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

Changwei Zou* , Yulei Sui†, Hua Yan*, Jingling Xue*
*School of Computer Science and Engineering, University of New South Wales, Australia
†Faculty of Engineering and Information Technology, University of Technology Sydney, Australia

***Abstract*—For performance reasons, C++, albeit unsafe, is often the programming language of choice for developing software infrastructures. A serious type of security vulnerability in C++ programs is type confusion, which may lead to program crashes and control flow hijack attacks. While existing mitigation solutions almost exclusively rely on dynamic analysis techniques, which suffer from low code coverage and high overhead, static analysis has rarely been investigated.**

This paper presents TCD, a static type confusion detector built on top of a precise demand-driven field-, context- and flow-sensitive pointer analysis. Unlike existing pointer analyses, TCD is type-aware as it not only preserves the type information in the pointed-to objects but also handles complex language features of C++ such as multiple inheritance and placement `new`, making it therefore possible to reason about type casting in C++ programs. We have implemented TCD in LLVM and evaluated it using seven C++ applications (totaling 526,385 lines of C++ code) from Qt, a widely-adopted C++ toolkit for creating GUIs and cross-platform software. TCD has found five type confusion bugs, including one reported previously in prior work and four new ones, in under 7.3 hours, with a low false positive rate of 28.2%.

***Index Terms*—type confusion, bug detection, software security, pointer analysis, static analysis**

I. Introduction

Large software systems, such as Linux kernels, compilers, browsers, and Java Virtual Machines, are the cornerstone for the modern software industry. To seek for high performance and low-level control over memory allocation, almost all of these fundamental products are implemented in C and/or C++, both of which lack memory safety [1]–[3] and type safety [4], leading to severe software security vulnerabilities [5]–[7].

With the rapid increase in both complexity and scale of software, teamwork is necessary for all large projects. 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 may point to. If an object is cast from a base class to a derived one, it is their responsibility to ensure that type casting is correct at runtime. If this fails, a type confusion error (or bug) will occur, allowing attackers to corrupt out-of-bound data and hijack control flow by tampering with code pointers [6]. As shown in Figure 1, the number of type confusion vulnerabilities reported on the CVE website [8] has surged rapidly in recent years. It is thus imperative to develop program analysis techniques to detect type confusion bugs.

Prior Work: Dynamic Analysis. Existing mitigation solutions [6], [9]–[15] are all dynamic. At compile-time, instrumentation code is added to collect the information for the objects in a program. At runtime, the safety of type casting for each object is verified. To trace all the objects efficiently, data structures, such as red-black trees [10], shadow memory [12], hash tables [6] and low fat pointers [13], have been used.

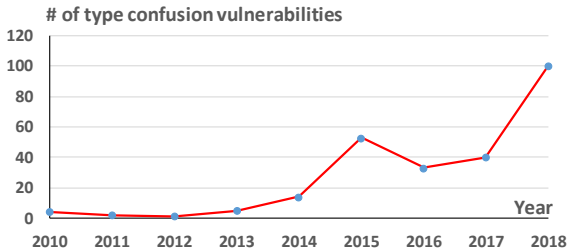

Fig. 1. The rapid increase of type confusion vulnerabilities [8].

This Work: Static Analysis. Dynamic analysis tools can detect type confusion bugs precisely at runtime, but suffer from the two well-known problems: low code coverage and high performance overhead. In contrast, static analysis tools can find potential software security vulnerabilities that are harder to find dynamically in the entire codebase earlier in the development life cycle. Thus, recent years have witnessed widespread adoption of static tools in software industries [16]. However, there has been little research on developing static techniques and tools for finding type confusion bugs in C++ source code. To the best of our knowledge, this work represents the first such investigation.

There are several challenges faced in finding type confusion bugs efficiently and precisely in a static manner. First, a precise and efficient inter-procedural analysis is needed, but current bug-finding tools [16] perform the majority of their analysis tasks intra-procedurally. Second, a precise and efficient type-aware pointer analysis is also needed to track the points-to information. Due to complex C++ language features such as multiple inheritance, the pointer analyses developed for C [17] and Java [18] can not be directly applied in type confusion detection. Worse still, some type information can be lost in the Intermediate Representations (IRs) operated by C++ compilers such as LLVM [19], even under `-O0`.

In this paper, we address these challenges by introducing TCD, a static type confusion detector built on top of a C++ compiler front-end modified to provide some cast-related type annotations required in the IR and a precise demand-driven

978-1-7281-4982-0/19/$31.00 ©2019 IEEE
DOI 10.1109/ISSRE.2019.00037

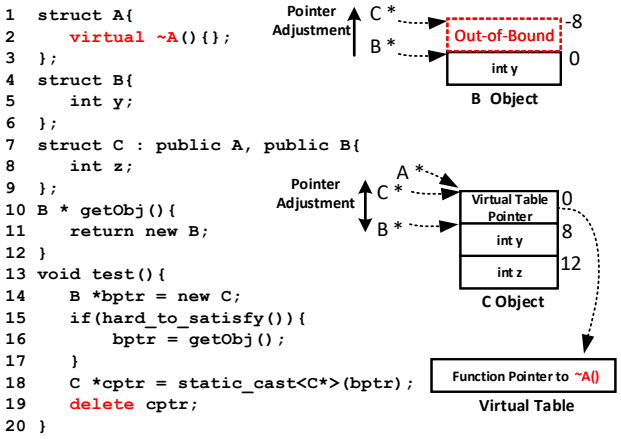

Fig. 2. A motivating example of TCD.

field-, context- and flow-sensitive pointer analysis leveraged to compute the points-to information inter-procedurally. However, unlike existing pointer analyses for C and Java, TCD is type-aware as it not only preserves the type information in the objects pointed to by a pointer but also handles complex language features of C++ such as multiple inheritance and placement `new`, making it therefore possible to reason about type casting in C++ programs.

Figure 2 illustrates how a type confusion bug occurs and why both existing pointer analyses and dynamic analyses are inadequate. Here, class `C` inherits from both `A` and `B`, with the memory layout of `B` and `C` objects shown. If `hard_to_satisfy()` in line 15 evaluates to true (even though very infrequently), `bptr` in line 18 will point to a `B` object created in line 11. Due to multiple inheritance, the downcasting from `B` to `C` in line 18 will trigger a pointer adjustment as shown by the single-head arrow. As a result, the `B` object will be mistakenly treated as a `C` object, leading to a type confusion error. The out-of-bound area will be misunderstood as containing a virtual table pointer. The `delete` operation in line 19 will attempt to invoke the virtual destructor of class `C` (inherited from `A`), causing an illegal dereference for the out-of-bound area, and consequently, a program crash. Worse still, if this area is controlled by some motivated attackers, they can easily hijack control flow by forging a virtual table pointer inside [20]–[22]. Note that if `hard_to_satisfy()` in line 15 evaluates to false, there are also pointer adjustments for the `C` object created, once from `C` to `B` in line 14 (upcasting) and once from `B` to `C` in line 18 (downcasting), as shown by the two-head arrow.

The state-of-the-art pointer analyses for C such as SVF [17] cannot be directly applied in detecting type confusion bugs in C++ programs. There are two reasons for this. First, the pointed-to objects discovered for a pointer do not carry enough type information. Second, the missing type information, even if added directly, can be incorrect in the presence of multiple inheritance, as the pointer adjustments as shown in Figure 2 are not taken into account. SVF [17] expects every C program being analyzed to be C-compliant. Given $p = q + \mathit{offset}$, where p and q are pointers and *offset* is a non-negative integer (which is not necessarily known at compile time), SVF therefore assumes that p will always point to the objects that q points to (regardless of what *offset* is). For a C++ program, however, this assumption no longer holds due to the pointer value adjustments between `B` and `C`, as shown in Figure 2. Specifically speaking, the pointer adjustment from `B` to `C` in line 18 will generate a negative offset (-8) in the LLVM-IR. In order to figure out the type of a pointed object obj, the information we need can be represented as $(t, \sum_1^n \mathit{off}_i)$, where `t` is the type of the object containing obj and $\sum_1^n \mathit{off}_i$ is the offset accumulated during field-sensitive pointer analysis. Ignoring the negative offset caused by downcasting (e.g., in line 18) will eventually lead to an incorrect offset accumulated in $(t, \sum_1^n \mathit{off}_i)$ during program analysis, thereby resulting in a wrong type inferred for the object obj.

Dynamic analysis tools [6], [9], [10], [12]–[15] can hardly find the type confusion bug in line 18. To find bugs at testing stage, these tools are usually driven by a fuzzer like AFL [23] to repeatedly run the program with different inputs. Suppose that `hard_to_satisfy()` in line 15 consists of testing an 8-byte integer (read as the standard input) against a magic number, `0x12345678deadbeef`. The modern grey-box fuzzer, AFL, could not expose the bug in 24 hours (as line 16 was never reached during a total of 431 million program runs). Recent fuzzing tools such as T-Fuzz [24] attempt to alleviate this issue. However, constraint solvers they rely on may still not be powerful enough to solve complex constraints.

In contrast, as a static detector, TCD does not need to really run the program, thus bypassing the `hard_to_satisfy()` condition in line 15 that thwart dynamic detectors.

This paper makes the following contributions:

- We describe a new type-aware pointer analysis that can reason about the type information in the pointed-to objects for C++.
- We introduce TCD, a type confusion detector implemented in LLVM, for finding type confusion bugs.
- We have evaluated it using seven C++ applications (totaling 526,385 lines of C++ code) from Qt, a widely-adopted C++ toolkit for supporting GUIs and cross-platform software. TCD has found five type confusion bugs, including one reported previously in prior work and four new ones, in under 7.3 hours, with a low false positive rate of 28.2%.

The rest of this paper is organized as follows. Section II reviews type casting and pointer analysis. Section III presents the design and implementation of TCD. Section IV evaluates TCD, showing that TCD is able to detect new bugs that evaded previous approaches. Section V discusses the related work. Finally, Section VI concludes the paper.

II. Background

A. Type Casting in C++

C++ introduces four keywords to support four different kinds of casts, `static_cast`, `reinterpret_cast`,

`dynamic_cast` and `const_cast`. In particular, `static_cast`, which is checked at compile time, is mainly used to cast a pointer of a base class to a pointer of a derived class. As illustrated in Figure 2, pointer adjustments will be performed for both upcasting and downcasting in the case of multiple inheritance if the two related types have different offsets. Unlike `static_cast`, `dynamic_cast` performs a type conversion between two polymorphic classes (with virtual tables). At compile time, a C++ compiler inserts code to call `__dynamic_cast()`, a C++ library function, to enforce the semantics required. At runtime, `__dynamic_cast()` will search the type information stored in virtual tables to check whether a `dynamic_cast` is safe or not. In other words, `dynamic_cast` does not lead to any type confusion bug.

As the C++ version of C-style casting, `reinterpret_cast` will not modify the underlying pointer, even in the presence of multiple inheritance. Furthermore, `reinterpret_cast` can be used between unrelated classes while `static_cast` and `dynamic_cast` are often conducted in the same class hierarchy. As for `const_cast`, its main purpose is to discard the read-only constness on an object. While `const_cast` may still introduce security issues [6], its protection is an orthogonal issue that we do not address in this paper.

In this paper, we consider the type confusion bugs caused by `static_cast` and `reinterpret_cast`. If a base-class pointer is cast to a derived-class pointer when the underlying object is incompatible with the derived class (at runtime), then a type confusion bug is said to have occurred.

Type confusion bugs can be used to corrupt sensitive data, code pointers or virtual table pointers. How to exploit these vulnerabilities is beyond the scope of this paper.

B. Pointer Analysis

Pointer analysis, which is virtually the basis of all other program analyses, determines statically the set of objects that may be pointed to by a pointer. To achieve precision, a pointer analysis is expected to be *field-sensitive* [25] (by distinguishing different fields of an object), *flow-sensitive* [26] (by distinguishing the flow of control), and *context-sensitive* [27] (by distinguishing the calling contexts for a function). A pointer analysis is a *whole-program* analysis if it computes the points-to information for all the pointers and *demand-driven* if it computes the points-to information only for some given pointers in the program.

SVF [17] is an open-source pointer analysis platform for C/C++ implemented in LLVM [19]. It has been used in a number of research projects, including a framework for analyzing Linux kernels [28] and a directed grey-box fuzzer for software testing [29]. SVF can perform both whole-program and demand-driven pointer analyses for C/C++ programs [30]. As discussed in Section I, however, SVF cannot be used directly in detecting type confusion bugs in C++ programs. In this paper, SVF will be leveraged to accomplish this objective.

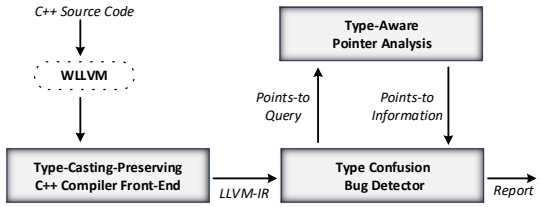

Fig. 3. The TCD framework for detecting type confusion bugs for C++.

III. TCD: Design and Implementation

We have designed and implemented TCD in LLVM for detecting type confusion bugs in C++ programs. As shown in Figure 3, TCD consists of three components: a type-casting-preserving C++ compiler front-end, a type-aware pointer analysis and a type confusion bug detector. Given a C++ program, its source files are compiled and linked by the LLVM tool chain into a single LLVM-IR (known as bitcode) with the type annotations added for all cast-related expressions by our modified C++ compiler front-end in LLVM. WLLVM [31], a python-based compiler wrapper, is used to facilitate building whole-program LLVM bitcode files. Our type confusion bug detector issues a points-to query for a cast expression and then reports whether the cast is safe or not based on the points-to information computed on-demand by our type-aware pointer analysis, which operates on the LLVM-IR of the program.

For a pointed object obj, its type information can be represented as $(t, \sum_1^n off_i)$, where t is the type of the object containing obj and $\sum_1^n off_i$ is the sum of field offsets accumulated during field-sensitive analysis. The code added into LLVM is used to make sure that `t` is not missing during compiling, as discussed in III-A. The code added into SVF will initialize, accumulate and propagate type information such as $(t, \sum_1^n off_i)$ during pointer analysis. We respect the negative offsets caused by complex C++ language features such as multiple inheritance, which are ignored in previous research.

A. Type-Casting-Preserving C++ Compiler Front-End

When translating a C++ program into LLVM-IR, LLVM does not maintain all the type information required for detecting type confusion bugs. Below we describe how to rely on type-annotating stubs to provide such missing type information. Without these modifications, some type information would be lost. Similarly, we have also added corresponding stubs for `static_cast` and `reinterpret_cast`. These stubs are introduced to facilitate static analysis only and will be removed during program execution.

Placement `new` is widely used in a variety of C++ libraries and applications, by separating memory allocation from initialization. In Figure 4(a), `ptr` in line 6 points to a `Derived` object, which is created by a placement `new` expression at the specified memory location `buf`. However, in the LLVM-IR in Figure 4(b), generated by the default LLVM even under `-O0`, the type `Derived` for the object created is not available. Thus, a pointer analysis cannot correctly model the types of the objects created by such placement `new` expressions.

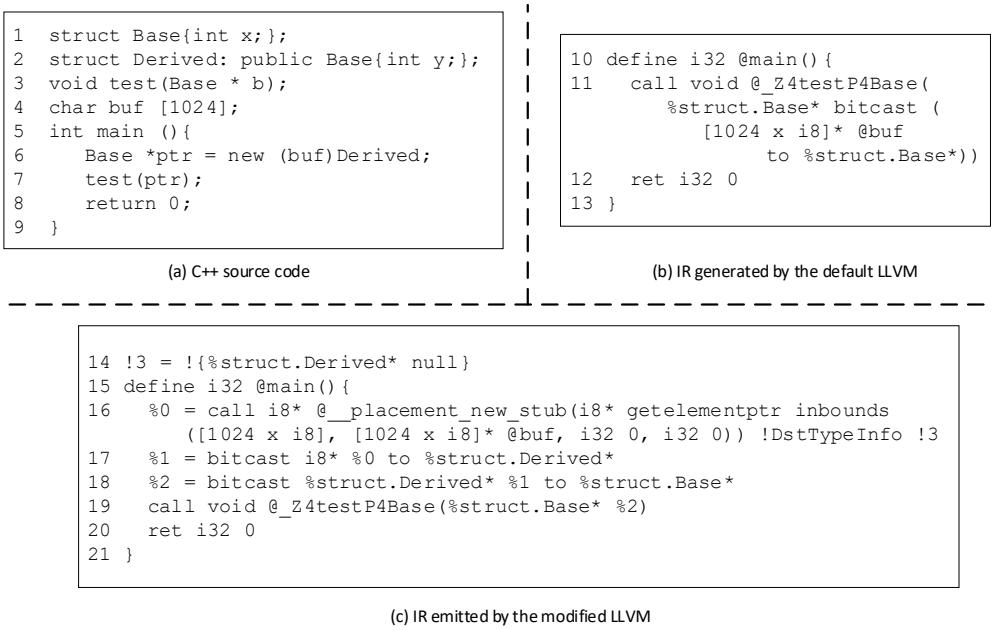

Fig. 4. Placement `new` stubs annotated with types.

Therefore, we have modified the LLVM C++ front-end to emit the LLVM-IR code shown in Figure 4(c). The type `Derived` is now made available as metadata (line 14) in a call to `__placement_new_stub()` in line 16. This stub can be modeled as a special memory allocator for a placement `new` expression during the pointer analysis, so that an object of an appropriate type can now be created for the placement `new` expression.

B. Type-Aware Pointer Analysis

We have developed our type-aware pointer analysis on top of the open-source SVF [17]. We describe only how to enrich SVF with the type information required for finding type confusion bugs.

1) Program Representation: For the purposes of performing pointer analysis, it suffices to consider the following six types of statements in LLVM-IR: $p = \&a$ (AddrOf), $p = q$ (Copy), $p = *q$ (Load), $*p = q$ (Store), $p = \&(q \rightarrow fld)$ (Field), and $fp(a_1, \cdots, a_n)$ (Call). Note that fp represents both a virtual and a static call. Passing arguments into and returning results from functions are modeled by copies. For an AddrOf statement $p = \&a$, known as an *allocation site*, a is a stack or global variable or a dynamically created abstract heap object. An array object is analyzed with its elements collapsed to a single field, denoted arr. For example, $x[i] = y$ can be seen as $x.arr = y$. For field accesses, $p = \&(q \rightarrow fld)$ is used. In LLVM-IR, $x = y \rightarrow fld$ is decomposed into $tmp = \&(y \rightarrow fld)$ and $x = *tmp$. Similarly, $x \rightarrow fld = y$ is decomposed into $tmp = \&(x \rightarrow fld)$ and $*tmp = y$.

SVF accelerates its analysis by computing the points-to information along the def-use edges pre-computed by a pre-analysis rather than along the control flow. Such def-use edges are often known as *value-flow edges*.

$$\frac{v = \&obj \qquad t = DeclaredType(obj) \qquad obj \text{ is a global/local object}}{\{obj\} \subseteq pt(v) \quad POS(obj,\ (t,\ 0))} \quad \text{[Global/Stack]}$$

$$\frac{v = \&obj \qquad t = InferType(obj) \qquad obj \text{ is a heap object}}{\{obj\} \subseteq pt(v) \quad POS(obj,\ (t,\ 0))} \quad \text{[Heap]}$$

$$\frac{p = \&(q \rightarrow fld) \quad obj \in pt(q) \quad POS(obj,\ (t,\ off_l)) \quad off_r = Offset(obj,\ fld)}{\{obj.fld\} \subseteq pt(p) \quad POS(obj.fld,\ (t,\ off_l + off_r))} \quad \text{[Field]}$$

$$\frac{p = q}{pt(q) \subseteq pt(p)} \quad \text{[Copy]}$$

$$\frac{POS(obj,\ (t,\ off)) \qquad t' = GetType(t,\ off)}{Type(obj) = t'} \quad \text{[AssignType]}$$

Fig. 5. Type-aware pointer analysis.

2) Rules: Figure 5 gives the rules for performing our type-aware pointer analysis with respect to AddrOf and Field. The rules for Load, Store and Call are the same as in traditional pointer analyses such as SVF and thus omitted. Once the analysis is completed, $pt(v)$ gives the points-to set

of v, i.e., set of objects obj pointed by v, where v is a variable or a field, and $Type(obj)$ gives the type of obj discovered.

Before going through the rules, we will first explain one key notation used in enabling the types of objects to be tracked in a field-sensitive analysis. In C++, an object of a particular type may contain multiple objects of different types residing at its different offsets. $POS(obj, (t, \mathit{off}))$ indicates that object obj resides at an offset off in an object of type t, where POS is a shorthand for position.

Below we will examine our five rules in turn. [GLOBAL/STACK] and [HEAP] are responsible for identifying the source of the type information. [FIELD] simply propagates the type information field-sensitively, by considering complex C++ language features such as multiple inheritance. [COPY] handles copy assignments in the standard way. Finally, [ASSIGNTYPE] maps each object to its type discovered.

a) [GLOBAL/STACK]: For a global or stack object allocation site, $v = \&obj$, where obj is created in the global area or on the stack, $t = DeclaredType(obj)$ can be directly read-off from the declared type of v. In this case, $obj \in pt(v)$, where $POS(obj, (t, 0))$ records the type t of obj. For example, given a local declaration "`T x`", `x` will be made to point to a stack object of type `T` in LLVM-IR.

b) [HEAP]: For a heap object allocation site, $v = \&obj$, where obj is created by calling a heap allocator (e.g., `malloc()`) in LLVM, we have $obj \in pt(v)$. In addition, $t = InferType(obj)$ is the type of obj inferred as follows. For standard allocators like `malloc()`, the types of their allocated objects are discovered by performing a standard def-use analysis [17]. For the objects created by placement `new` expressions, their types can be discovered from the placement `new` stubs introduced by our type-casting-preserving C++ compiler front-end (Figure 4). Finally, the `operator new/new[]` functions used in C++ classes will be recognized as special heap allocators for their corresponding classes.

c) [FIELD]: This rule handles $p = \&(q \rightarrow \mathit{fld})$. For an object $obj \in pt(q)$, suppose that fld resides at the offset off_r within obj, such that $POS(obj, (t, \mathit{off}_l))$ holds, then the location of the sub-object $obj.fld$ in an object of type t is identified by $(t, \mathit{off}_l + \mathit{off}_r)$. So $obj.fld \in pt(p)$, where $POS(obj.\mathit{fld}, (t, \mathit{off}_l + \mathit{off}_r))$ holds. Note that $Offset(obj, fld)$ returns the offset of $obj.\mathit{fld}$ in obj.

In traditional field-sensitive pointer analyses such as SVF [17], the byte offsets for accessing a field in an object are assumed to be non-negative. However, this assumption does not hold in the case of multiple inheritance in C++. Figure 6 illustrates the pointer adjustment that takes place due to downcasting. A negative offset, -8, is generated. Our type-aware pointer analysis will keep this negative offset and adjust internal memory model of pointer analysis to make sure that the accumulated offset is correct.

d) [COPY]: This rule handles $p = q$, which simply propagates the points-to information from q to p, such that $pt(q) \subseteq pt(p)$ holds.

e) [ASSIGNTYPE]: Once our analysis is completed, the type of obj, $Type(obj)$, is given by $GetType(t, \mathit{off})$ (defined below), where $POS(obj, (t, \mathit{off}))$ holds.

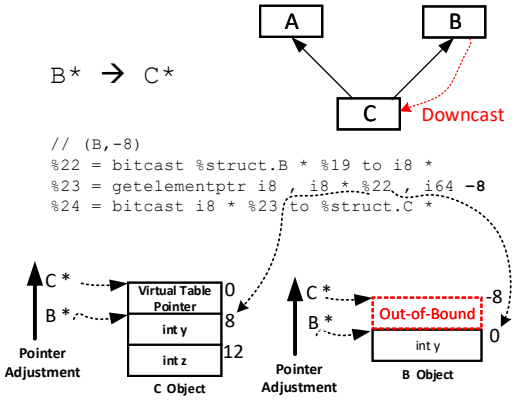

Fig. 6. Negative offsets for accessing fields in multiple inheritance.

In LLVM, inheritance in the high-level C++ source code is translated into composition in LLVM-IR. Consider the program in Figure 2 again. `A` is a base class of `C`. Thus, an object of `A` is contained inside an object of `C` (with both starting at the offset 0 as shown), implying that `A` is a type nested inside `C`. Recursively, `A` can have its own element types. Hence, at the offset 0 of a `C` object, its type can be treated as `C`, `A`, and the first element type of `A`. Among them, `C` is the largest one, as all the other types are directly or indirectly contained by `C`. As we start from $(t, 0)$ ([GLOBAL/STACK] and [HEAP]), $GetType(t, \mathit{off})$ therefore returns the largest type at (t, off) (nested inside $(t, 0)$), i.e., the type obtained after a sequence of field accesses starting from $(t, 0)$. In the special case when off is negative, $GetType(t, \mathit{off})$ returns t itself, as it is now outside the starting point $(t, 0)$.

TABLE I
THE LARGEST TYPE AT (t, off) ILLUSTRATED FOR FIGURE 2.

(t, off)	Possible Types	Largest Type
(C, 0)	{C, A, void ** }	C
(C, 8)	{B,int}	B
(C, 12)	{int}	int
(B, 0)	{B,int}	B
(A, 0)	{A,void **}	A

In Table I, we list all the possible types at different offsets off within `A`, `B`, and `C`, including the largest in each case, for the example program given in Figure 2. For example, at `(C,8)`, the largest type is `B`. Note that the type of the virtual table pointer in `A` is denoted as `void **`.

3) An Example: In Figure 7, we show how to apply our rules given in Figure 5 to analyze a simple C++ program, which includes one safe downcast in line 2 and one unsafe downcast in line 9. In each case, we focus on how the type information of an object is initialized, propagated, and determined during the pointer analysis.

In this example, we give only the instructions related to our rules. For simplicity, each copy $\%x = \%y$ here is an

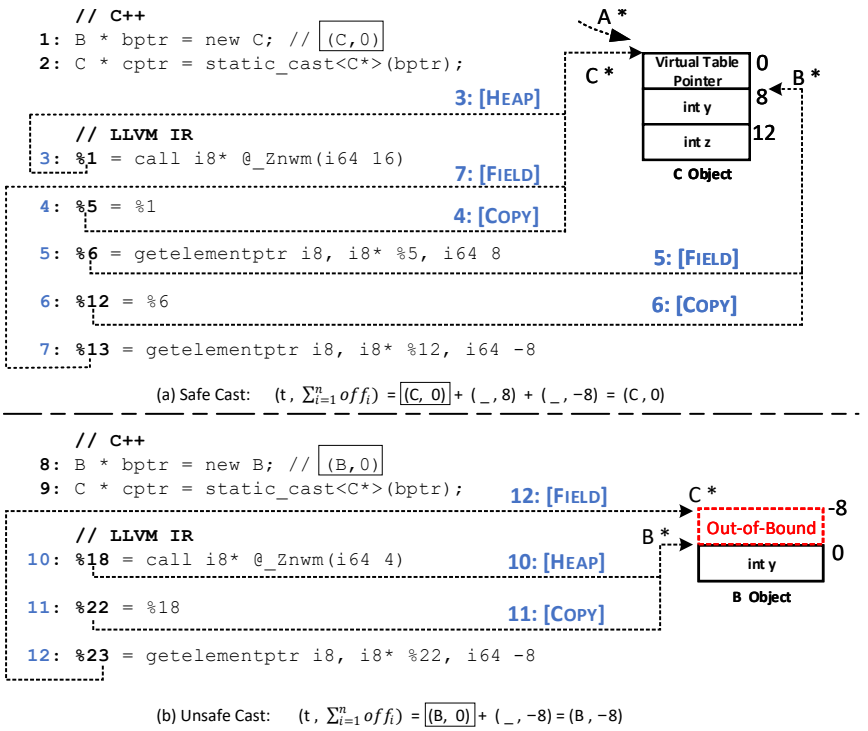

Fig. 7. An example illustrating the rules in our pointer analysis. For example, the label "3:[HEAP]" means [HEAP] is applied in line 3.

TABLE II
RULE APPLICATION FOR THE CODE IN FIGURE 7.

Rule	Inst. (Line #)	Pointer	Pointed-to Object	Type
[HEAP]	3	%1	obj_C	(C,0)
[COPY]	4	%5	obj_C	(C,0)
[FIELD]	5	%6	$obj_{C.8}$	(C,8)
[COPY]	6	%12	$obj_{C.8}$	(C,8)
[FIELD]	7	%13	obj_C	(C,0)
[HEAP]	10	%18	obj_B	(B,0)
[COPY]	11	%22	obj_B	(B,0)
[FIELD]	12	%23	$obj_{B.-8}$	(B,-8)

abstraction of several LLVM-IR instructions. The notations such as $\%x$ and $\%y$ represent virtual registers in LLVM-IR. As they act as top-level pointers in program analysis, we use them directly to represent pointers in the following discussion.

- **Safe Cast.** Let us consider the safe cast in the top part of Figure 7. We start with by issuing a points-to query $pt(\%13)$ in line 7, i.e., $pt(\texttt{cptr})$ in line 2. During the demand-driven pointer analysis, the IR instruction for allocating a heap object, denoted obj_C, in line 3 is analyzed. By applying [HEAP], we obtain $pt(\%1) = \{obj_C\}$ such that $POS(obj_C, (\texttt{C}, 0))$, highlighted by the dotted line labeled with "3:[HEAP]". Due to the copy $\%5 = \%1$ in line 4, we obtain $pt(\%5) = pt(\%1) = \{obj_C\}$, highlighted by the dotted line "4:[COPY]". The `getelementptr` instruction in line 5 performs the pointer adjustment due to upcasting. According to [FIELD], %6 points to obj_C but at its offset 8, indicated by $POS(obj_{C.8}, (\texttt{C}, 8))$, as highlighted by the dotted line labeled with "5:[FIELD]". Due to the copy $\%12 = \%6$ in line 6, %12 now also points to obj_C at its offset 8, highlighted by the dotted line "6:[COPY]". In line 7, the `getelementptr` instruction performs the pointer adjustment due to downcasting. By applying [FIELD] again, $\%13$ now points to obj_C at its beginning, i.e., the offset 0, indicated by $POS(obj_C, (\texttt{C}, 0))$. Finally, we apply [ASSIGNTYPE] to obtain $Type(obj_C) = \texttt{C}$. Now, we can conclude that `cptr` in line 2 actually points to a `C` object. The type cast is therefore safe.
- **Unsafe Cast.** Let us consider the unsafe cast in the bottom part of Figure 7. This time, we start with by issuing a points-to query $pt(\%23)$ in line 12, i.e., $pt(\texttt{cptr})$ in line 9. When applying [HEAP] to the IR instruction in line 10, which allocates a heap object, denoted obj_B, we obtain $pt(\%18) = \{obj_B\}$ such that $POS(obj_B, (\texttt{B}, 0))$ holds. Unlike the safe cast above, there is no upcasting here. By applying [COPY] to $\%22 = \%18$ in line 11, we find that %22 now also points to obj_B at its beginning. In line 12, the `getelementptr` instruction performs the pointer adjustment due to down-casting. According to [FIELD], %23 points to obj_B but at the offset -8, indicated by $POS(obj_{B.-8}, (\texttt{B}, -8))$. Finally, we apply [ASSIGNTYPE] to obtain $Type(obj_{B.-8}) = \texttt{B}$. As `cptr` in line 9 points to potentially a `B` object, TCD will issue a warning about this unsafe downcast in that line.

In Table II, we summarize the rules applied to the LLVM-IR instructions given in Figure 7, as explained above.

C. Type Confusion Bug Detector

An LLVM pass is implemented to collect all the cast expressions instrumented by our customized C++ compiler front-end. For a cast expression `static_cast<T*>(ptr)`, we will retrieve its destination type `T` and the declared type `S` of `ptr` from the metadata associated with its stub.

Let `dst = static_cast<T*>(ptr)`, with the pointer adjustment performed already (Figure 2). So `ptr` and `dst` may not point to the same location. Note that `T` will be used directly for detecting type confusion bugs (as shown below) and `S` will be used for bug-reporting purposes. As C++ classes are the main targets of type confusion attacks, we focus on detecting type confusion errors for C++ classes.

Now, a points-to query *pt*(`dst`) is issued. We handle the following two cases, depending on the types of the objects in *pt*(`dst`):

- **Unsafe Casts.** `static_cast<T*>(ptr)` is *unsafe* if $\exists\ o \in$ *pt*(`dst`), the type of o is incompatible with `T`.
- **Safe Casts.** `static_cast<T*>(ptr)` is *safe* if it is not an unsafe cast.

We can handle `reinterpret_cast` similarly. It should be noted that an unsafe cast reported by TCD may be a false positive (due to, e.g., the lack of path-sensitivity).

D. Implementation

We have implemented TCD in LLVM on top of the open-source SVF pointer analysis framework [17]. Given a type in the form of (t, off), Algorithm 1, which implements $GetType(t, \mathit{off})$ used in [AssignType] (Figure 5), returns the largest type at a designated byte offset *off* within a C++ class type t. A C++ class is represented as a recursive structure, where its elements can be a C++ class, a primary type or an array. In addition, the element type of an array can also be a C++ class, a primary type or an array.

In lines 8 – 15, the `while` loop finds an element in the class t that is closest to the specified byte offset *off*. As the class t has at least one element, the statements in lines 12 – 14 in the while body is expected to be executed at least once. If the element happens to locate at the given byte offset, then its type is returned in line 20. Otherwise, we recursively find the type being searched for in line 24. The function `getEleOffset()` in line 12 returns the offset of the specified element i within a class t and `getEleType()` in line 13 is used to retrieve its type. The functionalities of the other functions are reflected in their names.

There is another important implementation detail that is worth emphasizing, as it is critical to flow-sensitive analysis in the presence of global object initialization in C++ programs. In a C++ program (unlike in a C program), global objects must be initialized with their corresponding C++ constructors before `main()` is called. To obtain a flow-sensitive analysis that respects the original semantics in C++ programs, we need to analyze the whole program and synthesize a pseudo entry function `before.main()` to call all the initializer functions first and then invoke `main()`. The synthesized `before.main()` will guide our static analyzer to create a call graph correctly for flow-sensitive pointer analysis.

Algorithm 1: $GetType(t, \mathit{off})$ in Figure 5.

Input **:** A C++ class t and a byte offset *off*
Output**:** The largest type at the offset *off*

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

IV. Evaluation

Our evaluation demonstrates the effectiveness of TCD in detecting type confusion bugs in C++ applications by addressing the following two research questions (RQs).

- **RQ1.** Can TCD find new type confusion bugs in real-world C++ applications at a low false positive rate?
- **RQ2.** Can TCD overcome some limitations of dynamic detectors in detecting type confusion bugs?

We have evaluated TCD using Qt [32], a widely used open-source toolkit for creating GUIs and cross-platform software. We consider all its seven Qt tools (totaling 526,385 lines of C++ code), which share the same Qt base library. TCD has found five type confusion bugs, including one reported in prior work and four new ones.

To strike a balance between precision and scalability, our pointer analysis is demand-driven. The budgets for flow-sensitivity and context-sensitivity are both configured as a maximum of 10000 value-flow edges traversed per points-to query (Section III-B1). The maximum context length used for realizing context-sensitivity is set to be 3, implying that the calling context for a function is bounded by three call sites.

Our platform consists of a 3.20 GHz Intel Xeon(R) E5-1660 v4 CPU with 256 GB memory, running the Ubuntu OS. The analysis time of a program is the average of 3 runs.

A. RQ1: Bug-Finding Ability

1) Effectiveness: As shown in Table III, TCD reports a total of 39 warnings in the seven Qt tools identified as A1 – A7. After manual inspection, we found 28 true positives (TPs) and 11 false positives (FPs), achieving a low false positive rate of 28.2%. These 28 true positives represent a total of five distinct bugs identified as B1 – B5, which all reside in the Qt base library used, including one **known** bug, B1, reported by the dynamic detector HexType [6] (Table IV), and four **new** bugs, B2 – B5, that are found in this paper (Table V). Table VI provides a mapping from $\{\mathrm{B1}, \cdots, \mathrm{B5}\}$ to $\{\mathrm{A1}, \cdots, \mathrm{A7}\}$, showing all the Qt tools where a bug is exposed. These are all the type confusion errors caused by `static_cast<T*>(ptr)`, where the destination type `T` and the declared type of `ptr` are given in each case.

TABLE III
EXPERIMENTAL RESULTS FOR THE SEVEN Qt TOOLS (TP: TRUE POSITIVES; FP: FALSE POSITIVES).

APP ID	Qt tool	#TP	#FP	Analysis Time (secs)
A1	moc	2	0	171
A2	qdbuscpp2xml	4	2	4108
A3	qdbusxml2cpp	4	2	7247
A4	qlalr	7	4	4889
A5	qmake	6	1	3415
A6	rcc	1	0	109
A7	uic	4	2	6168
Total		28	11	26107

TABLE IV
ONE **KNOWN** TYPE CONFUSION BUG IN THE Qt BASE LIBRARY, REPORTED IN PRIOR WORK (HEXTYPE [6]) BUT REDISCOVERED BY TCD.

Bug ID	File Name	Function Template	Line
B1	qmap.h	Node *end()	216

TABLE V
FOUR **NEW** TYPE CONFUSION BUGS DETECTED IN THE Qt BASE LIBRARY.

Bug ID	File	Function	Line
B2	qjson.cpp	Data::compact()	91
B3	qjson.cpp	Data::compact()	110
B4	qjsonobject.cpp	QJsonObject::compact()	1236
B5	qjsonvalue.cpp	QJsonValue::detach()	688

As shown in Table IV, B1 represents a bug in a function template. It should be pointed out that function templates are widely used in C++ applications. In this particular case, the function template has been instantiated with different type parameters, resulting in different functions in LLVM-IR. As a result, this type confusion bug in the Qt base library has appeared 16 times in all the seven Qt tools.

As shown in Table V, B2 – B5 are all type confusion bugs appearing in ordinary functions. We have analyzed these four new bugs and found that they are caused by ad hoc implementations of C++ inheritance. Their common bug pattern is illustrated in Figure 8. In line 13, a memory block that is 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.

The function `root()` (defined in line 10) is called in line 14 to skip the `Header` object at the beginning of the memory block allocated by `malloc()`. If the sum operation in line 13 does not synchronize with the modification of the derived classes of `Base`, a dangerous software security vulnerability may arise. For example, software developers may add some new fields in a derived class and forget to update the sum

TABLE VI
MAPPING FIVE BUGS IN $\{\mathrm{B1}, \cdots, \mathrm{B5}\}$ TO SEVEN Qt TOOLS $\{\mathrm{A1}, \cdots, \mathrm{A7}\}$ WHERE A BUG IS DETECTED. FOR EACH BUG CAUSED AT `static_cast<T*>(ptr)`, THE DESTINATION TYPE `T` AND THE DECLARED TYPE OF `ptr` ARE GIVEN. FOR B1, `QMapNode.n`, WHERE $n \in \mathbb{Z}$, REPRESENTS AN INSTANTIATION OF CLASS TEMPLATE `QMapNode`.

Bug ID	APP ID	Declared Type	Destination Type
B1	A1	QMapNodeBase	QMapNode.148
B1	A1	QMapNodeBase	QMapNode.530
B1	A2	QMapNodeBase	QMapNode.3766
B1	A2	QMapNodeBase	QMapNode.86
B1	A3	QMapNodeBase	QMapNode.86
B1	A3	QMapNodeBase	QMapNode.3601
B1	A4	QMapNodeBase	QMapNode.36
B1	A4	QMapNodeBase	QMapNode.50
B1	A4	QMapNodeBase	QMapNode.36
B1	A4	QMapNodeBase	QMapNode.86
B1	A4	QMapNodeBase	QMapNode.3365
B1	A5	QMapNodeBase	QMapNode
B1	A5	QMapNodeBase	QMapNode.25
B1	A6	QMapNodeBase	QMapNode
B1	A7	QMapNodeBase	QMapNode.86
B1	A7	QMapNodeBase	QMapNode.4224
B2	A2/A3/A4/A5/A7	QJsonPrivate::Base	QJsonPrivate::Object
B3	A2/A3/A4/A5/A7	QJsonPrivate::Base	QJsonPrivate::Array
B4	A5	QJsonPrivate::Base	QJsonPrivate::Object
B5	A5	QJsonPrivate::Base	QJsonPrivate::Object

```
1  struct Base{
2    ...
3  };
4  struct Derived: public Base{
5    ...
6  };
7  struct Header{
8   int tab;
9   int version;
10  Base *root() { return (Base *)(this + 1); }
11 };
12 int main(){
13  Header *h = (Header *) malloc(sizeof(Header)
        + sizeof(Base) + const_sz + variable_sz);
    ...
14  Base *b = h->root();
15  static_cast<Derived*>(b);
16 }
```

Fig. 8. The bug pattern for the new type confusion errors detected in Qt.

```
============================  Report  ========================================
Bad static_cast:
    %"class.QJsonPrivate::Base"*   ====>   %"class.QJsonPrivate::Object"*
Where:
    line: 688 file: qtbase/src/corelib/json/qjsonvalue.cpp
Points-to:
    (1) line: 79 file: qtbase/src/corelib/json/qjson.cpp
    (2) line: 840 file: qtbase/src/corelib/json/qjson_p.h
    (3) line: 880 file: qtbase/src/corelib/json/qjson_p.h
================================================================================
//  qtbase/src/corelib/json/qjsonvalue.cpp
678 void QJsonValue::detach(){
683     QJsonPrivate::Data *x = d->clone(base);
687     d = x;
688     base = static_cast<QJsonPrivate::Object *>(d->header->root());
780 }
//  qtbase/src/corelib/json/qjson.cpp
58  void Data::compact(){
77      int size = sizeof(Base) + reserve + base->length*sizeof(offset);
78      int alloc = sizeof(Header) + size;
79      Header *h = (Header *) malloc(alloc);
82      Base *b = h->root();
130  }
//  qtbase/src/corelib/json/qjson_p.h
612 class Object : public Base{
622 };
761 class Header {
765     Base *root() { return (Base *)(this + 1); }
766 };
834 inline Data(int reserved, QJsonValue::Type valueType){
839     alloc = sizeof(Header) + sizeof(Base) + reserved + sizeof(offset);
840     header = (Header *)malloc(alloc);
849 }
865 Data *clone(Base *b, int reserve = 0){
867     int size = sizeof(Header) + b->size;
868     if (b == header->root() && ref.load() == 1 && alloc >= size + reserve)
869         return this;
871     if (reserve) {
872         if (reserve < 128)
873             reserve = 128;
874         size = qMax(size + reserve, qMin(size *2, (int)Value::MaxSize));
879     }
880     char *raw = (char *)malloc(size);
883     Header *h = (Header *)raw;
889 }
```

Fig. 9. A new type confusion bug, B5 (Table V), detected in Qt.

operation in line 13, so that the size of this derived class is larger than the memory block allocated by `malloc()`. In this situation, a downcast in line 15 from `Base` to the derived class will lead to out-of-bound memory access.

In Figure 9, we examine B5 given in Table V. In the top part, our bug report shows that B5 occurs in line 688, `static_cast < QJsonPrivate::Object *> (d->header->root())`, together with the lines where the three objects pointed to by `d->header->root()` reside. In the bottom part, the related C++ source files are listed briefly, showing that TCD is able to detect type confusion bugs across different files precisely and inter-procedurally. By performing our type-aware pointer analysis, we find that `d->header` in line 688 (`qjsonvalue.cpp`) points to the three `Header` objects allocated in line 79 (`qjson.cpp`), 840 (`qjson_s.h`) and 880 (`q_jsons.h`) respectively. In line 688, `d->header->root()` skips these three `Header` objects and points to their `Base` objects following them in memory layout. Then a dangerous downcast happens in line 688, where a `Base` object is downcast to a `QJsonPrivate::Object` object. The ad hoc implementations of C++ inheritance can be seen in lines 77 – 79 (`qjson.cpp`), 839 – 840 (`qjson_s.h`) and 867 – 880 (`q_jsons.h`), where memory blocks larger than the size of a `Base` object are allocated. As highlighted in Figure 8, this bug pattern can lead to out-of-bound memory access. As shown in Figure 9, a TCD warning can clearly pinpoint where such potential security vulnerabilities are in large C++ projects.

2) Efficiency: TCD spends a total of 26,107 seconds, i.e., 7.3 hours on analyzing the seven Qt tools totaling 526,385 lines of C++ code. This is not unreasonable.

B. RQ2: TCD vs. Dynamic Detectors

Of the five type confusion bugs listed in Tables IV and V, HexType [6], a dynamic detector, can only detect B1 in Table IV but not B2 – B5 in Table V. This demonstrates TCD's ability to find new bugs in large C++ projects that can be difficult to reach by dynamic tools (as motivated in Figure 2).

Given a C++ program, dynamic detectors such as HexType perform instrumentation at compile time. In order to detect type confusion bugs at testing stage, an instrumented C++ program will be run repeatedly with different inputs so that more bugs can be triggered. While much work has been done in various sorts of testing techniques [23], [33]–[38], dynamic detectors still suffer from low code coverage. As motivated in Figure 2, `hard_to_satisfy()`, which represents a complex condition that is very hard to satisfy, represents still an obstacle to dynamic analysis. However, TCD can often find potentially bugs despite its being path-insensitive.

Another rarely-discussed obstacle is container coverage, which may require every element of a container (e.g., an array) to be tested in order to find a particular bug. Consider a simple program in Figure 10. In line 7, a fuzzer [23], [38] can easily generate a random value `x` to satisfy `x >= 0 && x < N`. However, the type confusion bug in line 8 can only be triggered when `x = 2019` exactly. In our evaluation, even for such a simple program, this bug cannot be exposed in 72 hours with AFL [23], one of the state-of-the-art fuzzers.

```
1  #define  N  (1024*64)
2  Base *ptr[N];
3  for(int i = 0; i < N; i++){
4     ptr[i] = new Derived;
5  }
6  ptr[2019] = new Base;
7  if(x >= 0 && x < N){
8     static_cast<Derived*>(ptr[x]);
9  }
```

Fig. 10. Container-coverage-related obstacle to dynamic analysis.

In contrast, TCD is a static detector, which is currently path-insensitive. In Figure 2, TCD will ignore the `hard_to_satisfy()` condition in line 15 while still being able to detect the type confusion error in line 18. In Figure 10, TCD will ignore the condition in line 7 by analyzing `ptr` conservatively as a pointer rather than an array of pointers, so that `ptr` points to all the objects pointed to individually by its elements. Under this abstraction, `ptr` may point to either a `Derived` object (line 4) or a `Base` object (line 6). Thus, TCD can also expose the type confusion bug in line 8.

By being path-insensitive, TCD can improve code coverage but may suffer from unavoidable false positives. In a real program containing lines 7 – 8 in Figure 10, if `x` can never be 2019 under any program input, then the type confusion error in line 8 reported by TCD will be a false positive. In addition, as a static detector, TCD is expected to consume tens of gigabyte memory space in analyzing large C++ programs. Compared with dynamic detectors, precise static solutions may not scale to tens of millions lines of code [39].

V. Related Work

We review the work relevant to TCD, by focusing on dynamic techniques for detecting type confusion bugs and control flow integrity (CFI) techniques for enforcing CFI.

Dynamic Type Confusion Detectors. Undefined Behavior Sanitizer (UBSan) [9] relies on the type information stored in virtual tables to detect whether a type cast is safe or not and is thus limited to protecting polymorphic classes only. CAVER [10] instruments C++ programs and maintains metadata for both polymorphic and non-polymorphic classes. Since red-black trees are used to store metadata for stack and global objects at `O(log n)`, it can incur high instrumentation overhead if most of the allocated objects are on the stack. TypeSan [12] relies on a compact memory shadowing mechanism to trace all objects in a uniform way, such that the overhead of tracing objects is reduced. But it may conflict with address space layout randomization [40]. The limitation on high instrumentation overhead has subsequently been addressed by HexType [6], Bitype [14], and CastSan [15]. Finally, EffeciveSan [13] can detect not only type confusion bugs but also memory-related bugs.

These dynamic detectors can find type confusion bugs precisely, but suffer from low code coverage and high instrumentation overhead. In contrast, TCD can reveal potential type confusion bugs across the entire program statically, but at the expense of introducing false positives.

Control Flow Integrity. Type confusion bugs may lead to control flow attacks, which can be mitigated by control flow integrity [41]–[44]. CFI states that program execution must follow the control flow graph (CFG) generated at compile time. Two main challenges remain: how to make control-flow targets in the CFG precise and how to make dynamic checks at these control-flow targets efficient. In general, CFI defense mechanisms only protect code pointers. However, type confusion bugs can be exploited to corrupt not only code pointers but also other sensitive data.

VI. Conclusions

We have introduced a new static detector, TCD, in LLVM for finding type confusion bugs in C++ programs, based on a type-casting-preserving C++ compiler front-end and a type-aware pointer analysis. TCD has found four new type confusion bugs in Qt [32], which have evaded detection of previous (dynamic) approaches with a low false positive rate.

In one future work, we plan to extend TCD by considering path-sensitivity to reduce the false positives reported. In another future work, we plan to combine static and dynamic analyses to obtain the best of both worlds.

Acknowledgements

We thank all the reviewers for their valuable inputs. This work has been supported by Australian Research Council Grants (DP170103956 and DP180104069).

REFERENCES

[1] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 245–258.

[2] ——, "Cets: compiler enforced temporal safety for c," in *ACM Sigplan Notices*, vol. 45, no. 8, 2010, pp. 31–40.

[3] D. Ye, Y. Su, Y. Sui, and J. Xue, "Wpbound: Enforcing spatial memory safety efficiently at runtime with weakest preconditions," in *Proceedings of the 25th International Symposium on Software Reliability Engineering*, 2014, pp. 88–99.

[4] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of c," in *Proceedings of the USENIX Annual Technical Conference*, 2002, pp. 275–288.

[5] H. Yan, Y. Sui, S. Chen, and J. Xue, "Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 327–337.

[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 24th ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2373–2387.

[7] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th Conference on USENIX Security Symposium*, 1998, pp. 1–16.

[8] "Common vulnerabilities and exposures," https://cve.mitre.org/, accessed May 18, 2019.

[9] "Undefined behavior sanitizer," https://www.chromium.org/developers/testing/undefinedbehaviorsanitizer, accessed May 18, 2019.

[10] B. Lee, C. Song, T. Kim, and W. Lee, "Type casting verification: Stopping an emerging attack vector." in *Proceedings of the 24th USENIX Conference on Security Symposium*, 2015, pp. 81–96.

[11] S. Kell, "Dynamically diagnosing type errors in unsafe code," in *Proceedings of the 31st ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016, pp. 800–819.

[12] 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 23rd ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 517–528.

[13] 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*, 2018, pp. 181–195.

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

[15] P. Muntean, S. Wuerl, J. Grossklags, and C. Eckert, "Castsan: Efficient detection of polymorphic C++ object type confusions with LLVM," in *Proceedings of the 23rd European Symposium on Research in Computer Security*, 2018, pp. 3–25.

[16] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *Proceedings of the 37th International Conference on Software Engineering*, 2015, pp. 598–608.

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

[18] T. Tan, Y. Li, and J. Xue, "Efficient and precise points-to analysis: Modeling the heap by merging equivalent automata," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 278–291.

[19] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the 2nd International Symposium on Code Generation and Optimization*, 2004, p. 75.

[20] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015, pp. 745–762.

[21] C. Zhang, D. Song, S. A. Carr, M. Payer, T. Li, Y. Ding, and C. Song, "Vtrust: Regaining trust on virtual calls." in *Network and Distributed System Security Symposium*, 2016.

[22] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song, "Vtint: Protecting virtual function tables' integrity." in *Network and Distributed System Security Symposium*, 2015.

[23] M. Zalewski, "American fuzzy lop (AFL) fuzzer," http://lcamtuf.coredump.cx/afl/, accessed May 18, 2019.

[24] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: fuzzing by program transformation," in *Proceedings of the 39th IEEE Symposium on Security and Privacy*, 2018, pp. 697–710.

[25] D. J. Pearce, P. H. Kelly, and C. Hankin, "Efficient field-sensitive pointer analysis of c," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 1, 2007.

[26] Y. Sui, P. Di, and J. Xue, "Sparse flow-sensitive pointer analysis for multithreaded programs," in *Proceedings of the 14th International Symposium on Code Generation and Optimization*, 2016, pp. 160–170.

[27] R. P. Wilson and M. S. Lam, "Efficient context-sensitive pointer analysis for c programs," in *Proceedings of the 16th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1995, pp. 1–12.

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

[29] 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 25th ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2095–2108.

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

[31] T. Ravitch, "Whole program LLVM," https://github.com/travitch/whole-program-llvm, accessed May 18, 2019.

[32] "QT," https://www.qt.io/, accessed May 18, 2019.

[33] Y. Zhang, Z. Chen, J. Wang, W. Dong, and Z. Liu, "Regular property guided dynamic symbolic execution," in *Proceedings of the 37th International Conference on Software Engineering*, 2015, pp. 643–653.

[34] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008, pp. 209–224.

[35] H. Wang, T. Liu, X. Guan, C. Shen, Q. Zheng, and Z. Yang, "Dependence guided symbolic execution," *IEEE Transactions on Software Engineering*, vol. 43, no. 3, pp. 252–271, 2017.

[36] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *Network and Distributed System Security Symposium*, 2008.

[37] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 213–223.

[38] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[39] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang, "Pex: A permission check analysis framework for linux kernel," in *Proceedings of the 28th USENIX Conference on Security Symposium*, 2019, pp. 1205–1220.

[40] "Address space layout randomization," https://pax.grsecurity.net/docs/aslr.txt, accessed May 18, 2019.

[41] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM SIGSAC Conference on Computer and Communications Security*, 2005, pp. 340–353.

[42] 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*, 2014, pp. 941–955.

[43] B. Niu and G. Tan, "Per-input control-flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 914–926.

[44] X. Fan, Y. Sui, X. Liao, and J. Xue, "Boosting the precision of virtual call integrity protection with partial pointer analysis for c++," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 329–340.