Accelerating Type Confusion Detection with Pointer Analysis

Xiaokang Fan, Zeyu Xia, Sifan Long, Chun Huang, and Canqun Yang

***Abstract*—C++ is widely used in performance critical applications. Due to the lack of type safety, programs written in C++ are vulnerable to memory corruption errors. Type confusion bug is an emerging attack vector. Several type confusion detection tools have been proposed to detect unsafe cast instructions. However, these tools suffer from the problem of high runtime overhead. The perfromance problem has prevented the deployment of type confusion detection tools into production software.**

This paper proposes a new method to mitigate the performance problem. The novelty of this paper is that we use pointer analysis to identify redundant instrumentations. By removing the identified redundant instrumentations, we can significantly reduce the runtime overhead incurred by type confusion detection tools. We have applied our pointer analysis to Typesan - an-open source type confusion detection tool. Statically, 68.67% of instrumentations for tracing types of objects and 57.84% of instrumentations for verifying downcast instructions can be removed on average. Dynamically, the average (maximum) runtime overhead can be reduced from 40.10% (89.71%) to 12.22% (24.90%).

***Index Terms*—C++, type confusion, pointer analysis**

I. Introduction

C++ has been widely used in large software systems due to its abstraction and high performance. For example, major web browsers like Chrome and Firefox, and language runtimes like Oracle's Java Virtual machine are mainly implemented in C++.

Type casting, which allows a program to convert objects from one pointer type to another, is a very important feature to achieve polymorphism. C++ offers both static_cast and dynamic_cast. A static_cast is done at compile time. The compiler is responsible for checking whether the cast is valid. While a dynamic_cast is done at runtime. The validity of the cast is verified at runtime using RTTI (Runtime Type Information), which incurs expensive runtime overhead (e.g., 90 times slower than a static_cast) [1]. Thus, dynamic_cast is prohibited in performance critical applications like Chrome, Firefox.

Without verification, a static_cast may lead to serious errors that compromise the program. For instance, a program may cast a base object into a derived object. If the derived class lacks some data field, a following field access will lead to a memory corruption or memory leak. If the derived class lacks some virtual functions, a following virtual function call will lead to some unwanted functions called, which may result in a control flow attack. Figure 1(lines 4-6) shows an example of memory corruption caused by an unsafe downcast. Pointer `b1` of type `Base*` points to an object of type `Base` (line 4). Then it is casted into a pointer `d1` of type `Derived*` (line 5). Pointer `d1` is used to access a field `f2` outside the original object (line 6). Resulting in a memory corruption.

```
 1: class Base { public: double f1; }
 2: class Derived: public Base { public: double f2; }
 3:
 4: Base *b1 = new Base;
 +: type(*b1) = Base;
 +: assert(type(*b1)<:Derived) ✗
 5: Derived *d1 = static_cast<Derived*>(b1);
 6: d1->f2 ...; //memory corruption
 7:
 8: Base *b2 = new Derived;
 +: type(*b2) = Derived;
 +: assert(type(*b2)<:Derived) ✓
 9: Derived *d2 = static_cast<Derived*>(b2);
10: d2->f2 ...; //benign memory access
```

Fig. 1. An example showing how Typesan [2] detects an unsafe downcast instruction (line 5) and prevents a memory corruption (line 6). However, it introduces a redundant check before a benign downcast instruction (line 9), which will cause redundant runtime overhead. "+" represents instrumentations of Typesan. "<:" represents a subclass relation.

An unsafe downcast like in the previous example is called a *type confusion* or a *bad type cast*. Type confusions have been exploited in bugs found in a wide range of applications [2].

Type confusion has become a new attack vector and attracted a lot research effort [1], [2], [3]. Two types of methods have been proposed to address this problem: (1) methods that make use of vtable pointers embedded in C++ objects to identify the type of an object [3], and (2) methods with the help of disjoint metadata [1], [2].

The advantage of vtable based methods is that vtable contains the type information, so these methods avoid the overhead of tracing the types of C++ objects. However, as only polymorphic classes have vtables, the disadvantage is that these methods only support polymorphic classes. So these methods can only cover a limited number of downcast instructions.

With the help of disjoint metadata, methods like Caver [1] and Typesan [2] can support both polymorphic and non-polymorphic classes. So they can cover a much larger number of downcast instructions. Two kinds of instrumentations are required: (1) instrumentation after the creation of every C++ object to trace its type, and (2) instrumentation before every downcast instruction to verify whether the downcast is safe or not.

Figure 1 (lines 4-6) shows an example of how a type confusion bug can be detected by a disjoint metadata based method. An object of type `Base` (created at line 4) is casted into type `Derived` (line 5). This unsafe downcast will cause a memory corruption when a data field (`f2`) outside the original object is accessed. With the help of the

Manuscript received January 12, 2020; revised August 11, 2020.

X. Fan, Z. Xia, S. Long, C. Huang, and C. Yang are with the School of Computer, National University of Defense Technology, Changsha, 410073, Hunan, P.R.China. Email: fanxiaokang@nudt.edu.cn, xzyschumacher@hotmail.com, 164712110@csu.edu.cn chunhuang@nudt.edu.cn, canqun@nudt.edu.cn,

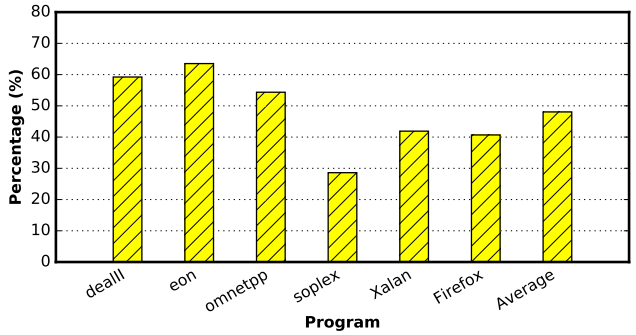

Fig. 2. Percentage of traced objects by Typesan that will not be used in any downcast instructions.

instrumentations to trace the type of the object (after line 4) and verify the downcast (before line 5). This type confusion bug can be detected.

Although disjoint metadata based methods can cover most downcast instructions. The high runtime overhead has prevented them from being deployed in production software. For example, Typesan incurs nearly 100% performance overhead on Firefox. The high runtime overhead are caused by large number of instrumentations for all C++ objects and every downcast instruction.

However, a large number of these instrumentations may be redundant, as a traced object may never be used in any downcast instructions, and a downcast instruction may never cast a base object into a derived object. We call a downcast instruction that will never cast a base object into a derived object a *safe downcast instruction*. Otherwise, a downcast instruction is referred as an *unsafe downcast instruction*.

Figure 1 (lines 8-10) shows an example of redundant instrumentations. The downcast instruction at line 9 is safe as the object being casted is of type `Derived`. So the instrumentations for object tracing and runtime check between line 8 and line 9 are redundant.

Figure 2 shows the percentage of objects that will never be used in any downcast instructions based on our pointer analysis. As the figure shows, nearly 65% of the objects traced by Typesan will never be used in any downcast instructions in the program eon. Nearly 50% of objects traced will never be used in any downcast instructions, on average.

A precise inter-procedural pointer analysis can identify the objects that may be pointed to by each pointer at runtime. With the help of pointer analysis, more safe downcast instructions and more objects that does not need tracing can be identified. By removing redundant instrumentations for these safe downcast instructions and objects, type confusion detection tools can be made deployable to production software.

However, developing a precise inter-procedural pointer analysis for identifying redundant instrumentations in C++ programs is quite challenging. Type information is not included in traditional whole program pointer analyses [4], [5], [6], [7]. Without type information for each C++ object, it would be impossible to identify redundant instrumentations.

To overcome the above challenges, we propose a pointer analysis for C++ programs with type information for all C++ objects included. With the help of the type information, we can identify safe downcast instructions, and objects that will never be used in any unsafe downcast instructions. Instrumentations for the above mentioned safe downcast instructions and objects can be removed.

Contributions In summary, this paper makes the following contributions:

- We propose a pointer analysis for C++ programs. The type information for every C++ object can be resolved using our pointer analysis.
- We present a prototype implementation of our pointer analysis in LLVM-3.9.0.
- We have applied our pointer analysis to Typesan [2], a typical type confusion detection tool. We conduct a thorough evaluation using SPEC 2000/2006 C++ benchmarks, and one open-source web browser: Mozilla's Firefox. Statically, we can remove 68.67% of instrumentations for tracing types of objects and 57.84% of instrumentations for verifying downcast instructions on average. Dynamically, the average (maximum) runtime overhead can be reduced from 40.10% (89.71%) to 12.22% (24.90%), making type confusion detection tools deployable to production software.

The rest of the paper is organized as follows: Section II provides some background knowledge on type casting of C++ and defenses against type confusion. Our threat model is defined in Section III. Section IV describes our pointer analysis in detail. And Section V explains how to remove redundant instrumentations based on the results of pointer analysis. We provide a thorough evaluation in Section VI. Related work are discussed in Section VII and Section VIII concludes the paper.

II. Background

In this section, we will first briefly explain type casting in C++ and the potential vulnerabilities caused by a type confusion bug. Followed by defense techniques against type confusion bugs.

A. Type Casting

C++ allows reinterpretation of memory through type casting to achieve polymorphism. An object declared of one class type can be casted into its base class, its derived class, or even another class outside the class hierarchy. As an example, line 5 in Figure 1 shows a cast from a base object to a derived type.

Based on the relation between the source type and the target type, a cast instruction can be divided into two classes:

- An *upcast* converts an object of a derived class into a base class. Since an object of a derived class always inherits all the data fields from the base class, the resulting pointer can not access memory outside the original object. An upcast is always safe.
- A *downcast* converts an object of a base class into a derived class. As a base object may lack some of the data fields or some virtual functions in the derived object. This may lead to a memory leak or a memory corruption when the resulting pointer is used to access memory outside the original base object.

C++ offers two kinds of type casts: dynamic_cast, and static_cast. dynamic_cast verifies the safety of type conversion at runtime using RTTI (Run-Time Type Information).

A runtime verification performed by dynamic_cast incurs expensive runtime performance overhead (e.g., over 90 times slower that static_cast) [1]. Thus, dynamic_cast is prohibited in performance critical applications.

static_cast relies on the compiler to verify the cast at compile time. The compiler only checks whether the source and the target type in the cast instruction are subtype or supertype relations.

reinterpret_cast allows a programmer to explicitly break the inheritance relation and interpret a memory block into a different type. It is out of the scope of this paper.

Without the verification for safety of the type conversion, static_cast can become a security vulnerability that could be exploited by attackers. Example exploitable vulnerabilities include Adobe Flash (CVE-2015-3077), Microsoft Internet Explorer (CVE-2015-6184), PHP (CVE-2016-3185), and Google Chrome (CVE-2013-0912).

B. Defense Techniques Against Type Confusion Bugs

Several recent techniques have been proposed to address the type confusion problem. Depending on whether disjoint metadata is required, type confusion detection techniques can be classified into two types: (1) methods that leverage vtable pointers embedded in C++ objects to identify an object's type, and (2) methods relying on disjoint metadata to trace the type of C++ objects.

Clang's UBSan [3] leverages the RTTI (Run-Time Type Information) stored in vtables to identify a C++ object's type. It can detect unsafe downcast instructions with the wrong dynamic type. Without the need for disjoint metadata, vtable based methods avoid the overhead for tracing types of C++ objects. However, there are a few limitations for these methods. First, as the RTTI is used to identify an object's type, it can not be disabled during compilation. Second, since only polymorphic classes have vtables, unsafe downcast instructions with non-polymorphic class objects can not be detected.

Caver [1] and Typesan [2] are examples of disjoint metadata based methods. With the help of disjoint metadata, objects of both polymorphic class and non-polymorphic class can be traced. As a result, these methods can significantly improve the coverage of downcast instructions. Two kinds of instrumentations are required: (1) every C++ object allocation site is instrumented with code to record the type of the object, and (2) every static downcast instruction is instrumented with an explicit runtime check to verify whether the downcast is safe. Caver [1] is the first tool which leverages disjoint metadata for type confusion detection. It uses red-black trees to track objects and a direct mapping for heap objects. Typesan [2] improves object allocation coverage by including C-style object allocation. It also uses a more efficient system to manage metadata. As a result, it can improve type confusion detection coverage and reduce runtime overhead compared with Caver. However, as every C++ object allocation site and every downcast instruction are instrumented, these methods suffer from the problem of high runtime overhead, which prevents them from being deployed into production software.

TABLE I
ANALYSIS DOMAINS.

$\mathtt{f}$	$\in$	$\mathcal{F}$	Program functions
$\mathtt{c}, \mathtt{fld}$	$\in$	$\mathcal{C}$	Constants
$\mathtt{t}$	$\in$	$\mathcal{T}$	Types
$\mathtt{p}, \mathtt{q}, \mathtt{ret}$	$\in$	$\mathcal{P}$	Top-level Pointers
$\mathtt{a}, \mathtt{a_f}, \mathtt{a.fld}, \mathtt{a[i]}$	$\in$	$\mathcal{A}$	Allocation-based objects
$\mathtt{v}$	$\in$	$\mathcal{V} = \mathcal{P} \cup \mathcal{A}$	Program variables

TABLE II
LLVM IR.

Prog	::=	$\overline{\mathtt{M}}$	PROGRAM
M	::=	$\overline{\mathtt{m}}$	MODULE
m	::=	$\mathtt{g} \mid \mathtt{f}(\mathtt{p_1}, \ldots, \mathtt{p_n})\{\ \overline{\mathtt{inst}};\ \}$	GLOBAL|FUNCTION
inst	::=	$\mathtt{p} = \&\mathtt{a}$	ADDRESSOF
		$\mathtt{p} = *\mathtt{q}$	LOAD
		$*\mathtt{p} = \mathtt{q}$	STORE
		$\mathtt{p} = (\mathtt{t})\ \mathtt{q}$	CAST
		$\mathtt{p_3} = \mathtt{phi}(\mathtt{p_1}, \mathtt{p_2})$	PHI
		$\mathtt{p} = \&(\mathtt{q} \rightarrow \mathtt{fld})$	FIELD
		$\mathtt{p} = \&\mathtt{q}[\mathtt{i}]$	ARRAY
		$\mathtt{ret} = \mathtt{fp}(\mathtt{p_1}, \ldots, \mathtt{p_n})$	CALL
		$\mathtt{return_f}\ \mathtt{q}$	RETURN

III. THREAT MODEL

We follow the threat model in previous type confusion detection works Caver [1] and TypeSan [2]. Our approach exclusively focuses on detecting type confusion errors. Other vulnerabilities such as integer overflow and memory safety are out of scope and we assume that orthogonal defense mechanisms for these vulnerabilities are deployed. The approach does not rely on information hiding, so we assume the attackers can read arbitrary memory. However, we assume that the attackers can not perform arbitrary writes.

IV. STATIC ANALYSIS

In this section, we will describe our pointer analysis for analyzing C++ programs. The program representation and LLVM IR will be first described as a basic knowledge. Then we will explain the inference rules in detail.

A. Analysis domain and LLVM IR

We use LLVM's partial SSA form to represent a program. It has been adopted by a lot of previous work on pointer analysis [4], [5], [8], as it avoids unnecessary propagation of pointer information along the CFG and enables efficient analysis.

a) Analysis domain: Our analysis domains are presented in Table I. Set $\mathcal{V}$ represents all the variables in a program. It is made up of two subsets: the set of all *top-level pointers* $\mathcal{P}$ and the set of all possible targets $\mathcal{A}$. The set $\mathcal{A}$ stands for all allocation based objects, which includes function objects $\mathtt{a_f}$, subobject $\mathtt{a.fld}$ due to field access and subobject $\mathtt{a[i]}$ due to array access. Each object in $\mathcal{A}$ corresponds to a memory location. Finally, $\mathcal{F}$ stands for all the functions. While $\mathcal{C}$ and $\mathcal{T}$ represent constants and types in the programs, respectively.

$$\texttt{[ADDRESSOF]}\frac{\texttt{p} = \&\texttt{a} \quad t = T(\texttt{p})}{\{\texttt{a}\} \subseteq pt(\texttt{p}) \quad \{\widetilde{t}\} \subseteq ts(\texttt{a})}$$

$$\texttt{[PHI]}\frac{\texttt{p}_3 = \texttt{phi}(\texttt{p}_1,\ \texttt{p}_2)}{pt(\texttt{p}_1) \subseteq pt(\texttt{p}_3) \quad pt(\texttt{p}_2) \subseteq pt(\texttt{p}_3)}$$

$$\texttt{[LOAD]}\frac{\texttt{p} = *\texttt{q} \quad \texttt{a} \in pt(\texttt{q}) \quad \texttt{a} \in \mathcal{A}}{pt(\texttt{a}) \subseteq pt(\texttt{p})}$$

$$\texttt{[STORE]}\frac{*\texttt{p} = \texttt{q} \quad \texttt{a} \in pt(\texttt{p}) \quad \texttt{a} \in \mathcal{A}}{pt(\texttt{q}) \subseteq pt(\texttt{a})}$$

$$\texttt{[FIELD]}\frac{\texttt{p} = \&(\texttt{q} \rightarrow \texttt{fld}) \ \ \texttt{a} \in pt(\texttt{q}) \ \ \texttt{a} \in \mathcal{A} \ \ t = T(p)}{\{\texttt{a.fld}\} \subseteq pt(\texttt{p}) \quad \{\widetilde{t}\} \subseteq ts(\texttt{a.fld})}$$

$$\texttt{[ARRAY]}\frac{\texttt{p} = \&\texttt{q[i]} \quad \texttt{a} \in pt(\texttt{q}) \quad \texttt{a} \in \mathcal{A} \quad t = T(\texttt{p})}{\{\texttt{a[i]}\} \subseteq pt(\texttt{p}) \quad \{\widetilde{t}\} \subseteq ts(\texttt{a[i]})}$$

$$\texttt{[FUNCTION]}\frac{\texttt{f}(\texttt{p}_1, ..., \texttt{p}_n) \quad t = T(\texttt{f})}{\{\texttt{a}_\texttt{f}\} \subseteq pt(\texttt{f}) \quad \{\widetilde{t}\} \subseteq ts(\texttt{a}_\texttt{f})}$$

$$\texttt{[CALL-EDGE]}\frac{\texttt{ret} = \texttt{fp}(\texttt{p}_1, ..., \texttt{p}_n) \rightsquigarrow \texttt{f}(\texttt{q}_1, ..., \texttt{q}_n) \quad \texttt{return}_\texttt{f}\ \texttt{q}}{\forall \texttt{i} \in \{1, ..., \texttt{n}\} : pt(\texttt{p}_\texttt{i}) \subseteq pt(\texttt{q}_\texttt{i}) \quad pt(\texttt{q}) \subseteq pt(\texttt{ret})}$$

$$\texttt{[CAST-U]}\frac{\texttt{p} = (t)\ \texttt{q} \quad \texttt{a} \in pt(\texttt{q}) \quad \texttt{a} \in \mathcal{A} \quad t' = T(\texttt{q}) \quad \widetilde{t} \not<: \widetilde{t'}\ and\ \widetilde{t'} \not<: \widetilde{t}}{\{\texttt{a}\} \subseteq pt(\texttt{p}) \quad \{\widetilde{t}\} \subseteq ts(\texttt{a})}$$

$$\texttt{[CAST-I]}\frac{\texttt{p} = (t)\ \texttt{q} \quad \texttt{a} \in pt(\texttt{q}) \quad \texttt{a} \in \mathcal{A} \quad t' = T(\texttt{q}) \quad \widetilde{t} <: \widetilde{t'}\ or\ \widetilde{t'} <: \widetilde{t}}{\{\texttt{a}\} \subseteq pt(\texttt{p})}$$

Fig. 3. Inference rules for pointer analysis. "<:" represents a subclass relation. $\widetilde{T}$ stands for the type of the object pointed to by a pointer of type T. $ts(\texttt{a})$ denotes the types an allocation-based object a may have. $pt(\texttt{p})$ represents the points-to set of pointer p.

b) LLVM IR: LLVM IR defines a comprehensive instruction set. Table II presents a subset of LLVM instructions that are relevant to our pointer analysis.

A program is made up of a set of modules M. Each module corresponds to a source file (compiled to a LLVM bitcode file using *clang* with option *-flto*). A module contains a set of global variables and function definitions. Each function definition consists of a set of instructions.

An ADDRESSOF instruction models an allocation site for both stack and heap objects. Read and write operations of address-taken variables are modeled with LOAD and STORE instructions, respectively. CAST represents a bitcast instruction in LLVM IR. And PHI models the LLVM instruction implementing the φ node in the SSA graph. FIELD models field access instructions. Array accesses are modeled with ARRAY. Call instructions are modeled with CALL. And finally, a return instruction in function f is modeled as $\texttt{return}_\texttt{f}$ q.

B. Inference rules

Figure 3 gives the inference rules. Before diving into the rules, we first explain three notations used in the rules:

1) $\widetilde{T}$ stands for the type of the object pointed to by a pointer of type T. For example, consider a pointer declaration A * p, then $T(\texttt{p}) = \texttt{A}*$ and $\widetilde{T(\texttt{p})} = \texttt{A}$.
2) $ts(\texttt{a})$ includes the types an allocation-based object a may have due to type casting.
3) $pt(\texttt{p})$ represents the points-to set of pointer p. It includes all the allocation-based object that pointer p may point to.

Below we explain our inference rules in detail.

[ADDRESSOF] handles address taken instructions. The type of an abstract object $\texttt{a} \in \mathcal{A}$ is read from its allocation site. Besides putting object a into the points-to set of pointer p, the pointee type ($\widetilde{T(p)}$) of p will be put into the type set of object a. [PHI] propagates the points-to sets between pointers. [LOAD] and [STORE] handle loads and stores instructions, respectively.

The two rules [ARRAY] and [FIELD] realize field- and array-senstivity. For an instruction $\texttt{p} = \&(\texttt{q} \rightarrow \texttt{fld})$ or $\texttt{p} = \&\texttt{q[i]}$, where q points to a base object $\texttt{a} \in \mathcal{A}$, a subobject a.fld or a[i] will be put into the points-to set of p. The pointee type ($\widetilde{T(p)}$) of p will also be put into the type set of the subobject. In LLVM IR, a temporary pointer will be introduced to help handle filed and array access instructions. A high-level statement p->f=q will be decomposed into tmp = &(p->f) and *tmp = q. Similarly, p[i]=q will be decomposed into tmp = &p[i] and *tmp = q.

[FUNCTION] creates a function object f whose address is taken by pointer p for making possible indirect calls. While [CALL-EDGE] handles function calls by passing arguments into and receiving return values from a callee.

Cast instructions can be classified into three kinds: (1) cast instructions between unrelated types (e.g. cast from float to double), (2) upcast instructions, which cast from a derived class type to a base class type, and (3) downcast instructions, on the contrary to upcast instructions, cast from a base class type to a derived class type.

[CAST-U] handles cast instructions between unrelated types, while [CAST-I] handles cast instructions between inherited types. Besides propagating the points-to information, [CAST-U] adds the pointee type of the target pointer to the typeset of abstract object $\texttt{a} \in pt(\texttt{q})$.

$$\texttt{safe}(\texttt{p} = (t)\texttt{q}) = \begin{cases} \texttt{false} \mid \exists \texttt{a} \in pt(\texttt{q}) \wedge t' \in ts(\texttt{a}) \wedge \widetilde{t} <: t' \\ \texttt{true} \ \mid \text{Otherwise} \end{cases}$$

Fig. 4. Rule for determining whether a downcast instruction is safe.

V. REMOVE REDUNDANT INSTRUMENTATIONS

The runtime overhead of a type confusion detection tool comes from two places: (1) the instrumentation for runtime check, and (2) the instrumentation for tracing the types of objects. In this section, we will describe how to remove redundant instrumentation for both runtime check and object tracing based on the results of pointer analysis.

A. Runtime check

A downcast instruction is potentially unsafe only if it may convert an object of a base class type into a derived class type. Figure 4 gives the rule for determining whether a downcast instruction is safe based on the results of pointer analysis. For a given downcast instruction, if an allocation

based object a in the points-to set of the source pointer q has a type t' that is a base of the target type $\widetilde{t}$, then we will mark this downcast instruction as unsafe. Otherwise, this downcast instruction will be regarded as safe. Runtime checks for safe downcast instructions can be removed to reduce runtime overhead.

B. Object tracing

After identifying all safe downcast instructions, we also need to identify objects that will never appear in any unsafe downcast instructions and remove the code for tracing the types of these objects.

Algorithm 1: Algorithm for collecting objects that may appear in a unsafe downcast instruction

Procedure COLLECTOBJS
begin
1 Let C be the set of unsafe downcast instructions;
2 Let O be the set of objects that need tracing;
3 **foreach** p = *(t)*q $\in C$ **do**
4 **foreach** a $\in$ pt(q) **do**
5 put a into O

First, we collect all the objects that may appear in unsafe downcast instructions, denoted as O. We only need to trace the types of objects in the set O. The instrumented code for tracing types of objects outside set O will be removed. Algorithm 1 describes how to construct the set O. For each unsafe downcast instruction $\mathtt{p} = (t)\mathtt{q}$, all allocation-based object a in the points-to set of pointer q will be put into O.

VI. EVALUATION

This section shows that our pointer analysis can significantly remove redundant instrumentations and reduce the runtime overhead for type confusion detection tools and make these tools deployable to production software.

We applied our pointer analysis to the open source type confusion detection tool Typesan [2]. Statically, we can remove 68.67% of instrumentations for tracing types of objects and 57.84% of instrumentations for verifying downcast instructions on average. Dynamically, we can reduce the average (maximum) runtime overhead from 40.10% (89.71%) to 12.22% (24.90%).

A. Implementation

Our pointer analysis is implemented in LLVM-3.9.0. Our pointer analysis is field- and array-sensitive. Each field of a struct is treated as a separate object. Elements of an array are distinguished when accessed using constant indexes. Distinct allocation sites (i.e., ADDRESSOF instructions) are modeled by distinct objects [4], [5]. Our inclusion-based pointer analysis uses a wave propagation solver for constraint resolution,

TABLE III
PROGRAM CHARACTERISTICS.

Program	KLOC	#Stmt	#Ptrs	#Objs	#CallSite
dealII	199	577482	530249	77894	94284
eon	41	65218	63385	15855	14033
omnetpp	48	95961	108349	8592	20601
soplex	41	54190	69287	4191	9878
Xalan	553	744971	703675	73973	106090
Firefox	10304	11094975	20768646	785042	1802572
Total	11186	12632797	22243591	965547	2047458

TABLE IV
NUMBER OF INSTRUMENTATIONS. ALLOCATION: INSTRUMENTATIONS FOR TRACING THE TYPES OF OBJECTS. CAST: INSTRUMENTATIONS FOR RUNTIME CHECK.

Program	Typesan		pta	
	Allocation	Cast	Allocation	Cast
dealII	203776	154	117782	88
eon	17220	7	8042	2
omnetpp	183349	161	59955	57
soplex	3225	2	1407	1
Xalan	144046	2806	60355	1071
Firefox	4169551	81673	1231589	34530

B. Experimental Setup

All our experiments were conducted on a platform consisting of a 3.60GHz Core i7-9700KF CPU with 16 GB memory, running Ubuntu Linux 16.04. The compiler we used is Clang-3.9.0.

In our evaluation, we consider the following three different configurations:

1) **Baseline**: All programs are compiled with Clang-3.9.0 at the default optimization level. No instrumentations are made.
2) **Typesan**: All programs are compiled with Clang-3.9.0 using Typesan [2] with instrumentation enabled.
3) **Pta**: All programs are first compiled with Clang-3.9.0 using the Typesan [2] with instrumentation enabled. Then our pointer analysis will be applied to remove redundant instrumentations.

The overhead of Typesan is the overhead incurred by the Typesan configuration compared with the baseline configuration. The overhead of pta is the overhead of the pta configuration compared with the baseline configuration.

C. Programs

We use five SPEC2000/2006 C++ programs: dealII, eon, omnetpp, soplex, and Xalan and one open-source web browser: Mozilla's Firefox (version 47.0) to evaluate. The rest three SPEC2000/2006 C++ programs: astar, namd, and povray are not selected because there is no runtime check emitted by Typesan [2]. The characteristics of these programs are listed in Table III.

D. Static Statistics

First, we measure how many instrumentations can be removed by pointer analysis. Table IV compares the number of instrumentations of Typesan before and after applying our pointer analysis. Column 2 and column 3 show the number of instrumentations of Typesan for tracing the types of objects

TABLE V
OVERHEAD OF BINARY SIZE AND INSTRUCTION NUMBER.

Program	Binary size (%)		Inst Num (%)	
	Typesan	pta	Typesan	pta
dealII	62.81	18.50	328.21	77.57
eon	78.36	11.38	333.83	15.28
omnetpp	98.57	24.16	369.30	77.13
soplex	30.91	16.23	82.07	38.28
Xalan	81.65	22.38	269.75	53.84
Firefox	128.78	58.70	304.10	64.04

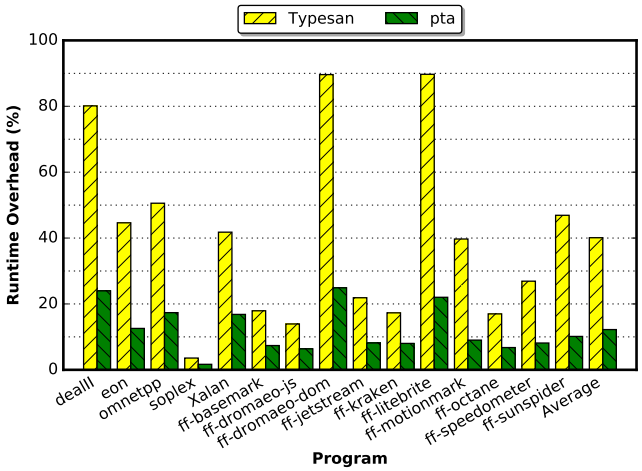

Fig. 5. Runtime overhead of Typesan and pta compared with native runs.

and runtime check, respectively. Column 4 and column 5 show the number of the two types of instrumentations after applying our pointer analysis. For the six programs evaluated, pointer analysis can remove 68.67% of the instrumentations for tracing the types of objects, and 57.84% of the instrumentations for runtime check, on average. The largest reduction in the instrumentations for object tracing and runtime check come from the program Firefox (70.46%) and eon (71.42%).

Table V compares the overhead in binary size (columns 2 and 3) and the number of LLVM instructions (columns 4 and 5) of Typesan and pta compared with the baseline. The average (maximum) overhead in binary size can be reduced from 80.18% (128.78%) to 25.23% (58.70%).

Typesan incurs an average (maximum) overhead of 281.21% (369.30%) in the number of instructions. After applying pointer analysis, the average (maximum) overhead can be reduced to 54.36% (77.57%).

E. Performance Overhead

We measure the performance of the SPEC programs using their reference inputs. We use nine industry browser benchmarks, including basemark [9], dromaeo [10], jetstream [11], kraken [12], litebrite [13], motionmark [14], octane [15], speedometer [16], and sunspider [17] to test Firefox. These benchmarks are designed for evaluating a browser's performance on JavaScript execution and HTML5 3D rendering.

Figure 5 compares the runtime performance overhead of Typesan and pta. For CPU intensive SPEC programs, Typesan can incur as much as 80.12% performance overhead. For Firefox, the highest performance overhead is 89.71%. Such

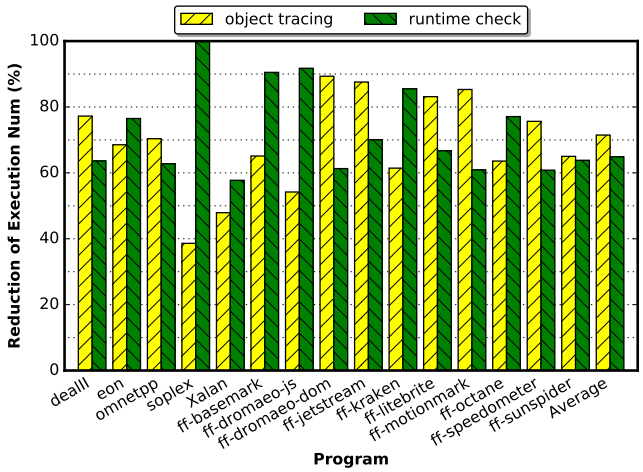

Fig. 6. Reduction in the execution number of instrumentations for object tracing and runtime check.

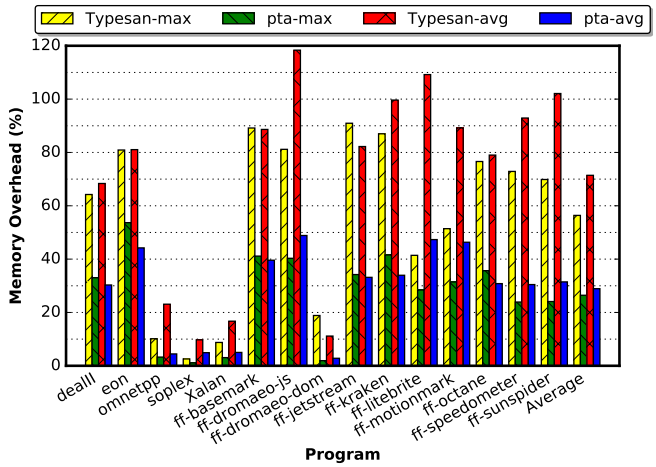

Fig. 7. Memory overhead of Typesan and pta compared with native runs in terms of maximum and average resident set size (RSS).

high runtime performance overhead makes it unpractical to be deployed in production software. After removing the redundant instrumentations by pointer analysis, the highest performance overhead of SPEC programs and Firefox can be reduced to 23.97% and 24.90%, respectively. The average performance overhead of all programs can be reduced from 40.10% to 12.22%, The result shows that our pointer analysis can make type confusion detection tool deployable to production software.

Figure 6 shows the reduction of the execution number of instrumentations for both object tracing and runtime check after applying pointer analysis. On average, 71.47% of instrumentations for object tracing and 64.85% of instrumentations for runtime check can be removed. For the program soplex, we have removed one of the two runtime checks instrumented by Typesan (see Table IV). While the remained runtime check has not been executed, so the reduction in the execution number of runtime checks is 100%.

F. Memory Overhead

We also measure the runtime memory overhead of Typesan and pta compared with native runs. Most of the runtime memory overhead come from (1) the instrumented instructions for object tracing and runtime check, (2) the metadata

required to record the type of each object traced. Since pointer analysis can identify redundant instrumentations, it reduces (1) the number of instructions instrumented, and (2) the metadata required. As a result, it can reduce the runtime memory overhead.

Figure 7 shows the memory overhead in terms of the maximum and average resident set size (RSS). For all the programs, Typesan incurs as much as 90.95% (118.36%) overhead in terms of maximum (average) resident set size. After applying pointer analysis, the highest overhead in maximum (average) resident set size can be reduced to 53.67% (48.85%). On average, the overhead in maximum (average) resident set size can be reduced from 56.39% (71.42%) to 26.45% (28.91%).

VII. Related work

Software security has been studied extensively during the last several decades [18], [19], [20], [21], [22], [23]. Attacks against C/C++ programs usually starts from a memory corruption. So a great deal of research [24], [25], [26], [27] has been devoted to detecting and eliminating memory errors in C/C++ programs. Memory safety measures can stop the attacks against C/C++ programs in their first step. However, as memory safety measures usually incur non-negligible runtime performance overhead, few of them have been deployed in production software.

Control Flow Integrity (CFI) [28] aims to keep a program in a statically computed Control Flow Graph (CFG), even if a memory error took place. Thus it can prevent attackers from compromising the program. After CFI was proposed in 2005, it has drawn much attention from the research community. A lot of effort has been devoted to general CFI [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39], and virtual call protection [40], [41], [42], [43], [44], [45], [46], [47]. General CFI techniques protect both forward (indirect function call) and backward (return instruction) control flows. However, as class hierarchy information is generally missing, the CFG constructed is not precise enough to prevent vtable hijacking attacks in C++ programs. Virtual call protection techniques focus on the protection of virtual calls only. Both source-level and binary-level mitigation techniques have been proposed.

Caver [1] and TypeSan [2] are two typical techniques proposed for detecting type confusion bugs. This paper proposed a novel method to reduce the high runtime performance overhead of type confusion detection tools by using pointer analysis to remove redundant instrumentations.

Pointer analysis is an important static analysis technique. It has been studied intensively during the last several decades [4], [5], [48]. This paper proposes a pointer analysis for C++ programs. Type information for every C++ objects can be inferred using our inference rules. We enforced our pointer analysis results for identifying redundant instrumentations of Typesan [2], and achieved significant performance improvement.

VIII. Conclusion

Type confusion bug is an emerging attack vector in the widely used C++ programming language. Modern type confusion detection tools suffer from the problem of high runtime overhead. This paper proposes a new method to reduce the runtime overhead of type confusion detection tools by applying pointer analysis. We have applied our pointer analysis to Typesan, a type confusion detection tool, and successfully removed large number of redundant instrumentations. As a result, we can reduce the average (maximum) runtime performance overhead from 40.10% (89.71%) to 12.22% (24.90%), making type confusion detection tools deployable to production software.

References

[1] B. Lee, C. Song, T. Kim, and W. Lee, "Type casting verification: Stopping an emerging attack vector," in *USENIX Security '15*, 2015, pp. 81–96.

[2] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe, "Typesan: Practical type confusion detection," in *CCS '16*. ACM, 2016, pp. 517–528.

[3] *Clang UndefinedBehaviorSanitizer*, 2020, https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[4] B. Hardekopf and C. Lin, "Flow-sensitive pointer analysis for millions of lines of code," in *CGO '11*, 2011, pp. 289–298.

[5] O. Lhoták and K.-C. A. Chung, "Points-to analysis with efficient strong updates," in *POPL '11*, 2011, pp. 3–16.

[6] F. M. Q. Pereira and D. Berlin, "Wave propagation and deep propagation for pointer analysis," in *CGO '09*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 126–135.

[7] M. Hind, M. Burke, P. Carini, and J.-D. Choi, "Interprocedural pointer alias analysis," *TOPLAS '99*, vol. 21, no. 4, pp. 848–894, 1999.

[8] G. Balatsouras and Y. Smaragdakis, "Structure-sensitive points-to analysis for C and C++," in *SAS '16*, 2016, pp. 84–104.

[9] *Basemark 3.0 benchmark*, 2020, https://web.basemark.com/.

[10] *Dromaeo JavaScript performance test suite*, Mozilla, 2020, http://dromaeo.com/.

[11] *JetStream JavaScript performance test suite*, BrowserBench, 2020, http://browserbench.org/JetStream/.

[12] *Kraken 1.1 Javascript benchmark suite*, Mozilla, 2020, http://krakenbenchmark.mozilla.org/.

[13] *LiteBrite: HTML, CSS and JavaScript Performance Benchmark*, Microsoft, 2020, https://testdrive-archive.azurewebsites.net/Performance/LiteBrite/.

[14] *MotionMark 1.1 benchmark*, Browserbench, 2020, https://browserbench.org/MotionMark1.1/.

[15] *Octane JavaScript benchmark suite*, Google, 2020, https://chromium.github.io/octane/.

[16] *Speedometer benchmark*, Browserbench, 2020, https://browserbench.org/Speedometer2.0/.

[17] *Sunspider 1.0.2 javascript benchmark suite*, Apple, 2020, https://webkit.org/perf/sunspider/sunspider.html.

[18] W. R. Simpson and K. E. Foltz, "Minimal instantiation of enterprise level security," in *Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering and Computer Science 2019, 22-24 October, 2019, San Francisco, USA*, 2019, pp. 88–93.

[19] M. N. Gedam and B. B. Meshram, "Vulnerabilities & attacks in srs for object-oriented software development," in *Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering and Computer Science 2019, 22-24 October, 2019, San Francisco, USA*, 2019, pp. 94–99.

[20] I. A. Ibraheem, W. Zhang, A. M. Abdelgader, and F. Shu, "Analysis of possible security attacks and security challenges facing vehicular-ad hoc networks," in *Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering and Computer Science 2019, 22-24 October, 2019, San Francisco, USA*, 2019, pp. 144–149.

[21] P. Uma, K. Siddivinayak, and P. Ramachandra, "Smart captcha to provide high security against bots," in *Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering 2019, 3-5 July, 2019, London, U.K*, 2019, pp. 144–149.

[22] N. Jones, Q. Yu, K. Schell, and H. Yu, "Teaching secure program design," in *Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering 2019, 3-5 July, 2019, London, U.K*, 2019, pp. 240–245.

[23] A. Das, S. K. Sarma, and S. Deka, "Data security with dna cryptography," in *Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering 2019, 3-5 July, 2019, London, U.K*, 2019, pp. 246–251.

[24] 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*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 245–258. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542504

[25] ——, "Cets: Compiler enforced temporal safety for c," in *Proceedings of the 2010 International Symposium on Memory Management*, ser. ISMM '10. New York, NY, USA: ACM, 2010, pp. 31–40. [Online]. Available: http://doi.acm.org/10.1145/1806651.1806657

[26] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 3, pp. 477–526, 2005.

[27] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 89–100. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250746

[28] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *CCS' 05*. ACM, 2005, pp. 340–353.

[29] M. Zhang and R. Sekar, "Control flow integrity for cots binaries." in *USENIX Security Symposium*, 2013, pp. 337–352.

[30] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity." in *NDSS '15*, vol. 26, 2015, pp. 27–30.

[31] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses." in *USENIX Security Symposium*, 2014, pp. 385–399.

[32] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection." in *USENIX Security Symposium*, vol. 14, 2014, pp. 401–416.

[33] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 575–589.

[34] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "Ccfi: cryptographically enforced control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 941–951.

[35] B. Niu and G. Tan, "Modular control-flow integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 577–587. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594295

[36] X. Ge, W. Cui, and T. Jaeger, "Griffin: Guarding control flows using intel processor trace," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 585–598.

[37] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan, "Transparent and efficient cfi enforcement with intel processor trace," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 529–540.

[38] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent rop exploit mitigation using indirect branch tracing." in *USENIX Security Symposium*, 2013, pp. 447–462.

[39] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive cfi," in *CCS '15*. ACM, 2015, pp. 927–940.

[40] I. Haller, E. Göktas, E. Athanasopoulos, G. Portokalidis, and H. Bos, "Shrinkwrap: Vtable protection without loose ends," in *ACSAC '15*. ACM, 2015, pp. 341–350.

[41] V. van der Veen, E. Göktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A tough call: Mitigating advanced code-reuse attacks at the binary level," in *S&P '16*, 2016, pp. 934–953.

[42] D. Bounov, R. Kici, and S. Lerner, "Protecting C++ dynamic dispatch through vtable interleaving," in *NDSS '16*, 2016.

[43] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in gcc & llvm," in *USENIX Security '14*, 2014, pp. 941–955.

[44] A. Prakash, X. Hu, and H. Yin, "vfguard: Strict protection for virtual function calls in cots c++ binaries," in *NDSS '15*, 2015.

[45] D. Jang, Z. Tatlock, and S. Lerner, "Safedispatch: securing C++ virtual calls from memory corruption attacks," in *NDSS '14*, 2014.

[46] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song, "Vtrust: Regaining trust on virtual calls," in *NDSS '16*, 2016.

[47] X. Fan, Y. Sui, X. Liao, and J. Xue, "Boosting the precision of virtual call integrity protection with partial pointer analysis for c++," in *ITTSA '17*. ACM, 2017, pp. 329–340.

[48] Y. Sui, X. Fan, H. Zhou, and J. Xue, "Loop-oriented array- and field-sensitive pointer analysis for automatic simd vectorization," in *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems*, ser. LCTES 2016. New York, NY, USA: ACM, 2016, pp. 41–51. [Online]. Available: http://doi.acm.org/10.1145/2907950.2907957