integer value from standard input and compare it with a magic value, 0x12345678deadbeef, in function `hard_to_satisfy()`. If they are equal, the `hard_to_satisfy()` condition is satisfied. Even in so simple a program, line 20 has not been visited in a 24-hour fuzzing test, after 431 million executions. Although there are some researches like T-Fuzz [11] to mitigate this issue, symbolic execution and constraint solvers they rely on might get into trouble when solving extremely complex constraints.

By contrast, if we adopt path-insensitive pointer analysis (ignoring conditional guards), it will report that the pointer `bptr` in line 22 may point to two objects, the `B` object in line 11 and `C` object in line 17 respectively. In other words, there might be a bad cast in line 22. In spite of over-approximation, it provides just enough details for C++ programmers to detect the vulnerability.

Another motivation is to expose safe casts that are ignored by current dynamic detectors. Based on inter-procedural pointer analysis, we know that the pointer `bptr` in line 14 can only point to the `C` object in line 17. So the `static_cast` in line 14 is safe. It means that points-to information we obtain from pointer analysis could be used to remove redundant checks and to reduce runtime overhead of existing dynamic solutions.

In the literature, pointer analysis is usually discussed in the context of Java or C language. C++ is relatively overlooked. During the course of developing TCD, we have to solve some challenges caused by C++ characteristics.

Our major contributions are listed as follows:

- A type confusion detector based on SVF [12] and LLVM [13], which performs static analysis to provide C++ programmers with a global vision about potential bad casts. To the best of our knowledge, this is the first static solution to this problem.
- An enhanced location set mechanism which puts type information into consideration. We bridge the gap between traditional pointer analysis and type confusion detection by solving challenges imposed by C++ characteristics such as placement `new` and multiple inheritance.
- A novel optimization option for current dynamic tools to further reduce their runtime overhead, by removing unnecessary instrumentation and checks. TCD finds that on average 14.22% runtime checks are redundant in one group of test cases, and 71.78% in the other.

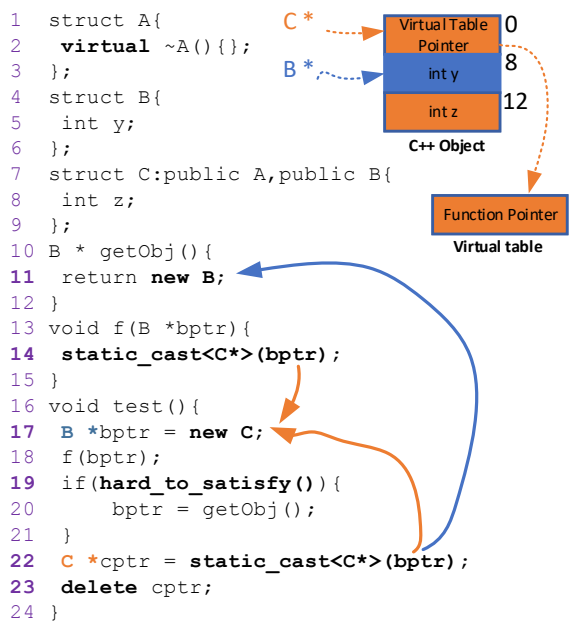

Fig. 1. A motivation example of TCD.

The rest of this paper is organized as follows. Section II explains type casting in C++, as well as basic knowledge of pointer analysis. Section III illustrates the scope of topic discussed in this paper. Section IV presents the design and implementation of our static bad-cast detector, focusing on concrete issues and our solution. Section V reports the result of our experiment, demonstrating that TCD is able to expose potential downcast errors and to reduce unnecessary runtime checks for existing dynamic detectors. Related work and conclusions are summarized as the last two sections, that is, Section VI and Section VII respectively.

II. Background

The reports about type confusion errors on CVE [1] website have surged rapidly since 2010 [8], not only in quantity, but also in its impact.

A. Type Casting in C++

C++ introduces four keywords to support different casts, namely, `static_cast`, `reinterpret_cast`, `dynamic_cast` and `const_cast`. Specifically speaking, `static_cast` is mainly used between parent class

and its descendants. Pointer value might be adjusted in case of multiple inheritance. Since type checking happens at compile time, it is called a static cast. On the contrary, `dynamic_cast` is a type conversion between polymorphic classes which should contain at least one virtual table. At compile time, C++ compiler inserts code to call `__dynamic_cast()`, a C++ library function, to enforce the semantics required. At runtime, function `__dynamic_cast()` will search type information stored in virtual tables to check whether the cast is correct. The default RTTI mechanism based on virtual table is not efficient enough in current C++ implementations. Hence, a number of C++ applications set up their own RTTI mechanism. These custom RTTI mechanisms could reduce runtime overhead, at the cost of requiring extra manual code modifications.

As the C++ version of C-style cast, `reinterpret_cast` will keep the pointer value same as the original one, even when it is applied in a multiple-inheritance situation. Furthermore, `reinterpret_cast` can be used between unrelated types while both `static_cast` and `dynamic_cast` are often conducted in the same class hierarchy. As for `const_cast`, its main purpose is to discard read-only limitation on an object. Since the memory layout of the object is respected, it is not in the scope of type confusion error we discuss in this paper.

Given a base-class pointer, if it is cast to a derived-class pointer and the actual object it points to at runtime is incompatible with the derived class, a type confusion error ensues. One of the major reasons for this error is that C++ programmers are not quite certain about which objects the pointer might point to. This situation will be worsen if communication in a project team does not go well.

B. Pointer Analysis

There is no shortage of indirection in software, such as data pointers, function pointers and C++ virtual tables. The aim of pointer analysis is to figure out which objects a pointer may or must point to. Approximation plays an important role in solving points-to issues. There are different kinds of precision in pointer analysis. Flow-sensitive analysis maintains the order of statements, meaning that control flow is respected. Context-sensitivity and path-sensitivity are about modeling calling stack and branch conditions respectively. Field-sensitivity is able to distinguish different sub-objects in a struct object.

Sui et al. implemented SVF [12], an open-source platform for pointer analysis of C and C++ programs. Taking as input the IR module generated by LLVM [13], SVF enables scalable and inter-procedural static value-flow analysis. Based on SVF, a number of applications have been implemented, such as a framework for analyzing Linux kernel [14], as well as a directed grey-box fuzzer [15].

However, during the course of developing TCD, we find that there is still some gap between SVF and the need of detecting C++ type confusion errors. More details will be discussed in Section IV.

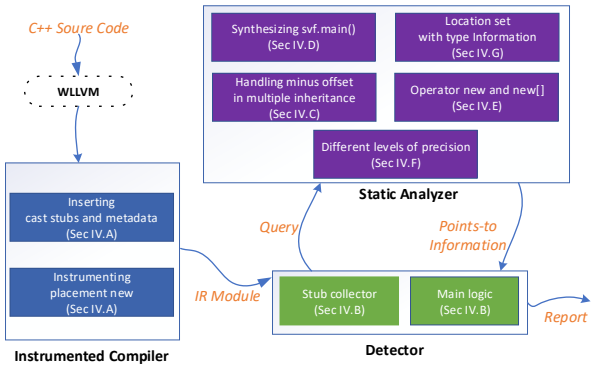

Fig. 2. An overview of TCD.

III. Threat Model

In this paper, we mainly focus on detecting type confusion errors in C++ programs. Other kinds of vulnerabilities like buffer overflow and use-after-free are beyond our discussion. Type confusion errors might be used to corrupt sensitive data, code pointers or virtual table pointers in an attack. How to exploit these vulnerabilities is also out of scope of this paper.

IV. TCD - Design and Implementation

An overview of TCD is described in Figure 2, which consists of three components: an instrumented LLVM compiler, a static analyzer based on SVF and a type detector. With regard to LLVM and SVF, we only highlight the work we add or change in these two frameworks. At the first stage, C++ source files are compiled and linked by the instrumented LLVM tool chain. Although LLVM itself supports linking-time optimization, there is a convenient tool, WLLVM [16], for building whole-program LLVM bitcode files. WLLVM is a python-based compiler wrapper while the real compiling job is still done by LLVM. Once the IR module is ready, our detector will traverse the whole module to collect all cast stubs inserted at the previous stage. It will then query the analyzer for points-to information. After checking whether the actual type is compatible with the specified type in a cast operation, a report about potential type confusion errors will be given.

A. Keeping type information

Programmers tend to believe that an LLVM IR module contains as much type information as the original C++ source code. Unfortunately, it is not true. According to the placement `new` expression in the following C++ code snippet , it is clear that the pointer `ptr` points to a `Sub` object. However, even compiled at `-O0` optimization level, the type information about `Sub` is lost in the generated LLVM IR.

```
char buf[1024];       // C++ Source Code
int main(){
    Base *ptr = new (buf) Sub;
    g(ptr);
    return 0;
}
define i32 @main() { // LLVM IR
  call void @_Z1gP4Base(
   %struct.Base* bitcast (
```

```
    [1024 x i8]* @buf to %struct.Base*))
  ret i32 0
}
```

Placement `new` is a C++ mechanism to separate memory allocation from initialization. It is widely used in a great range of C++ libraries and applications. To keep type information for placement `new` reliably, we modify the C++ frontend of LLVM to emit the following IR when a placement `new` expression is compiled.

```
!909 = !{%struct.Sub* null}
%0 = call i8* @__placement_new_stub(
        [1024 x i8]* @buf) DstTypeInfo !909
```

Stub functions like `__placement_new_stub` are implemented as a runtime library in the `compiler-rt` project [17]. Type information is assigned to the stub call instruction as metadata. Moreover, the `extapi` module in SVF is updated, such that `__placement_new_stub` is treated as a special kind of memory allocation function during pointer analysis. Hence, SVF will create an abstract object and its type can be inferred. Function `__placement_new_stub` only needs to return the value of the parameter passed to it. In this way, the original semantics is respected. After static analysis, all these stubs could be removed and the performance of the program will not be hurt.

In order to maintain type information reliably for a cast expression, the following cast stub functions are also added into the `compiler-rt` project. Given a `static_cast` expression, at code generation stage, the instrumented LLVM compiler will emit a call instruction to invoke `__static_stub()`. Related type information is attached to this call instruction as metadata. The other two cast operations, `reinterpret_cast` and `dynamic_cast`, can be handled in the same way.

```
void * __static_stub(void *, void *);
void * __reinterpret_stub(void *, void *);
void * __dynamic_stub(void *, void *);
```

This not only maintains type information completely, but also provides a method for us to classify different casts in the IR module easily. In this way, we are able to focus on only `static_cast` expressions when needed.

B. Main logic of TCD

Once the whole-program LLVM bitcode file is generated by LLVM, the next step is to collect cast stubs inserted in the IR module. An LLVM pass is written for this purpose, which visits every instruction in each function of the IR module. If it is an instrumented call instruction, it will be added into a set for further processing.

The main logic for detecting type confusion error is demonstrated in Algorithm 1. Type information about the source pointer and destination pointer of a cast expression is fetched from metadata. As there might be some pointer value adjustment in case of multiple inheritance, a destination pointer does not always equal its source pointer. To reflect the semantics correctly, the destination pointer, rather than the source pointer, is passed as an argument to query points-to information in line 6. Different command line options are supported for users to choose different levels of precision during pointer analysis.

Algorithm 1: Find Potential Type Confusion Errors

Input : A set containing cast stub callsites
Output: Positions of potential type confusion errors

1 **Procedure** FINDPOSSIBLETYPECONFUSIONS($castset$)
2 **foreach** $stub \in castset$ **do**
3 $(totalCnt, badCnt, rightCnt) \leftarrow (0,0,0)$;
4 $(srcTy, dstTy) \leftarrow getTypeViaMetadata(stub)$;
5 $dstPtr \leftarrow getDstPointer(stub)$;
6 $pts \leftarrow getPointsTo(dstPtr)$;
7 **foreach** $obj \in pts$ **do**
8 $totalCnt$++;
9 $actualTy \leftarrow getActualType(obj)$;
10 **if** *isBad*($srcTy$,$dstTy$,$actualTy$) **then**
11 $badCnt$++;
12 **end**
13 **else if** *isRight*($srcTy$,$dstTy$,$actualTy$) **then**
14 $rightCnt$++;
15 **end**
16 **end**
17 **if** $rightCnt = totalCnt$ **then**
18 insert $stub$ into the set for all-safe casts
19 **end**
20 **else if** $badCnt = totalCnt$ **then**
21 insert $stub$ into the set for all-bad casts
22 **end**
23 **else**
24 insert $stub$ into the set for uncertain casts
25 **end**
26 **end**

For each abstract object pointed by the destination pointer, its type is inferred and checked. Since path-insensitive pointer analysis is adopted in TCD, all the branch conditions are ignored. In other words, we do not check whether a series of branch conditions are satisfiable. If all the abstract objects have compatible types, it is a safe cast. However, if they are all incompatible, it is a only strong suggestion that it might be a bad cast. The reason is that it may be unreachable for these incompatible objects to get to the cast statement. If both compatible types and incompatible ones exist simultaneously, the cast expression is regarded as an uncertain cast.

C. Handling minus offset in multiple inheritance

In general, the offset generated by LLVM to access a field in a C++ object is either positive or zero. However, this does not hold any more in case of multiple inheritance. As depicted in Figure 1, when a `B *` pointer is converted into a `C *` pointer, the pointer value is adjusted. The LLVM IR for this downcast is shown as below. It is clear that there is a minus offset (-8) in the `getelementptr` instruction. On a 64-bit machine, the size of the virtual table pointer in Figure 1 is 8 bytes.

```
%1 = bitcast %struct.B* %bptr to i8*
%ptr = getelementptr i8, i8* %1, i64 -8
%2 = bitcast i8* %ptr to %struct.C*
```

Before pointer analysis, SVF needs to set up a memory model for the program by processing each IR instruction. The offset in a `getelementptr` instruction is modeled as an index of a flatten struct. If the actual object pointed by the pointer `%1` in the `getelementptr` instruction is a `B` sub-object within a larger `C` object, the minus offset makes sense. Otherwise, if it points to a separate B object, the minus offset is a hint of type confusion error. However, memory modeling happens earlier than constraints solving in SVF, meaning that points-to information of the pointer `%1` is still unclear when modeling memory.

To tackle this issue, a mechanism for lazy evaluation is implemented when a minus offset exists. At memory modeling stage, the minus offset is only saved for later processing. When solving constraints, the object pointed by pointer `%1` can be figured out. If it is a sub-object at byte offset 8 in a larger `C` object, then after adding the minus offset (-8), it points to the offset 0 of the `C` object.

D. Synthesizing svf.main() as the entry function

When it comes to initialization of global objects, the semantics of C++ is quite different from that of C. In a C program, global variables are initialized with constant data. After linking, the data already exists in an executable and will be loaded into memory by operating system when needed. It means that the `main()` function in C is often executed before other user-defined functions. However, global C++ objects should be initialized with corresponding C++ constructors. In other words, these constructors should be run before the `main()` function in a C++ application.

In an LLVM IR module, these constructors are put into a special code section (`.text.startup`) and the linker makes sure that these startup functions will be executed before `main()`. To obtain flow-sensitivity, we synthesize a pseudo entry function `svf.main()` to call all the initializer functions first and then invoke `main()`. An example of `svf.main()` is shown as follows.

```
define void @svf.main(i32, i8**, i8**) {
entry:
    call void @_GLOBAL__sub_I_1.cpp()
    call void @_GLOBAL__sub_I_2.cpp()
    %3 = call i32 @main(i32 %0, i8** %1)
    ret void
}
```

The number of arguments passed to `main()` could be 0, 2, or 3, which can be determined after analyzing the signature of `main()` in the IR module. The synthesized `svf.main()` will guide SVF to create a call graph correctly for flow-sensitive pointer analysis.

E. Supporting operator new and new[]

C++ applications are often performance-sensitive. Sometimes, they need to take control of how memory is allocated by customizing their own `operator new` function. Even a C++ class might have its distinctive memory allocator. For example, the class `cObject` in Omnetpp of SPEC CPU2006 defines its own version of `operator new`. After C++ name mangling, its function name becomes `_ZN7cObjectnwEm`. This may cause an issue: `operator new` functions defined in different classes will have absolutely different names. All these functions should be considered as special memory allocators in SVF; otherwise, it will be hard to associate type information with the memory allocated.

Although SVF models library functions like `malloc()`, it lacks a pattern matching mechanism to capture `operator new` functions defined in different C++ classes. For this reason, we modify the `extapi` module in SVF to model these functions. The mangled name of `operator new` in a class has a fixed pattern, which is related to its class name and function signature. Similarly, `operantor new[]` can be supported.

F. Different levels of precision

Based on SVF, TCD supplies different levels of analysis precision, including flow-insensitivity, context-insensitivity, flow-sensitivity and context-sensitivity and field-sensitivity. Field-sensitivity plays an important role in type confusion detecting. Without it, it would be impossible to distinguish different sub objects in multiple inheritance or composition. Demand-driven analysis [18] is also supported when time and memory budgets are limited (like IDEs), which allows users to select only those pointers they are interested in.

G. Location set with type information

The location set [19] in SVF is an important structure in field-sensitive analysis. It represents a flatten field in an abstract object. Elements in the same array could be collapsed into a single flatten field when array-insensitive analysis is adopted. In LLVM IR, a `getelementptr` instruction could access a sub-object at a specified offset in its base object. To further access a deeper field of the sub-object, another `getelementptr` instruction might be generated by LLVM. Fundamentally, a location set is the abstraction of a `getelementptr` instruction or the result of a series of `getelementptr` instructions.

As described in Section IV-C, lazy evaluation is necessary when handling minus offset. For the purpose of type confusion detection, base type information and a byte offset are added to each location set. We define an `add` operation between two location sets, which is not commutative. The result of this `add` operation is a new location set. Its base type is the same as left operand and its byte offset is the sum of the two operands. The `add` operation is executed when SVF is solving constraints during pointer analysis. An abstract object might correspond to multiple concrete objects allocated on stack or heap memory. Since we have already kept type information for `new` expressions, its type can be fetched via metadata in the IR module. As for other heap allocation functions like `malloc()`, they should be modeled in the `extapi` module

Algorithm 2: Get the Largest Sub-Type

Input **:** Base struct type and a byte offset
Output**:** The largest sub-type

```
1  Procedure GETSUBSTRUCT(baseType,offset)
2  structSize ← getSizeInBytes(baseType);
3  if offset < 0 OR offset % structSize = 0 then
4  |  return baseType;
5  end
6  offset %= structSize;
7  i ← 0;
8  while i < numOfElements(baseType) do
9  |  if getEleOffset(baseType, i) > offset then
10 |  |  break;
11 |  end
12 |  eleOffset ← getEleOffset(baseType, i);
13 |  eleType ← getEleType(baseType, i);
14 |  i++;
15 end
16 while eleType is ArrayType do
17 |  eleType ← getArrayElementType(eleType);
18 end
19 if eleOffset = offset then
20 |  return eleType;
21 end
22 else
23 |  eleOffset ← offset - eleOffset;
24 |  return GETSUBSTRUCT(baseType,offset);
25 end
```

in SVF and tracked through a def-use list to find its type. Type information of global and local objects is explicit.

Given a specified offset in an object, there may be multiple possible types at that position. For instance, at offset 0 in Figure 1, the object could be treated as either `struct C` or `struct A`. To keep it simple, we choose the largest sub-object's type as their representative. In this example, the largest sub-object at offset 0 is `C`, and `B` at offset 8 in Figure 1. Although offset 12 is still within the scope of `C` object, it is not the beginning position of any struct object. Algorithm 2 illustrates how to get type information for the largest sub-object at a designated byte offset within a base type. The `while` loop in line 8 finds a sub-object that is closest to the specified byte offset. If the sub-object happens to locate at the given byte offset, its type is returned at line 20; otherwise, we recursively find it in line 24 in Algorithm 2.

V. EVALUATION

There are two objectives of TCD: one is to expose safe casts to provide further optimization opportunity for existing dynamic tools; the other is to supply a big picture for C++ programmers about where potential type confusion errors might be, including those that are hard to reach for current dynamic detectors.

In order to evaluate TCD, we have applied it to a set of more than 40 C++ applications, which represent typical system software, standard benchmarks, cross-platform tools, and widely-adopted machine learning applications respectively. Our experiment shows that on average 14.22% casts are identified as safe. This figure can even jump to 71.78% in machine learning applications of `dlib` [20]. TCD is also able to detect six type confusion errors in our test cases.

A. Experiment Setup

The platform for our experiment consists of a 3.20GHz Intel Xeon(R) E5-1660 v4 CPU and 256 GB memory, running Ubuntu operating system. It should be noted that, even with sparse analysis techniques, pointer analysis still consumes much memory, especially when flow-sensitivity is enabled. So a large-volume memory chip is necessary for our experiment.

We use WLLVM as a compiler wrapper to build whole-program bitcode files for all C++ applications in our test. In order to promote memory references to be register ones, `-mem2reg` is set.

Applications from LLVM tools, SPEC CPU2006, QT and a machine learning platform are used to evaluate TCD. To keep a balance between precision and scalability, we adopt the demand-driven solution introduced in [18] for pointer analysis. The budgets for flow-sensitivity and context-sensitivity are both configured as no more than 10000 value-flow edges per query. The maximum context limit for context-sensitivity is set to 3, meaning that at most 3 calling contexts are modeled. When budgets run out, the analysis might regress to Andersen-style pointer analysis, leading to loss of precision.

B. Safe casts

In Table I, the number of safe casts in each application is given in column 2. The next column is for all-bad casts, in which all the abstract objects pointed by a pointer are of incompatible type. Column 4 presents the number of uncertain casts, pointing to both compatible and incompatible objects. The total number of cast expressions is listed in column 5 and the percentage of safe casts is shown in the last column.

The last three C++ applications in Table I are from CPU2006. There is not any all-bad cast detected in `soplex` and `namd`. However, 54.45% of the casts in `soplex` are safe. The percentage of safe casts in the QT tool, `qmake`, is much lower, only 3.68%. The second largest figure occurs in `yaml2obj`, where 38.11% of the cast operations can be determined statically to be safe. In other applications, the percentage fluctuates between 7.88% and 18.58%.

On average, in the test cases shown in Table I, 14.22% of the casts are safe. It provides a considerably large space for existing dynamic detectors to remove unnecessary runtime checks.

Due to the widespread use of deep learning techniques in mission-critical domains like face recognition and self-driving systems, we also evaluate `dlib`, a popular machine learning library (with 6475 stars) on github. In this test suit, `TCD` finds that 71.78% casts can be considered as safe. In Figure 3,

TABLE I
STATISTICS OF LLVM TOOLS, QT AND CPU2006

	Safe	Bad	Uncertain	Total	Safe_Percentage
llvm-objcopy	880	14	4510	5404	16.28%
llvm-cxxdump	876	25	4477	5378	16.29%
llvm-cvtres	859	13	4420	5292	16.23%
llvm-size	850	13	4418	5281	16.10%
clang-format	433	7	3854	4294	10.08%
llvm-diff	661	5	3360	4026	16.42%
llvm-tblgen	543	65	2999	3607	15.05%
llvm-dis	616	6	2973	3595	17.13%
llvm-stress	390	6	2794	3190	12.23%
llvm-profdata	321	9	2353	2683	11.96%
clang-tblgen	201	20	861	1082	18.58%
yaml2obj	383	2	620	1005	38.11%
llvm-rc	64	6	404	474	13.50%
llvm-opt-report	42	4	408	454	9.25%
yaml-bench	42	2	346	390	10.77%
FileCheck	44	3	321	368	11.96%
llvm-bcanalyzer	37	3	292	332	11.14%
llvm-cxxfilt	26	2	302	330	7.88%
llvm-mt	35	3	289	327	10.70%
llvm-config	40	2	254	296	13.51%
llvm-strings	25	2	267	294	8.50%
llvm-undname	25	2	245	272	9.19%
not	26	2	240	268	9.70%
qmake	61	9	1586	1656	3.68%
soplex	18	0	15	33	54.55%
omnetpp	13	1	234	248	5.24%
namd	0	0	1	1	0.00%

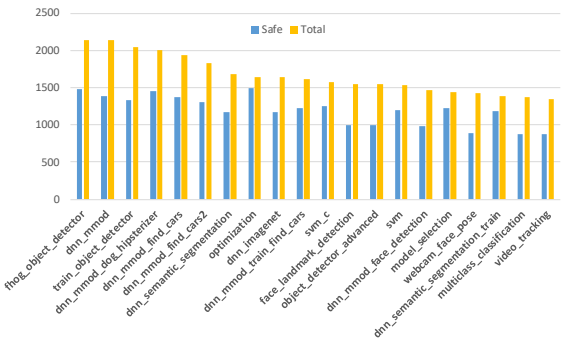

Fig. 3. Safe casts in Dlib.

the numbers of safe and total casts are shown in parallel for comparison. Since this group of test cases exhibit much higher average percentage of safe casts, we conservatively separate them from other applications. However, it is a hint that there are some applications like `dlib` which might have extremely large space for static optimization.

C. Type confusion errors detected

To evaluate whether TCD could expose some potential type confusion errors in our test, we manually check the all-bad casts shown in column 3 of Table I. Since conditional guards are ignored in our analysis, there are some false positives in our results. For example, in the test case `clang-format`, all objects pointed by a pointer are of type `DiagnosticInfoInlineAsm`, and they are incompatible with the destination type, `DiagnosticInfoOptimizationBase`. However, since there is a branch condition to check whether it is legal to conduct the type cast, the cast operation would never happen.

Even so, TCD can expose six type confusion errors in libstdc++ and QT, shown in Table II. The first three errors have already been reported by dynamic tools, namely, CAVER [3] and HexType [6], Our experiment shows that TCD is able to expose these bugs with static analysis. Moreover, TCD also finds three new type confusion errors in QT, namely, the last three errors in Table II, which have not been reported by previous research.

TABLE II
TYPE CONFUSION ERRORS DETECTED

Project	File	Function
libstdc++	stl_tree.h	_Link_type _M_end()
libstdc++	stl_tree.h	_Const_Link_type _M_end() const
QT	qmap.h	Node *end()
QT	qjsonvalue.cpp	void QJsonValue::detach()
QT	qjsonobject.cpp	bool QJsonObject::detach2()
QT	qjson.cpp	void Data::compact()

Function templates in libstdc++ are widely used in C++ applications, including the test cases in our experiment. The function templates listed in Table II might be instantiated multiple times with different type parameters. These instantiations are treated as different functions in the IR module. In the test case `llvm-tblgen`, all the 65 bad casts are in different instantiations of the C++ function template `_M_end()`.

The last three type confusion errors in Table II are caused by ad hoc implementation of C++ inheritance, the pattern of which is shown as below.

```
Base *bp = (Base*)malloc(
        sizeof(Base) + const_sz + variable_sz);
if(flag)
  Derived1 *d1 = static_cast<Derived1 *>(bp);
else
  Derived2 *d2 = static_cast<Derived2 *>(bp);
```

At the memory-allocation site, a memory block larger than the size of class `Base` is allocated, where the sum of `const_sz` and `variable_sz` represents the size of extra memory needed by its derived classes. If the sum operation in `malloc()` does not synchronize with the modification of these derived classes, a dangerous vulnerability will lurk when the size of a derived class is larger than the memory block allocated by `malloc()`. In other words, a downcast from `Base` to its derived class might lead to an out-of-bound memory access.

As for the uncertain casts in Table I, a command line option is provided for users to decide which uncertain casts should be reported. For example, the motivation program in Figure 1 can be reported with `-report-threshold=1` in less than a second. By contrast, dynamic detectors are still idling after a 24-hour fuzzing test. It is an evidence that TCD, as a static detection tool, can easily escape the trap caused by hard-to-satisfy conditions.

VI. RELATED WORK

In this section, we summarized research work similar with TCD, including existing runtime type confusion detectors and control flow integrity techniques.

Runtime Bad-casting Detectors. Undefined Behavior Sanitizer (UBSan) [2] modifies C++ programs at compile time and substitutes `static_cast` with `dynamic_cast`. In this way, the C++ runtime library function `__dynamic_cast()` acts as a dynamic detector. It relies on RTTI to detect whether it is a bad cast. However, this method can not be used to protect non-polymorphic classes.

To include non-polymorphic classes, CAVER [3] performs runtime type check for all classes in C++. By customizing a heap allocator, objects allocated on heap are aligned and can be mapped to their metadata at `O(1)` complexity. However, it is not easy to apply this direct-mapping scheme for stack objects, as radical semantic changes are needed. The authors in [3] chose red-black trees to store metadata of stack and global objects at `O(log N)` complexity. As a result, their solution incurs high overhead if most of the allocated objects are on stack.

Their work was further improved by TypeSan [4], HexType [6] and EffectiveSan [7]. TypeSan relies on a compact memory shadowing mechanism to trace all objects (including stack, global and heap objects) in a uniform way, such that the overhead of tracing stack objects is greatly reduced. One limitation of TypeSan is that it might conflict with ASLR. Considering that, HexType employs a hash table and a red-black tree to trace objects. EffeciveSan handles not only type errors but also memory errors.

Control Flow Integrity. The seminal research on Control-Flow Integrity (CFI) was proposed by Abadi et al. [21] in 2005. They noticed that control flow graph is an inherent property of every program and all runtime execution paths should be constrained within the scope of its legal control flow graph. They implemented the first CFI enforcement at binary level by machine-code rewriting. Their work has been followed by a vast number of other research groups in the past decade. Two key challenges exist for all CFI schemes: how to make dynamic checks more efficient and how to make control flow targets more precise.

To reduce high overhead of CFI runtime checks, coarse-grained CFI was proposed. Given an indirect call instruction, it only tests whether the target address is the entry of a function. As a result, incompatible functions are merged together, leading to a coarse-grained solution. CCFIR [14] is one of such coarse-grained CFI schemes. Later, fine-grained CFI mechanisms such as PICFI [22] and IFCC [23] were explored, which utilize high-level semantics of the program to classify different functions.

VII. CONCLUSIONS

In this paper, we have presented TCD, a static type confusion detector which adopts pointer analysis techniques to report potential type confusion errors. Our static method could provide C++ programmers with a whole picture about where type confusion errors might be, which is helpful to discover errors at earlier stage. On the contrary, existing dynamic solutions are usually combined with fuzzing techniques. Even so, only part of the whole program space is explored in practice. Our experiment demonstrates that TCD is able to report potential bad casts. It can also be used as an auxiliary optimizer to remove redundant runtime checks for existing dynamic detectors. The shortage of TCD is that it might contain false positives and a considerable amount of manual work is required to verify whether it is a true positive. In order to reduce manual labor, we are considering machine learning as a possible method to filter out fake targets.

REFERENCES

[1] "Common vulnerabilities and exposures," https://cve.mitre.org/, accessed January 23, 2019.

[2] "Undefined behavior sanitizer," https://www.chromium.org/developers/testing/undefinedbehaviorsanitizer, accessed January 23, 2019.

[3] B. Lee, C. Song, T. Kim, and W. Lee, "Type casting verification: Stopping an emerging attack vector." in *USENIX Security Symposium*, 2015, pp. 81–96.

[4] 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.

[5] "Address space layout randomization," https://pax.grsecurity.net/docs/aslr.txt, accessed January 28, 2019.

[6] Y. Jeon, P. Biswas, S. Carr, B. Lee, and M. Payer, "Hextype: Efficient detection of type confusion errors for c++," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 2373–2387.

[7] G. J. Duck and R. H. Yap, "Effectivesan: type and memory error detection using dynamically typed c/c++," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2018, pp. 181–195.

[8] 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*, ser. ACSAC '18. New York, NY, USA: ACM, 2018, pp. 518–528.

[9] P. Muntean, S. Wuerl, J. Grossklags, and C. Eckert, "Castsan: Efficient detection of polymorphic c++ object type confusions with llvm," in *European Symposium on Research in Computer Security*. Springer, 2018, pp. 3–25.

[10] M. Zalewski, "American fuzzy lop (afl) fuzzer," http://lcamtuf.coredump.cx/afl/, accessed January 23, 2019.

[11] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 697–710.

[12] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 2016, pp. 265–266.

[13] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.

[14] D. Gens, S. Schmitt, L. Davi, and A.-R. Sadeghi, "K-miner: Uncovering memory corruption in linux," in *2018 Network and Distributed System Security Symposium (NDSS'18)*, 2018.

[15] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: towards a desired directed grey-box fuzzer," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 2095–2108.

[16] T. Ravitch, "Whole program llvm," https://github.com/travitch/whole-program-llvm, accessed January 23, 2019.

[17] "The compiler-rt project," https://compiler-rt.llvm.org/, accessed January 23, 2019.

[18] Y. Sui and J. Xue, "On-demand strong update analysis via value-flow refinement," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 460–473.

[19] Y. Sui, X. Fan, H. Zhou, and J. Xue, "Loop-oriented pointer analysis for automatic simd vectorization," *ACM Trans. Embed. Comput. Syst.*, vol. 17, no. 2, pp. 56:1–56:31, Jan. 2018.

[20] "Dlib c++ library," https://github.com/davisking/dlib/, accessed January 26, 2019.

[21] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 340–353.

[22] B. Niu and G. Tan, "Per-input control-flow integrity," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 914–926.

[23] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in gcc & llvm," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 941–955.