Recovering Container Class Types in C++ Binaries

Xudong Wang*, Xuezheng Xu*, Qingan Li†, Mengting Yuan†, Jingling Xue*
* School of Computer Science and Engineering, UNSW Sydney, Australia
† School of Computer Science, Wuhan University, China
* {xudongw, xuezhengxu, jingling}@cse.unsw.edu.au † {qingan, ymt}@whu.edu.cn

***Abstract*—We present TIARA, a novel approach to recovering container classes in C++ binaries. Given a variable address in a C++ binary, TIARA first applies a new type-relevant slicing algorithm incorporated with a decay function, TSLICE, to obtain an inter-procedural forward slice of instructions expressed as a CFG to summarize how the variable is used in the binary (as our primary contribution). TIARA then makes use of a GCN (Graph Convolutional Network) to learn and predict the container type for the variable (as our secondary contribution). According to our evaluation, TIARA can advance the state of the art in inferring commonly used container types in a set of eight large real-world COTS C++ binaries efficiently (in terms of the overall analysis time) and effectively (in terms of precision, recall and F1 score).**

***Index Terms*—Binary Code Analysis, Type Inference, Containers, Template Classes**

I. Introduction

Binary type inference aims to recognize typed variables from untyped memory locations in binary executables [1]. This has many applications, such as binary code reuse [2], [3], reverse engineering [4], [5], vulnerability detection [4], [6], and memory forensics [7]. For most COTS binaries, neither source code nor debugging information is available. With binary type inference, it is possible, albeit extremely challenging, to recover semantic information from their binaries.

Problem Statement. We address the problem of recovering the container class type of a given variable address in a C++ binary statically. For C++ programs, the C++ STL provides a set of template classes for implementing standard data structures such as linked lists, vectors, and maps. C++ templates support compile-time polymorphism instead of run-time polymorphism, enabling C++ programs to reuse template classes without the overhead of run-time performance (incurred for resolving virtual functions). As a result, templates are widely used in implementing container classes in modern C++ programs [8]. Discovering container classes in binaries leads to better understanding of C++ executables.

Prior Work. To the best of our knowledge, there is no prior work on recovering container classes in C++ binaries statically. Some past efforts identify certain data structures in C/C++ binaries dynamically [9]–[12]. Some other past efforts infer ordinary classes in C++ binaries statically [13]–[19], but they are inapplicable in our setting, since they rely on their virtual function tables (for polymorphic classes), their member functions, the "this" pointer, and their data

```
    std::list<int> l;
    std::vector<int> v;
    l.push_back(10);
    v.push_back(20);
```

```
  l.push_back(10);
00071164  mov  esi,dword ptr [l (074004h)]
0007116A  lea  eax,[argn]
  ...
00071179  call std::_List_buy<int>::_Buynode<int>(
  ...
00071192  call dword ptr [_Xlength_error (073034h)]
00071198  inc  ecx
  v.push_back(20);
00071199  mov  daword ptr [ebp+8],14h
  l.push_back(10);
000711A0  mov  dword ptr ds:[74408h],ecx
  ...
000711AC  mov  dword ptr [eax],edx
  v.push_back(20);
000711AE  lea  eax,[ebp+8]
  ...
```

Fig. 1. The disassembled code for the code snippet given.

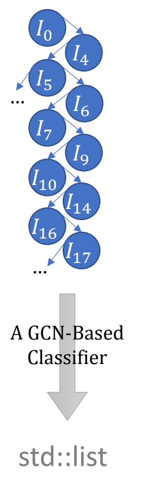
sizes. OOANALYZER [20], which solves a different type inference problem for C++ binaries, distinguishes different ordinary classes by solving constraints in terms of code usage patterns. In the presence of container classes, which have different type-dependent instantiations, OOANALYZER can only discover that these instantiations represent different but unknown classes, but TIARA recognize, for example, which are `std::list` and which are `std::vector`. For C binaries, DEBIN [21] represents the state of the art, but it focuses on identifying primitive types (by applying machine learning techniques). As for compound types, DEBIN can only classify all different types of structs as one non-primitive type.

Challenges. There are several challenges in inferring container classes in C++ binaries statically. First, C++ containers embrace compile-time polymorphism (without resorting to virtual member functions). Therefore, it is not possible to identify container classes by looking for their virtual function tables (VFTs) in binaries, as is done for polymorphic classes [14], [17]. Second, the C++ compiler often applies function inlining to improve performance. This has two consequences: (1) multiple copies of a member function may co-exist in the binary, and (2) the instructions of the functions from different container classes may be interleaved. In Figure 1, `l.push_back()` and `v.push_back()` are inlined, with their inlined code sequences mixed together. Therefore, it is difficult to identify container classes based on their member

The last two authors are the corresponding authors of this paper.

I	Disassembly	Tracing	Rules	Faith	Dep	
I_0	mov esi,dword ptr [v_0(074404h)]	$esi \mapsto \{(ref, 0)\}$	[MOV-RIV]	1	T	Add a new node and link it to the list
I_1	lea eax, [argn]		[MOV-RV-KILL]	0.999	F	
I_2	push eax	$\mathcal{S}[0] \mapsto \{\}$	[STK-PUSH]	0.994	F	
I_3	mov dword ptr [argn], 0ah		[MOV-RC];[MOV-SR]	0.993	F	
I_4	push dword ptr [esi+4]	$\mathcal{S}[1] \mapsto \{(other, *)\}$	[MOV-RI];[STK-PUSH]	0.983	T	
I_5	push esi	$\mathcal{S}[2] \mapsto \{(ref, 0)\}$	[STK-PUSH]	0.978	T	
I_6	call _List_buy::_Buynode		[STK-PUSH]	0.218	T	
I_7	mov ecx, dword ptr ds:[v_0+4]	$ecx \mapsto \{(ref, 4)\}$	[MOV-RV]	0.217	T	Increase the list size by 1
I_8	mov edx, eax		[MOVE-RR]	0.216	F	
I_9	sub ebx, ecx	$ebx \mapsto \{(other, *)\}$	[OP-RR]	0.215	T	
I_{10}	cmp ebx, 1	Use of ebx	[USE-DEP]	0.214	T	
I_{11}	jae main+128h (I_{14})			0.213	F	
I_{12}	push offset string...		[MOV-RC];[STK-PUSH]	0.208	F	
I_{13}	call _Xlength_error		[STK-PUSH]	0	F	
I_{14}	inc ecx	$ecx \mapsto \{(other, *)\}$	[OP-RC]	0.213	T	
I_{15}	mov dword ptr [ebp+8], 14h		[MOV-RC];[MOV-SR]	0.212	F	
I_{16}	mov dword ptr ds:[v_0+4], ecx		[MOV-RV];[MOV-DR]	0.211	T	
I_{17}	mov dword ptr [esi+4], edx		[MOV-DR]	0.201	T	Link the list to the new node added
I_{18}	mov eax, dword ptr [edx+4]		[MOV-RI]	0.191	F	
I_{19}	mov dword ptr [eax], edx			0.181	F	
I_{20}	lea eax, [ebp+8]		[MOV-RIV-KILL]	0.180	F	
...	...					

(a) Type-relevant slicing (*other* is an unknown v_0-dependent value)



(b) Type classification

Fig. 2. An example illustrating how TIARA predicts the container type of an address v_0, which is actually the address of variable `l` of type `std::list` in Figure 1. The instructions in gray are translated from `l.push_back(10)` and the remaining ones from `v.push_back(20)` in Figure 1. The instructions marked with a `T` in (a), i.e., those in (b) are in the slice of v_0 found.

functions and the "this" pointer, as for polymorphic classes [15]. Third, for a variable of a container type T, the data size of T depends on its type parameters. If T is $T*$ instead, the data size of $T*$ is fixed but no size information about T is revealed. Thus, it is also difficult to identify the container type of a variable from its data size. Finally, for large COTS C++ binaries such as `clang` (57MB stripped) containing more than 100K variable addresses, how to infer their container classes efficiently and accurately is non-trivial.

Our Solution. We present TIARA, a novel approach to recovering the container types of the variables in stripped COTS C++ binaries statically. Our key insight is that variables of different container types exhibit different behaviors in terms of how they are used. For example, `std::vector` and `std::list` share an identically-named member function, `push_back()`, but the former performs heap reallocation (via `malloc()` and `free()`) internally while the latter performs only allocation (via `malloc()`). Thus, the variables of these two types can be identified by exploiting such use-related features.

Therefore, TIARA infers the container type of an address representing a variable in a binary in two stages. In the first stage (as our primary contribution), we apply a new type-relevant slicing algorithm incorporated with a decay function, TSLICE, to obtain an inter-procedural forward slice of instructions context-sensitively, expressed as a CFG to summarize how the variable is used. For large binary programs, traditional slicing algorithms are unscalable or "very imprecise, often including essentially the entire program" [22]. Thus, TSLICE is designed to find a small yet relevant slice to capture where and how the variable is used both efficiently (0.2 seconds per slice, on average) and effectively (with 50 instructions per slice, on average, that are sufficient to identify its container type in most cases). In the second stage (as our secondary contribution), we make use of a GCN (Graph Convolutional Network) [23], [24] to predict the container type for the given variable (based on a learned classifier).

Contributions. TIARA (https://sites.google.com/view/tiara-tool) represents a general-purpose approach to recovering the container classes in C++ binaries statically. TIARA is expected to provide benefits for many other binary type inference tasks mentioned earlier.

1) We present TIARA, a novel type inference approach to recovering container types in COTS C++ binaries.
2) We offer a type-relevant slicer, TSLICE, which extracts just enough relevant instructions for a variable in C++ binaries in order to characterize its container type.
3) We design a GCN-based classifier to learn and predict the container types of the variables in C++ binaries, where data labeling is fully automatic.
4) We show that TIARA advances the state of the art in inferring STL container types in eight real-world COTS C++ binaries efficiently (in terms of analysis times) and effectively (in terms of precision, recall and F1 score).

The rest of the paper is organized as follows. Section II, motivates TIARA with an example. Section III, introduces TIARA. Section IV evaluates TIARA. Section V discusses the related work. Finally, Section VI concludes the paper.

II. MOTIVATION

We illustrate the basic idea behind TIARA by using a binary program in Figure 2, translated from the code snippet in Figure 1. Given v_0 = `074404h`, which represents the address of variable l of type `std::list` at the source code, we describe how TIARA predicts its type at the binary level. TIARA works by first finding a type-dependent slice starting from v_0, as shown in Figure 2(a) (Section II-A) and then using a machine-learning-based type classifier to infer its type, as shown in Figure 2(b) (Section II-B). In this work, we focus on inferring container types and will thus treat all the primitive types as one single primitive type (non-discriminately).

A. Type-Relevant Slicing

We explain first how TSLICE works and then how we address the efficiency and precision challenges in slicing binary code to recover the container types in COTS C++ binaries.

In Figure 2, its instructions are translated from Figure 1, where `l.push_back(10)` and `v.push_back(20)` are inlined and interleaved. $I_0 - I_{14}$ and $I_{16} - I_{19}$ are from `l.push_back(10)` and the rest from `v.push_back(20)`. We can divide the instructions for `l.push_back(10)` into (1) $I_0 - I_6$ for calling `_Buynode(_Nodeptr _Next, _Nodeptr _Prev, _Valty&& ... _Val)`, where its actual arguments are $*v_0$, $*v_0 + 4$ and 10 (stored in `argn`), to create a new node and link it to `_Next` and `_Prev`, (2) $I_7 - I_{14}$ and I_{16} for calling `_Incsize(size_t _Count)` (inlined) to increase the size of the underlying `std::list` by 1 (with error handling), and (3) $I_{17} - I_{19}$ for linking `_Next` and `_Prev` to the new node created.

Given v_0 as a slicing criterion, Figure 2(a) illustrates the slicing process. Column "Disassembly" lists the instructions in the assembly format. Column "Tracing" shows the dependence-based analysis for each instruction along the control flow, attempting to establish its dependence with v_0 context-sensitively. Column "Rules" gives the rules used. Column "Faith" indicates the amount of faith we have on an instruction being v_0-dependent (calculated by a decay function along the control flow). Finally, Column "Dep" indicates whether an instruction actually depends on v_0 or not.

Starting from v_0, TSLICE computes context-sensitively an inter-procedural forward slice of v_0, $\mathscr{S}_{v_0}$, by searching for the instructions that depend on v_0, i.e., operate on values derived from v_0. TSLICE starts from I_0, the first instruction operating on $v_0 = \mathtt{074404h}$, and searches for some v_0-dependent instructions forwards. As I_0 moves $*v_0$ to register `esi`, we can assert that (1) I_0 depends on v_0, and (2) any future instruction that uses `esi` will also depend on v_0 (if not killed). Thus, we record `esi` $\mapsto \{(\mathit{ref}, 0)\}$, where (ref, c) denotes the value $*(v_0 + c)$, and mark `esi` as being dependent on v_0. I_1 is skipped as it does not depend on v_0. When I_2 is analyzed, we keep track of the values in the call stack, abstracted as $\mathcal{S}$, in order to find the stack values that are dependent on v_0. As I_2 does not depend on v_0, $\mathcal{S}[0] \mapsto \{\}$ (with an empty set of v_0-dependent values recorded). I_3 is skipped as it does not depend on v_0. For I_4, we record $\mathcal{S}[1] \mapsto \{(\mathit{other}, *)\}$, where $(\mathit{other}, *)$ represents a v_0-dependent but unknown value (i.e., $*v_0 + 4$, a value computed by an arithmetic operation on a heap address v_0) that will not be further tracked precisely. I_5 is analyzed similarly as I_2, except that $\mathcal{S}[2] \mapsto \{(ref, 0)\}$ is recorded. As I_6 is a call instruction, we analyze context-sensitively `_Buynode()`, which allocates a new node in the heap. At the end of its analysis, the top three elements in $\mathcal{S}$ will be removed (due to its three pop instructions, which match the three push instructions I_2, I_4 and I_5). When I_7 is analyzed, we record $ecx \mapsto \{(\mathit{ref}, 4)\}$, as I_7 also depends on v_0 (i.e., $*(v_0 + 4)$). I_9 and I_{14} are each found to operate a v_0-dependent but unknown value, obtained by an arithmetic instruction on $*(v_0{+}4)$. $I_{17}{-}I_{19}$ serve to adjust the pointers in the underlying `std::list` to point to the new node created by the call to _Buynode() at I_6. As TSLICE keeps track of the dependences on v_0 only, L_{17} is considered to be dependent on v_0 but I_{18} and I_{19} are not. Finally, $\mathscr{S}_{v_0}$ consists of the instructions given in $\{I_0, I_4\text{–}I_7, I_9\text{–}I_{10}, I_{14}, I_{16}, I_{17}\}$.

TSLICE is designed to recover container types in COTS C++ binaries with the following two salient properties:

- **Type-Relevant Tracing.** TSLICE traces context-sensitively from v_0 by including in $\mathscr{S}_{v_0}$ only a sample of instructions characterizing how v_0 is used. In our example, $\mathscr{S}_{v_0}$ contains (1) $\{I_0, I_4 - I_6\}$, a subset of the instructions for calling `_Buynode()` to allocate a new node, (2) $\{I_7, I_9, I_{10}, I_{14}\}$, a subset of the instructions for incrementing the size of the underlying `std:list` by 1, and (3) $\{I_{17}\}$, a subset of the instructions for adjusting the `std::list` to accommodate the new node inserted. Unlike existing techniques [13]–[20], which are inapplicable to recovering container types (as discussed in Section I), TSLICE overcomes their limitations by characterizing v_0 with a number of its use-related instructions.
- **Type-Relevant Pruning.** TSLICE prunes away instructions based on three principles, by (1) ignoring some potentially v_0-dependent instructions, such as I_{18} and I_{19}, that do not operate on some values derived from v_0, (2) abstracting v_0-dependent values in the heap (obtained by performing arithmetic operations on heap addresses of the form $*(v_0 + c)$, where c is a constant), such as I_4, I_9 and I_{14}, by an v_0-dependent but unknown value, $(\mathit{other}, *)$, and (3) using a decay function to decease the faith or likelihood of an instruction's dependence on v_0 along the control flow. For C++ container types, our key insight is that the slice $\mathscr{S}_{v_0}$ that contains the v_0-dependent instructions captured via the register and stack dependences according to our inference rules is sufficient to predict its type in most cases (Section IV). This is in sharp contrast to the state-of-the-art (sound) slicing algorithms for binary codes, which are either unscalable or imprecise [22] (Section I).

As a result, TSLICE is both efficient (by finding type-relevant slices in seconds each) and effective (by enabling predicting the C++ container types of variables accurately). To the best of our knowledge, this is the first paper for recovering C++ container types in real-world stripped COTS C++ binaries.

B. Type Classification

Once a slice $\mathscr{S}_{v_0}$ (represented as a CFG) has been found for v_0, TIARA will make use of a GCN-based classifier (for the first time), which is pre-trained with automatically labeled data, to predict the type of v_0, as illustrated in Figure 2(b).

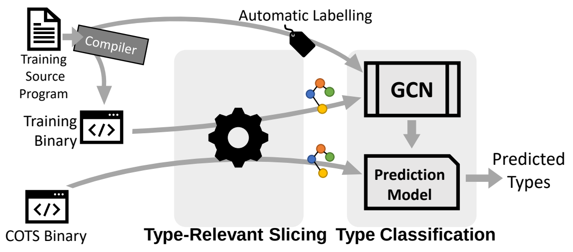


Fig. 3. Workflow of TIARA (with two stages).

III. DESIGN OF TIARA

TIARA proceeds in two stages (Figure 3). In the type-relevant slicing stage (Section III-A), an inter-procedural forward slice for a a variable address is found context-sensitively. In the type classification stage (Section III-B), a GCN-based classifier is used to predict the type for the given variable.

A. Type-Relevant Slicing

To infer the type of a variable at a given address v_0 in a C++ binary program, TIARA will construct a small forward slice that reflects the behavior of v_0 by using v_0 as the slicing criterion. To reason about the data dependences in the binary program, TIARA will disassemble it into an intermediate representation (IR). We introduce our slicing algorithm for a small language, where an instruction $\mathcal{I}$ has the form:

$$\begin{aligned} \mathcal{I} &:= \textbf{mov}\ opr^1, opr^2 \mid \textbf{op}_{\oplus}\ opr^1, opr^2 \\ &\quad\mid \textbf{use}\ \ldots\ opr^k \ldots \mid \textbf{push}\ r \mid \textbf{pop}\ r \\ opr &:= c \mid loc \mid [loc] \\ loc &:= addr \mid addr + c \\ addr &:= r \mid m \end{aligned} \tag{1}$$

A **mov** instruction moves a value from opr^2 to opr^1. An **op**$_\oplus$ instruction represents a binary arithmetic operation such as `add` or `mul`. For example, "**op**$_+$ opr^1, opr^2" adds opr^1 and opr^2 and then stores (i.e., moves) the result to opr^1. A **use** instruction represents any instruction (e.g., **jmp**) that reads the given operands without any side effect. An operand opr can be either a constant c, a reference to a location loc, or an indirect reference to loc. A location loc may be an address $addr$, or an $addr$ with an offset c. Finally, an address $addr$ may denote either a register r or a memory address m.

In our language, function calls are represented implicitly. A call instruction can be modeled simply as a **push** followed by a **use** (i.e., **jmp**) while a return instruction is modeled as a **pop** followed by a **use** (i.e., **jmp**). As a result, we can use one single CFG, $G = (I, E)$, to represent a binary program, where $I = \{I_0, I_1, ..., I_n\}$ is the set of instructions and $E \subseteq I \times I$ is the set of edges. If $(I_p, I_s) \in E$, I_s may be executed right after I_p. Let I_0 be the entry of the program. For a variable v_0, TSLICE aims to find a forward slice, i.e., a subset $\mathscr{S}_{v_0} \subseteq I$.

TSLICE builds $\mathscr{S}_{v_0}$, starting from I_0 (as any instruction may operate on v_0), by finding the v_0-dependent instructions along the control flow through reasoning about data dependences. To keep $\mathscr{S}_{v_0}$ small, TSLICE will keep track of only approximately

```
Input: A CFG $G = (I, E)$ and a variable address $v_0$
Output: A subset $\mathscr{S}_{v_0} \subseteq I$.
Data: $\mathcal{V} : I \to (R \to 2^{\mathcal{A}})$, $\mathcal{S} : I \to (\mathbb{Z} \to 2^{\mathcal{A}})$,
      $\mathcal{D} : I \to \{true, false\}$, and $\mathcal{F} : I \to [0, 1]$.
1  $\mathcal{V}(I_0) \leftarrow \lambda r.\emptyset$;
2  $\mathcal{S}(I_0) \leftarrow \lambda z.\emptyset$;
3  $\mathcal{D} \leftarrow \lambda i.$if $i = I_0$ $true$ else $false$;
4  $\mathcal{F} \leftarrow \lambda i.1$;
5  foreach $i \in I$ do
       $Decay(i) = \begin{cases} 0.01 & \textbf{if } i \text{ is in an indirect addressing mode} \\ 0.005 & \textbf{elif } i \text{ is push or pop} \\ 0.001 & \textbf{otherwise} \end{cases}$
6  foreach $j \in I$ such that $(I_0, j) \in E$ do
       CompDependences($I_0, j$);
7  return $\mathscr{S}_{v_0} = \{i \in I \mid \mathcal{D}(i) = true\}$
   procedure CompDependences($pre : I, i : I$)
8      if $\mathcal{D}(pre) = true$ and $\mathcal{F}(pre) = 0$ then return;
9      Update $(\mathcal{V}(i), \mathcal{S}(i), \mathcal{D}(i))$ by the rules in Figure 4;
10     $\mathcal{F}(i) \leftarrow \max(\min(\mathcal{F}(pre), \mathcal{F}(i)) - Decay(i), 0)$;
11     if $(\mathcal{V}(i), \mathcal{S}(i), \mathcal{D}(i))$ is not changed then return;
12     foreach $j \in I$ such that $(i, j) \in E$ do
           CompDependences($i, j$);
   end
```

Algorithm 1: Finding a forward slice.

the v_0-dependent heap values that are obtained by arithmetic operations on $*(v_0 + c)$, where c is a constant. As the majority of template-related values (such as iterators) are already allocated in registers and the call stack by modern C++ compilers, $\mathscr{S}_{v_0}$ will usually still contain the relevant instructions for the type of v_0 to be deduced.

We make use of four functions to reason about data dependences. The function $\mathcal{V} : I \to (R \to 2^{\mathcal{A}})$ records all possible values in a register after an instruction $i \in I$ has been executed, where R is the set of registers and $\mathcal{A} = \{ptr, ref, const\} \times \mathbb{Z} \cup \{(other, *)\}$ denotes the set of all possible values that TSLICE may care about. If $(ptr, c) \in \mathcal{V}(i)(r)$, register r may contain a pointer to $(v_0 + c)$ after i has been executed. Similarly, (ref, c) means that r may contain the value stored in $v_0 + c$ (aka. $*(v_0 + c)$), $(const, c)$ represents a constant c, and $(other, *)$ denotes a v_0-dependent but unknown value. The function $\mathcal{S} : I \to (\mathbb{Z} \to 2^{\mathcal{A}})$ reveals the set of stack values at an offset $z \in \mathbb{Z}$ from $fp \in R$ after an instruction $i \in I$ has been executed. In order to track the inter-procedural data flow, TSLICE monitors the frame pointer register $fp \in R$ and the stack pointer register $sp \in R$ (i.e., `ebp` and `esp` in x86) to update $\mathcal{S}$. The function $\mathcal{D} : I \to \{true, false\}$ indicates whether an instruction is data-dependent on v_0. The function $\mathcal{F} : I \to [0, 1]$ estimates the faith, i.e., the likelihood of an instruction i's dependence on v_0. If $\mathcal{F}(i) = 1$, TSLICE is fully confident that i depends on v_0. If $\mathcal{F}(i) = 0$, TSLICE believes that i does not depend on v_0 at all. When computing $\mathcal{F}(i)$, TSLICE may opt to decay $\mathcal{F}(i)$, as it only needs to find a decent number of instructions to capture the behavior of v_0.

As depicted in Algorithm 1, TSLICE starts from I_0 (lines 1 – 5) and invokes *CompDependences()* recursively to update $\mathcal{V}$, $\mathcal{S}$, $\mathcal{D}$, and $\mathcal{F}$ for each reachable instruction (line 6). After all the dependent instructions have been found, $\mathscr{S}_{v_0}$ is obtained

$$\frac{r \notin \{fp, sp\} \qquad \Delta = \{(ptr, c)\} \qquad U = \mathcal{V}(i)(r) \cup \Delta}{\Gamma; \mathit{mov}\ r, v_0 + c \vdash \mathcal{V}(i)[r \mapsto U] \qquad \mathcal{D}[i \mapsto \mathit{true}]}\text{[Mov-rv]} \qquad \frac{r \notin \{fp, sp\} \qquad v \neq v_0}{\Gamma; \mathit{mov}\ r, v + c \vdash \mathcal{V}(i)[r \mapsto \emptyset]}\text{[Mov-rv-kill]}$$

$$\frac{r \notin \{fp, sp\} \qquad \Delta = \{(ref, c)\} \qquad U = \mathcal{V}(i)(r) \cup \Delta}{\Gamma; \mathit{mov}\ r, [v_0 + c] \vdash \mathcal{V}(i)[r \mapsto U] \qquad \mathcal{D}[i \mapsto \mathit{true}]}\text{[Mov-riv]} \qquad \frac{r \notin \{fp, sp\} \qquad v \neq v_0}{\Gamma; \mathit{mov}\ r, [v + c] \vdash \mathcal{V}(i)[r \mapsto \emptyset]}\text{[Mov-riv-kill]}$$

$$\frac{r_1 \notin \{fp, sp\} \qquad \Delta = \mathcal{V}(pre)(r_2) \qquad U = \mathcal{V}(i)(r_1) \cup \Delta}{\Gamma; \mathit{mov}\ r_1, r_2 \vdash \mathcal{V}(i)[r_1 \mapsto U] \qquad \mathcal{D}[i \mapsto \text{hasDep}(\Delta) \vee \mathcal{D}(i)]}\text{[Mov-rr]}$$

$$\frac{r_1 \notin \{fp, sp\} \qquad \Delta = \{(ref, c + c') \mid (ptr, c') \in \mathcal{V}(pre)(r_2)\} \cup \{(other, *) \mid (ref, c') \in \mathcal{V}(pre)(r_2)\} \qquad U = \mathcal{V}(i)(r_1) \cup \Delta}{\Gamma; \mathit{mov}\ r_1, [r_2 + c] \vdash \mathcal{V}(i)[r_1 \mapsto U] \qquad \mathcal{D}[i \mapsto \text{hasDep}(\Delta) \vee \mathcal{D}(i)]}\text{[Mov-ri]}$$

$$\frac{r \notin \{fp, sp\} \qquad \{(const, n)\} = \mathcal{V}(pre)(fp) \qquad \Delta = \mathcal{S}(pre)(n + c) \qquad U = \mathcal{V}(i)(r) \cup \Delta}{\Gamma; \mathit{mov}\ r, [fp + c] \vdash \mathcal{V}(i)[r \mapsto U] \qquad \mathcal{D}[i \mapsto \text{hasDep}(\Delta) \vee \mathcal{D}(i)]}\text{[Mov-rs]} \quad \frac{\{(const, n)\} = \mathcal{V}(pre)(fp) \qquad \Delta = \mathcal{V}(pre)(r) \qquad U = \mathcal{S}_i(n + c) \cup \Delta}{\Gamma; \mathit{mov}\ [fp + c], r \vdash \mathcal{S}(i)[n + c \mapsto U] \qquad \mathcal{D}[i \mapsto \text{hasDep}(\Delta) \vee \mathcal{D}(i)]}\text{[Mov-sr]}$$

$$\frac{r \notin \{fp, sp\} \qquad \Delta = \{(const, c)\} \qquad U = \mathcal{V}(i)(r) \cup \Delta}{\Gamma; \mathit{mov}\ r, c \vdash \mathcal{V}(i)[r \mapsto U]}\text{[Mov-rc]} \qquad \frac{r \notin \{fp, sp\} \qquad addr \neq v_0}{\Gamma; \mathit{mov}\ r, addr + c \vdash \mathcal{V}(i)[r \mapsto \emptyset]}\text{[Mov-rc-kill]} \qquad \frac{r \in \{fp, sp\} \qquad \Delta = \{(const, c)\}}{\Gamma; \mathit{mov}\ r, c \vdash \mathcal{V}(i)[r \mapsto \Delta]}\text{[Mov-rc-1]}$$

$$\frac{\{(const, s)\} = \mathcal{V}(pre)(sp)}{\Gamma; \mathit{mov}\ fp, sp \vdash \mathcal{V}(i)[fp \mapsto \{(const, s)\}]}\text{[Mov-fp]} \qquad \frac{\{(const, n)\} = \mathcal{V}(pre)(fp)}{\Gamma; \mathit{mov}\ sp, fp \vdash \mathcal{V}(i)[sp \mapsto \{(const, n)\}]}\text{[Mov-sp]} \qquad \frac{B = \text{hasDep}(\mathcal{V}(pre)(r))}{\Gamma; \mathit{mov}\ [r + c], r' \vdash \mathcal{D}[i \mapsto B \vee \mathcal{D}(i)]}\text{[Mov-dr]}$$

$$\frac{r \notin \{fp, sp\} \qquad \Delta = \{(t, c' \oplus c) \mid (t, c') \in \mathcal{V}(pre)(r)\} \qquad U = \mathcal{V}(i)(r) \cup \Delta}{\Gamma; op_{\oplus}\ r, c \vdash \mathcal{V}(i)[r \mapsto U] \qquad \mathcal{D}[i \mapsto \text{hasDep}(\mathcal{V}(pre)(r)) \vee \mathcal{D}(i)]}\text{[Op-rc]} \qquad \frac{r \in \{fp, sp\} \qquad \{(const, n)\} = \mathcal{V}(pre)(r) \qquad \Delta = \{(t, n \oplus c)\}}{\Gamma; op_{\oplus}\ r, c \vdash \mathcal{V}(i)[r \mapsto \Delta]}\text{[Op-rc-1]}$$

$$\frac{r_1 \notin \{fp, sp\} \qquad U = \mathcal{V}(i)(r_1) \cup \{(t, c \oplus c') \mid (const, c) \in \mathcal{V}(pre)(r_1), (t, c') \in \mathcal{V}(pre)(r_2)\} \cup \{(t, c \oplus c') \mid (t, c) \in \mathcal{V}(pre)(r_1), (const, c') \in \mathcal{V}(pre)(r_2)\}}{\Gamma; op_{\oplus}\ r_1, r_2 \vdash \mathcal{V}(i)[r_1 \mapsto U] \qquad \mathcal{D}[i \mapsto \text{hasDep}(\mathcal{V}(pre)(r_2)) \vee \mathcal{D}(i)]}\text{[Op-rr]}$$

$$\frac{r_1 \notin \{fp, sp\} \qquad \Delta = \{(other, *) \mid (t, c) \in \mathcal{V}(i)(r_1), t \neq const, (t', c') \in \mathcal{V}(pre)(r_2), t' \in \{ref, other\}\}}{\Gamma; op_{\oplus}\ r_1, r_2 \vdash \mathcal{V}(i)[r_1 \mapsto \mathcal{V}(i)(r_1) \cup \Delta] \qquad \mathcal{D}[i \mapsto \text{hasDep}(\mathcal{V}(pre)(r_2)) \vee \mathcal{D}(i)]}\text{[Op-rref]}$$

$$\frac{r_1 \notin \{fp, sp\} \qquad \Delta = \{(other, *) \mid (ptr, c') \in \mathcal{V}(pre)(r_2)\} \qquad U = \mathcal{V}(i)(r_1) \cup \Delta}{\Gamma; op_{\oplus}\ r_1, [r_2 + c] \vdash \mathcal{V}(i)[r_1 \mapsto U] \qquad \mathcal{D}[i \mapsto \text{hasDep}(\mathcal{V}(pre)(r_2)) \vee \mathcal{D}(i)]}\text{[Op-ri]}$$

$$\frac{r \notin \{fp, sp\} \qquad \{(const, n)\} = \mathcal{V}(pre)(fp) \qquad \Delta = \{(other, *) \mid (t, c') \in \mathcal{S}(pre)(n + c), t \neq const\}}{\Gamma; op_{\oplus}\ r, [fp + c] \vdash \mathcal{V}(i)[r \mapsto \mathcal{V}(i)(r) \cup \Delta] \qquad \mathcal{D}[i \mapsto \text{hasDep}(\mathcal{S}(pre)(n + c)) \vee \mathcal{D}(i)]}\text{[Op-rs]}$$

$$\frac{\{(const, n)\} = \mathcal{V}(pre)(fp) \qquad \Delta = \{(other, *) \mid (t, c') \in \mathcal{V}(pre)(r), t \neq const\}}{\Gamma; op_{\oplus}\ [fp + c], r \vdash \mathcal{S}(i)[n + c \mapsto \mathcal{S}(i)(n + c) \cup \Delta] \qquad \mathcal{D}[i \mapsto \text{hasDep}(\mathcal{V}(pre)(r)) \vee \mathcal{D}(i)]}\text{[Op-sr]}$$

$$\frac{\{(const, s)\} = \mathcal{V}(pre)(sp) \qquad \Delta = \mathcal{V}(i)(r) \qquad U = \mathcal{S}(i)(s) \cup \Delta}{\Gamma; \mathit{push}\ r \vdash \mathcal{V}(i)[sp \mapsto \{(const, s + 1)\}] \quad \mathcal{S}(i)[s \mapsto U] \quad \mathcal{D}[i \mapsto \text{hasDep}(\Delta) \vee \mathcal{D}(i)]}\text{[Stk-Push]} \quad \frac{\{(const, s)\} = \mathcal{V}(pre)(sp) \qquad \Delta = \mathcal{S}(pre)(s) \qquad U = \mathcal{V}(i)(r) \cup \Delta}{\Gamma; \mathit{pop}\ r \vdash \mathcal{V}(i)[sp \mapsto \{(const, s - 1)\}, r \mapsto U] \quad \mathcal{D}[i \mapsto \text{hasDep}(\Delta) \vee \mathcal{D}(i)]}\text{[Stk-Pop]}$$

$$\frac{P = \bigvee_{k} \textbf{if } opr^k \in \{r, [r + c]\}\ s.t.\ r \neq fp \textbf{ then } \text{hasDep}(\mathcal{V}(i)(r)) \textbf{ elif } opr^k = [fp + c]\ s.t.\ \{(const, n)\} = \mathcal{V}(pre)(fp) \textbf{ then } \text{hasDep}(\mathcal{S}(i)(n + c))}{\Gamma; \mathit{use}\ \ldots opr^k \ldots \vdash \mathcal{D}[i \mapsto P \vee \mathcal{D}(i)]}\text{[Use-dep]}$$

Fig. 4. Rules for updating $\mathcal{V}(i)$, $\mathcal{S}(i)$ and $\mathcal{D}(i)$ based on $\mathcal{V}(pre)$, $\mathcal{S}(pre)$ and $\mathcal{D}(pre)$ at instruction i.

as desired (line 7). When *CompDependences()* is called, i is an instruction and pre is one of its predecessors. If $\mathcal{F}(pre)$ (the faith of pre) has decayed to 0, *CompDependences()* simply returns (line 8). Otherwise, $\mathcal{V}(i)$, $\mathcal{S}(i)$, and $\mathcal{D}(i)$ are updated according to the rules in Figure 4 (line 9) and $\mathcal{F}(i)$ is updated (line 10), as explained below. If $\mathcal{V}$, $\mathcal{S}$ or $\mathcal{D}$ has changed after i has been analyzed, *CompDependences()* is invoked recursively to update the dependence for each successor of i (line 12). Otherwise, *CompDependences()* returns (line 11).

In TSLICE, we use the faith function $\mathcal{F}$ and a decay function, *Decay* : $I \rightarrow \mathbb{N}$, to find a small slice with relevant instructions quickly. Initially, if an instruction i is found to depend on v_0, we are fully confident about the dependence, since $\mathcal{F}(i) = 1$ (line 4). Every time when we descend to i from one of its predecessor instructions, pre, $\mathcal{F}(i)$ is decayed according to line 10 to ensure that $\mathcal{F}(i)$ drops monotonically (since *Decay*$(i) > 0$) and quickly (due to the use of min). As a result, no successor instructions of pre will ever be visited once $\mathcal{F}(pre) = 0$ (line 8), since we are now fully confident about their independence on v_0. To define *Decay*, we use a linear decay function (line 5). When an instruction i is visited (line 10), we decrease our confidence about its dependence on v_0 by 0.001, in general. However, our decrement will be 0.005 for each push or pop instruction and 0.01 for each indirect addressing instruction, as we become less and less confident about the dependence of i on v_0. These heuristically tuned parameters work well in practice (Section IV). Of course, other more sophisticated decay functions can also be used.

Let us explain the notations used in our inference rules (Figure 4) for updating $(\mathcal{V}(i), \mathcal{S}(i), \mathcal{D}(i))$ based on $(\mathcal{V}(pre), \mathcal{S}(pre), \mathcal{D}(pre))$ given. Given an evaluation environment consisting of (1) both $\Gamma = (\mathcal{V}(pre), \mathcal{S}(pre), \mathcal{D}(pre), \mathcal{V}(i), \mathcal{S}(i), \mathcal{D}(i))$ and (2) i is an instruction, each rule gives the updated $(\mathcal{V}(i), \mathcal{S}(i), \mathcal{D}(i))$ in the conclusion under its given premises. Given a function $F \in \{\mathcal{V}(i), \mathcal{S}(i), \mathcal{D}\}$, $F[x \mapsto n]$ represents the same function F except that $F(x)$ has been updated as $F(x) = n$. In addition, we also make use of the following auxiliary function to test if i depends on v_0:

$$\mathit{HasDep}(X) = \textbf{if } \exists\ (t, c) \in X\ s.t.\ t \neq \mathit{const} \textbf{ then } \mathit{true} \textbf{ else } \mathit{false} \qquad (2)$$

By convention, sp (fp) is the stack (frame) pointer register.

Let us examine the rules given in Figure 4, which are applied in line 9 of Algorithm 1. Two points are in order.

- We distinguish fp and sp from every other register $r \notin \{fp, sp\}$ so that $\mathcal{V}(i)(fp)$ and $\mathcal{V}(i)(sp)$ are always strongly updated and $\mathcal{V}(i)(r)$ is always weakly updated except in [MOV-RV-KILL], [MOV-RIV-KILL] and [MOV-RC-KILL]. This design decision allows us to keep track of only one stack frame instead of multiple stack frames when analyzing a call in order to achieve efficiency at some slight loss of precision for the slice obtained. In practice, most COTS C++ programs are compiled without turning on a so-called frame-pointer omission flag, which can be checked easily. Consider the binaries on x86 produced by the Microsoft Visual C++ compiler for Windows. If a function's prologue and epilogue are of the form "`push sp; mov fp,sp`" and "`leave; ret`", respectively, then its `/Oy` (frame-pointer omission) flag is off. If we see something like "`sub sp, ...`" and "`add sp, ...; ret`", then `/Oy` is on. When `/Oy` is on (causing fp to be used as a general register), we can modify each rule by simply changing $r \in \{fp, sp\}$ (if it exists) to $r \in \{sp\}$ in its premises.
- When reaching a call instruction (flagged by IDA Pro [25] as discussed in Section IV), we record the address of its ensuing instruction as a return address. As discussed earlier, we handle a call as a **push** and a **use** (i.e., **jmp**) in our formalism. When reaching a return instruction (modeled as a **pop** and a **use**, i.e., **jmp**) in analyzing the function called, we will continue to analyze the instruction marked by the previously recorded return address. Thus, TSLICE finds an inter-procedural slice context-sensitively.

We go through our four groups of rules, [MOV-*], [OP-*], [STK-*] and [USE-*], for handling four types of instructions in our language. In our rules, $\mathcal{S}(i)$ is only updated in [MOV-SR], [OP-SR], as well as [STK-*]s, $\mathcal{D}(i)$ is updated in the majority of the rules similarly except that i is checked to see if it becomes now dependent on v_0 (if it is not before), and $\mathcal{V}(i)$ is updated also in the majority of the rules but differently, depending on the nature of instruction i being analyzed. Therefore, we focus naturally more on describing how $\mathcal{V}(i)$ is updated (i.e., how each register is updated) below.

- $\boxed{mov\ opr^1,\ opr^2}$. All the *mov* instructions are handled by the 14 [MOV-*] rules. [MOV-RV] is simple. $\mathcal{V}(i)$ is updated to reflect the fact that r may now also contain a pointer to $v_0 + c$, and consequently, instruction i is now known to depend on v_0. [MOV-RIV] is similar except that r contains (ref, c), i.e., $*(v_0 + c)$ (known as a reference to v_0). [MOV-RR] handles a register move instruction i of the form "$mov\ r_1, r_2$", by updating $\mathcal{V}(i)(r_1)$ with $\mathcal{V}(i)(r_2)$ and making i dependent on v_0 (if it is not yet) as long as r_2 contains any v_0-dependent value. Consider [MOV-RI] now. If r_2 contains (ptr, c'), i.e., a pointer to v_0, r_1 is added with $(ref, c + c')$, i.e., a reference to v_0 after the update. If r_2 contains (ref, c'), i.e., a reference to v_0, r_1 is added with $(other, *)$. If r_2 contains $(other, *)$ already, $(other, *)$ is ignored to reduce the number of irrelevant instructions added to $\mathscr{S}_{v_0}$. [MOV-RS] and [MOV-SR] propagate the value information between a stack slot and a register r in either direction, except that $r \notin \{fp, sp\}$ holds in [MOV-RS] and $\mathcal{S}(i)$ is updated in [MOV-SR]. [MOV-RC] recognizes that $r \notin \{fp, sp\}$ may now also contain a constant c. [MOV-RC-1] behaves similarly for $r \in \{fp, sp\}$ except that a strong update to $\mathcal{V}(i)(r)$ is performed. [MOV-FP] handles an assignment of sp to fp while [MOV-SP] handles an assignment of fp to sp. In both cases, a strong update is performed. [MOV-DR] says that an instruction that writes into a v_0-dependent address depends on v_0. Finally, [MOV-RV-KILL], [MOV-RIV-KILL] and [MOV-RC-KILL] perform a strong update to $r \in \{sp, fp\}$ as described.
- $\boxed{op_{\oplus}\ opr^k,\ opr^2}$. The binary arithmetic instructions are handled by the seven [OP-*] rules. [OP-RC] is similar to [MOV-RC] except that we must now account for the semantics of $\oplus$. Similarly, [OP-RC-1] is an analogue of [MOV-RC-1]. For an instruction of the form "$op_{\oplus}\ r_1, r_2$", [OP-RR] updates $\mathcal{V}(i)(r_1)$ only when either r_1 or r_2 contains a constant (in order to keep $\mathscr{S}_{v_0}$ small and relevant), and additionally, [OP-RREF] propagates a reference or $(other, *)$ conservatively as $(other, *)$ from r_2 to r_1. [OP-RI] is a (conservative) analogue of [MOV-RI], as it handles an $op_{\oplus}$ instead of a *mov* instruction. Finally, [OP-RS] ([OP-SR]) is an analogue of [MOV-RS] ([MOV-SR]), except that it is much more conservative. Let us examine [OP-RS] for handling "$op_{\oplus}\ r, [fp + c]$". If $[fp + c]$ contains a v_0-dependent value, then r will contain an over-approximation of that value as $(other, *)$. If $[fp+c]$ contains a constant $(const, c)$, r will not be made to contain also $(const, c' + c)$, even if r contains $(const, c')$, for two reasons. First, $(const, c' + c)$ often results in no extra relevant instructions added to $\mathscr{S}_{v_0}$ than $(const, c')$ already does. Second, tracking only relevant constants makes TSLICE lightweight.
- $\boxed{push/pop\ r}$. These two stack manipulating instructions are straightforward. [STK-PUSH] stores every pushed value at the top stack location and increases sp by 1. [STK-POP] handles the information flow in the opposite direction.
- $\boxed{use\ \ldots opr^k \ldots}$. All the *use* instructions are handled by one single rule, [USE-DEP], to update $\mathcal{D}(i)$ only.

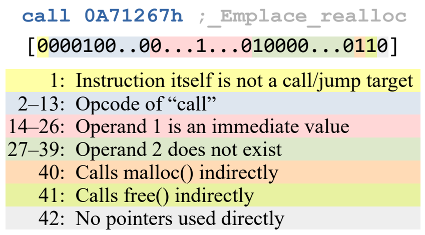
There is no rule for $mov\ [fp+c'], c$, as it can be modeled as a sequence of two instructions, $mov\ r, c$ and $mov\ [fp+c'], r$.

B. Type Classification

Given a slice $\mathscr{S}_{v_0}$, we have designed a GCN-based classifier to infer its type. In Section III-B1, we describe how to encode each node (i.e., instruction) in a slice with a feature vector in order to find a feature vector representation for the entire slice. In Section III-B2, we introduce a GCN-based classifier.

1) Encoding Instructions with Feature Vectors: To apply a GCN [23], [24] to turn a slice into a graph representation characterized by a feature vector, we first encode each node in the slice, which contains instruction i, as a 42-dimensional feature vector, according to the following seven features:

- F_i^1: Whether i is a direct target of a call/jump or not.
- F_i^2: A 12-bit binary representation of the opcode of i. One-hot encoding is not used as there are more than 1700



```
call 0A71267h ;_Emplace_realloc
[0000100..00...1...010000...0110]
```

1:	Instruction itself is not a call/jump target
2–13:	Opcode of “call”
14–26:	Operand 1 is an immediate value
27–39:	Operand 2 does not exist
40:	Calls malloc() indirectly
41:	Calls free() indirectly
42:	No pointers used directly

Fig. 5. An example for encoding a `call` instruction.

opcodes. Since the opcodes with similar semantics are close together (e.g., `push`/`pushaw`/`pusha` assigned with 143/144/ 145), our opcode representation is preferred.

- F_i^3 (1st operand) and F_i^4 (2nd operand): A 13-bit one-hot encoding of the operand type of every such an operand in i. There are 13 operand types: nil operand, register, direct memory reference, memory reference with base and index registers, memory reference with base and index registers plus a displacement, immediate value, immediate far address, immediate near address, and five additional processor-specific types provided by IDA Pro [25].
- F_i^5: Whether i calls a heap allocation function (e.g., `malloc()`) directly or indirectly or not (along a call chain).
- F_i^6: Whether i calls a heap free function (e.g., `free()`) or not.
- F_i^7: The number of levels of pointer indirections for using v_0 in i represented by an integer (as i depends on v_0).

Figure 5 illustrates how a `call` instruction (identified by IDA Pro [25] as discussed in Section IV) is encoded.

2) A GCN-based Classifier: We first describe how to obtain a GCN-based classifier during a training process. We then discuss how to use it to predict the type of a variable.

Training. Let $T = \{t_1, t_2, \ldots, t_n, t_{\text{primitive}}\}$ be a set of types to recover from C++ binaries, where $t_1, t_2, \ldots, t_n$ are candidate container types and $t_{\text{primitive}}$ represents any of all the possible primitive types (which are not distinguished as discussed in Section I). Let $\mathcal{G} = \{G_1, G_2 \ldots, G_m\}$ be a set of slices with each slice of a variable address being labeled by a unique type $t \in T$, implying that the variable is of type t or a pointer to t (with one or more levels of indirections). Our objective is to learn a classifier to predict the type of G_i.

We have designed a GCN [23] to learn a graph representation h_G of a slice G in a message-passing manner. Each node $v \in G$ starts with X_v encoded as per Section III-B1:

$$h_v^{(0)} = X_v \tag{3}$$

We then perform k iterations of aggregation along the edges in G. We update the representation of a node by aggregating its representation with those of its predecessor neighboring nodes by using the element-wise *mean* pooling mechanism:

$$h_v^k = \text{ReLU}(W^k \times \sum_{u \in \mathcal{N}(v) \cup \{v\}} \frac{h_u^{k-1}}{|\mathcal{N}(v) \cup \{v\}|}) \tag{4}$$

where $N(v)$ gives the set of predecessor nodes of v in G.

At the end of the k-th iteration, we use a simple readout function to obtain (with $\mathcal{V}$ being the set of nodes in G):

$$h_G = \sum_{v \in \mathcal{V}} h_v \tag{5}$$

Finally, we connect the output h_G to a linear transformation layer and apply softmax to estimate the probabilities for G to have different types in T. Thus, we can choose the predicted type, $\hat{y}_G$, as the one with the largest probability:

$$\hat{y}_G = \underset{t_i \in T}{\text{argmax}}(\text{softmax}(W^L h_G)_i) \tag{6}$$

Note that W^L above and W^k given in (4) are the parameters in the prediction model learned during the training process.

Prediction. Given a variable address, the slice found by TSLICE will be fed into our classifier to predict its type.

IV. Evaluation

TIARA is the first to infer container types in COTS C++ binaries (Section I). Due to the lack of earlier tools to compare with, we focus on demonstrating that TIARA is effective in recovering container types by answering four RQs:

- RQ1: Is TIARA effective in identifying container types by restricting training and testing to the same project?
- RQ2: Is TIARA effective in identifying container types by performing training and testing in different projects?
- RQ3: Is TIARA more effective when compared with a version of TIARA that uses a simple slicer (due to the lack of an open-source slicer for C++ binaries)?
- RQ4: Is TIARA efficient in slicing and training?

COTS C++ Binaries. We consider the binaries in the Microsoft Portable Executable (PE) format targeting x86, which are generated by using the Microsoft Visual C++ 15 2017 toolchain (abbreviated to MSVC). For a binary, we use IDA Pro [25] to disassemble it and find the information required by TIARA. In type-relevant slicing, we find its entry point I_0 as needed. For type classification, we need to encode each instruction in terms of a feature vector (Section III-B1). We use IDA Pro to find the functions calling `malloc()` and `free()` (possibly indirectly). When it fails to provide the information of a particular feature for an instruction, the default 0 is used.

Benchmarks. We consider eight programs from eight projects (Table I), which mostly include more than one executable. We have selected only one program from each project in order to prevent code duplications due to reasons like static linking. We have compiled these programs with the “release mode” settings, by enabling the most aggressive optimization (/O2), to simulate how COTS programs are compiled when released.

For the eight programs selected, Table I gives their binary sizes and the number of variable addresses having type t or a pointer type to t (with one or more levels of indirections), where t is one of the four types considered, `std::list`, `std::vector`, `std::map`, and `primitive` (representing all possible primitive types). We have selected `std::list`, `std::vector` and `std::map`, since they are, respectively,

TABLE I
BENCHMARK STATISTICS FOR THE COTS BINARIES (WITH THE NUMBER OF VARIABLES ADDRESSES OF A PARTICULAR TYPE GIVEN).

#	GitHub Repository	Version	Program Name	Binary Size	#std::list	#std::vector	#std::map	#primitive
1	llvm/llvm-project	3.8.0	clang	47 M	89	1685	2446	98495
2	Kitware/CMake	v3.17	cmake	9.4 M	25	2340	2074	35676
3	bitcoin/bitcoin	v0.18	bitcoind	5.4 M	23	1042	1125	30023
4	gabime/spdlog	v1.x	spdlog-utests	1.6 M	3	106	51	4600
5	SOCI/soci	3.2.3	soci_odbc_test_mysql_static	1.3 M	0	123	117	4353
6	google/re2	2020-04-01	filtered_re2_test	410 k	2	76	89	1137
7	bblanchon/ArduinoJson	6.x	TextFormatterTests	484 k	0	41	72	1377
8	List Extension	11/04/2016	list_ext	76 k	47	6	0	227

the representatives of non-contiguous sequential, contiguous sequential, and associative containers, the three common STL container categories in C++ programs. The first seven are the popular C++ programs from GitHub. However, `std::list` is not as frequently used as the other two due to the reasons explained by Stroustrup [26]. As the first seven programs contain relatively few variables of type `std::list`, we have added an eighth program, called `list_extension`, which contains a few list-related code snippets taken directly from a Microsoft documentation [27], resulting in an increase of `std::list`-related variables by 33%.

For a binary, we find its addresses representing variables by using Microsoft Debug Interface Access SDK [28]. We then apply TIARA to predict their types. For a COTS binary without debugging information, its variable addresses must be detected orthogonally. However, finding such addresses is much less challenging than finding their types [29].

To summarize, we use `std::vector`, `std::list`, and `std::map`. to measure the effectiveness of TIARA. $T = \{t_{\texttt{list}}, t_{\texttt{vector}}, t_{\texttt{map}}, t_{\text{primitive}}\}$ is the set of type labels used.

Training. The GCN used in TIARA is made up of two graph convolutional layers of size 64 each. This GCN is implemented in terms of Deep Graph Library [30] with PyTorch [31] as the back-end. We train it by using the Adam algorithm [32] as the optimizer with the cross-entropy loss function. We have used a learning rate of 0.001 and an epoch size of 300. In TIARA, data labeling is fully automatic. When labeling a slice constructed for a variable, we use Microsoft Debug Interface Access SDK [28] to find its type automatically.

When evaluating TIARA in recovering container types within a project (RQ1) and across the projects (RQ2), we will discuss how their training and testing programs are selected.

Metrics. We evaluate TIARA by considering three metrics: precision, recall and F1 score. *Precision* is the percentage of variables with a correctly inferred type among all the variables that are inferred to have that type. *Recall* is the percentage of variables with a correctly inferred type among all the variables that actually have that type. Finally, F1 score is the harmonic mean of *Precision* and *Recall*. We evaluate the efficiency of TIARA by considering its slicing and training times.

Computing Platforms. TIARA has two stages (Figure 3). Its type-relevant slicing stage (TSLICE) runs on a Windows 10 desktop containing an Intel Core i9-10900X CPU of 3.70 GHz with 64G memory. Its type classification stage runs on a Ubuntu server containing two Intel Xeon CPUs of 2.6 GHz with 128G memory, accelerated by a 16G Tesla P100 GPU.

Results. Table II gives the results for addressing RQ1 – RQ3 (with each row representing one independent experiment to be explained when we discuss these RQs). In addition, Table III contains some additional results for addressing RQ3 only. Finally, Table IV gives the results for addressing RQ4.

A. RQ1: Intra-Project Type Prediction

To address RQ1, we report and analyze the results of five independent experiments listed in five rows marked as I1a – I5a in Table II. In each experiment, the project(s) considered are given. In each experiment, the data, which is given as a set of variable addresses with their associated types in $T = \{t_{\texttt{list}}, t_{\texttt{vector}}, t_{\texttt{map}}, t_{\text{primitive}}\}$ (Table I), is divided into a training set and a testing set. The ratio of training over testing samples is $4 : 1$ (with both randomly selected).

According to the results reported in I1a – I5a of Table II, TIARA is highly effective, achieving the precision (ranging from 0.86 to 1.00) with an average of 0.94, the recall (ranging from 0.79 to 1.00) with an average of 0.89, and the F1 scores (ranging from 0.88 to 0.98) with an average of 0.91 across all the variables of the four types in the five experiments. TIARA's effectiveness is also revealed by the average precision, recall and F1 score both across the four types for an experiment and across the five experiments for a given type.

Several observations are in order. First, TIARA can identify all the three container types equally well (as reflected by the average precision, recall and F1 score for each container type across the five experiments). Second, TIARA achieves a precision or recall of 1.0 in five cases when its corresponding samples are small (Table I). This explains why TIARA is more effective in I4a – I5a than in I1a – I3a for `std::list`. Third, TIARA can recover primitive types well, due to (1) a relatively large number of samples available for primitives (Table I), (2) relatively smaller slices found (Table III), and (3) the fact that different primitive types are not distinguished. Finally, TIARA loses some precision since the compiler may optimize variables of different types that do not have overlapping scopes to share the same stack slot, i.e., the same binary address.

B. RQ2: Cross-Project Type Prediction

To address RQ2, we report and analyze the results of four independent experiments listed in four rows marked as C6a – C9a in Table II. In each experiment, we simulate real-world inference scenarios by performing training in one set

TABLE II
EXPERIMENTAL RESULTS FOR RQ1–RQ3. FOR ALL METRICS, LARGER IS BETTER.

	#	Training Data	Testing Data	`std::list`			`std::vector`			`std::map`			Primitive			Macro Average		
				Pr.	*Re.*	F_1	*Pr.*	*Re.*	F_1	*Pr.*	*Re.*	F_1	*Pr.*	*Re.*	F_1	*Pr.*	*Re.*	F_1
Intra-Program	I1a	clang		0.92	0.80	0.86	0.87	0.76	0.82	0.95	0.79	0.86	0.98	0.99	0.99	0.93	0.84	0.88
	I1b			0.50	0.33	0.40	0.57	0.32	0.41	0.82	0.78	0.80	0.98	0.99	0.98	0.72	0.60	0.65
	I2a	cmake + list_ext		1.00	0.85	0.92	0.89	0.84	0.87	0.93	0.80	0.86	0.94	0.98	0.96	0.94	0.87	0.90
	I2b			0.50	0.75	0.60	0.46	0.56	0.51	0.75	0.74	0.74	0.94	0.93	0.94	0.66	0.75	0.70
	I3a	bitcoind + list_ext		0.86	0.92	0.89	0.91	0.88	0.89	0.95	0.81	0.87	0.95	0.98	0.97	0.92	0.90	0.90
	I3b			1.00	0.50	0.67	0.58	0.44	0.50	0.80	0.89	0.84	0.97	0.97	0.97	0.84	0.70	0.74
	I4a	spdlog + list_ext		0.90	1.00	0.95	0.94	0.89	0.91	0.90	1.00	0.95	0.99	0.99	0.99	0.93	0.97	0.95
	I4b			0.60	1.00	0.75	0.27	0.50	0.35	0.75	0.33	0.46	0.97	0.96	0.97	0.65	0.70	0.63
	I5a	soci_odbc_test_static + list_ext		1.00	0.88	0.93	1.00	0.80	0.89	1.00	0.81	0.89	0.96	1.00	0.98	0.99	0.87	0.92
	I5b			0.75	1.00	0.86	0.38	0.50	0.43	0.68	0.81	0.74	0.96	0.91	0.93	0.69	0.80	0.74
		Average (TIARA)		0.94	0.89	0.91	0.92	0.83	0.88	0.95	0.84	0.89	0.96	0.99	0.98	0.94	0.89	0.91
		Average (TIARA$_{\text{SSLICE}}$)		0.67	0.72	0.66	0.45	0.46	0.44	0.76	0.71	0.72	0.96	0.95	0.96	0.71	0.71	0.69
Cross-Program	C6a	clang+cmake+bitcoind	all - {clang+cmake+bitcoind}	0.85	0.80	0.84	0.94	0.74	0.83	0.90	0.82	0.86	0.86	0.95	0.90	0.89	0.83	0.86
	C6b			0.48	0.30	0.37	0.44	0.48	0.46	0.90	0.83	0.86	0.88	0.91	0.90	0.68	0.63	0.65
	C7a	all - clang	clang	0.85	0.73	0.79	0.78	0.74	0.76	0.77	0.82	0.79	0.95	0.96	0.95	0.84	0.81	0.82
	C7b			0.00	0.00	N/A	0.32	0.51	0.40	0.78	0.88	0.83	0.95	0.90	0.92	0.51	0.57	0.54
	C8a	all - cmake	cmake	0.61	0.59	0.60	0.94	0.63	0.75	0.93	0.76	0.83	0.74	0.97	0.84	0.81	0.74	0.76
	C8b			0.39	0.43	0.41	0.58	0.37	0.45	0.74	0.84	0.79	0.84	0.89	0.87	0.64	0.63	0.63
	C9a	all - bitcoind	bitcoind	0.71	0.67	0.69	0.92	0.70	0.79	0.91	0.70	0.79	0.67	0.95	0.79	0.80	0.75	0.76
	C9b			0.39	0.50	0.44	0.73	0.48	0.58	0.75	0.70	0.72	0.76	0.85	0.80	0.66	0.63	0.64
		Average (TIARA)		0.76	0.70	0.73	0.90	0.70	0.78	0.88	0.78	0.82	0.81	0.96	0.87	0.84	0.78	0.80
		Average (TIARA$_{\text{SSLICE}}$)		0.32	0.31	0.31	0.52	0.46	0.47	0.79	0.81	0.80	0.86	0.89	0.87	0.62	0.62	0.62

of projects but testing in all the remaining ones given in our benchmark suite (Table I). For example, in row C7a, `clang` is the testing project and `all-clang` indicates that the remaining projects are used as the training programs.

According to the results in Table II, TIARA is effective, achieving the precision (ranging from 0.61 to 0.95) with an average of 0.84, the recall (ranging from 0.59 to 0.97) with an average of 0.79, and the F1 scores (ranging from 0.60 to 0.95) with an average of 0.80 across all the variables of all the four types in the four experiments.

When comparing C6a – C9a (for cross-project type recovery) with I1a – I5a (intra-project type recovery) in Table II, we find that TIARA is only slightly less effective, with the average precision, recall and F1 score dropping from 0.94, 0.89 and 0.91 to 0.84, 0.78 and 0.80, respectively (calculated again across all the four types in all their respective experiments). The slight performance degradation is as expected since, for example, different coding styles and conventions in different projects will lead to different program behaviors in their binaries. Given this, TIARA is considered to be effective in recovering container types in real-world COTS binaries.

C. RQ3: Comparing with the State of the Art

To the best of our knowledge, TIARA is the first tool for recovering container types in C++ binaries. To evaluate TIARA against the state of the art, we compare TIARA with a version of TIARA in which a different slicing algorithm is used. However, we are not aware of any open-source tool for computing inter-procedural slices in COTS binaries. BEST [33], which targets PowerPC binaries, is designed to estimate statically the WCET (Worst-Case Execution Times) of a program and is thus limited to single-function programs.

Therefore, we have decided to compare TIARA with a version of TIARA, denoted TIARA$_{\text{SSLICE}}$, where TSLICE has been replaced by a simple slicer, named SSLICE. Given a variable address v_0, SSLICE produces a slice consisting of all the instructions in the function that contains the first access to v_0 and all the instructions in its *directly* called functions. This comparison is reasonable, as existing scalable binary slicers are very imprecise, often producing a slice that includes nearly the entire program [22]. Table III compares the average slice sizes obtained by TSLICE and SSLICE in terms of average node and edge counts. For each type in $T = \{t_{\texttt{list}}, t_{\texttt{vector}}, t_{\texttt{map}}, t_{\text{primitive}}\}$, the number of slices produced by each slicer is the same as the number of variable addresses in Table I. TSLICE is lightweight, producing one slice in 0.2 seconds, on average. The slices found by TSLICE are one order (two orders) of magnitude smaller than those found by SSLICE for a container (primitive) type, one average.

TABLE III
AVERAGE SLICE SIZES PRODUCED BY TSLICE AND SSLICE.

	SSLICE		TSLICE	
Type	#Nodes	#Edges	#Nodes	#Edges
std::list	1873.41	2055.12	68.39	95.53
std::vector	2518.07	2769.04	64.01	86.58
std::map	860.30	945.50	54.04	71.64
primitive	1537.04	1676.24	12.73	17.86

In Table II, we have compared TIARA (rows I1a – I5a and C6a – C9a) with TIARA$_{\text{SSLICE}}$ (rows I1b – I5b and C6b – C9b). Except for a few cases highlighted by red boxes, TIARA$_{\text{SSLICE}}$ is substantially less effective than TIARA, resulting in significantly lower average precision, recall and F1 score (both across the different types for an experiment and across the different experiments for a given type).

These results show that, given a variable address, the slice produced by TSLICE, while substantially smaller than that from SSLICE (in general), contains still enough type-relevant instructions for characterizing its type (Figure 2). Consider the extreme but illuminating case reported in C7a and C7b of Table II when inferring `std::list` from the 89 variables in `clang` by training with the 100 `std::list`-related samples in `all-clang` (Table I). As the ratio of training over testing samples is low, the impact of the imprecision of SSLICE on the effectiveness of TIARA$_{\text{SSLICE}}$ is maximally exposed. TIARA$_{\text{SSLICE}}$ failed to make any correct prediction, producing the following classification for the 89 variables in `clang`: 53 of `std::vector`, 3 of `std::map`, and 33 of primitives. In contrast, TIARA has achieved a precision, recall and F1 score of 0.85, 0.73, and 0.79, respectively, predicting correctly 65 out of the 89 variables are typed `std::list`.

D. RQ4: Efficiency

TIARA, shown in Table IV, is highly efficient. The average slicing time per intra-project (cross-project) experiment is 2 hours (10 hours, identically for each cross-project experiment as all programs are involved). The average training time per intra-project (cross-project) experiment is 5.1 (20.5) minutes. On the other hand, $\text{TIARA}_{\text{SSLICE}}$, which is substantially less effective in recovering container types than TIARA (Table II), is even slightly slower overall. The average slicing time per intra-project (cross-project) experiment is 1.7 hours (8.6 hours). However, due to the larger slices produced, its average training time per intra-project (cross-project) experiment is now much longer, reaching 11.4 (94.5) minutes.

TABLE IV
THE EFFICIENCY OF TIARA AND $\text{TIARA}_{\text{SSLICE}}$.

		I1a	I2a	I3a	I4a	I5a	C6a	C7a	C8a	C9a
Slicing	TIARA	5.5 h	2.2 h	1.8 h	0.3 h	0.3 h		10 h		
	$\text{TIARA}_{\text{SSLICE}}$	4.7 h	1.8 h	1.5 h	0.2 h	0.2 h		8.6 h		
Training	TIARA	14.4 m	5.6 m	4.5 m	0.4 m	0.4 m	25.5 m	12.2 m	22 m	22.3 m
	$\text{TIARA}_{\text{SSLICE}}$	34.6 m	11.4 m	8.9 m	1.6 m	0.7 m	1.4 h	2.5 h	1.2 h	1.2 h

V. RELATED WORK

We review both rule- and machine-learning-based approaches for performing binary analysis in the literature.

Rule-based Binary Type Inference. While there are many previous efforts [34]–[36] on inferring primitive types, recursive types, polymorphic types, variables, and function prototypes from binaries, we review only a few related ones on class type recovery. In C++ binaries, C++ classes may leave some class-specific clues such as the "this" pointer and virtual function tables (VFTs). SmartDec [13] exploits the "this" pointer to construct class hierarchies from binaries. `vfGuard` [37] and `VTint` [38] reconstruct VFTs by using the dynamic dispatch mechanism in C++. The run-time type information (RTTI) in C++ has also been leveraged [17] to find class hierarchies and member functions/variables. OBJDIGGER [15] recovers objects by tracking the "this" pointer, by combining symbolic execution and inter-procedural data flow analysis.

C++ templates support compile-time polymorphism instead of run-time polymorphism. Existing efforts [13]–[19] for recovering polymorphic classes in C++ binaries cannot be applied to recover container classes (Section I).

OOANALYZER [20] recovers ordinary classes from C++ binaries by using a Prolog-based reasoning system. It distinguishes class types by distinguishing their related methods. However, as each method of a container type has different type-dependent instantiations in binaries, OOANALYZER can only recognize these instantiations as belonging to different classes without actually knowing what their types are.

TIARA (as proposed here) relies on a new type-relevant slicing algorithm for finding a small slice from a variable address in C++ binaries to predict its type. According to a recent study [22], the slices computed by state-of-the-art techniques for binaries are very imprecise. The only open-source slicer that we are aware of works only for PowerPC binaries, limited to small single-function programs only [33].

Machine-Learning-based Binary Analysis. Machine learning techniques are increasingly being used in reverse engineering and binary analysis. There are efforts on recovering partial source-level information in binaries, including functions [39]–[41], coding style and programmers' names [42], and their toolchains used [43]. EKLAVYA [44] utilizes an RNN to identify function signatures from binaries. These approaches tend to learn properties from blocks of binary codes, while TIARA aims to predict the type for a single address.

Katz et al. [18] use a Markov model to predict types from their object tracelets. Katz et al. [19] propose a variable-order Markov model to recover the class hierarchies from binaries. These techniques rely on the "this" pointer and VFTs to extract the function calls related to receiver objects.

DEBIN [21] focuses on predicting the primitive types of variables in COTS binaries compiled from C programs. It transforms binary programs into dependence graphs and trains a conditional random field (CRF) model from the graphs thus obtained. During the inference, the model is used to assign the types to unknown graph nodes to maximize the joint probability. DEBIN applies a global strategy to infer a variable's properties from its relationships with other variables. TypeMiner [45] recovers types in C binaries statically by relying on dependence analysis and a classification of execution traces of data objects. Like DEBIN, however, it focuses on recovering primitive types and does not handle template class types. TIARA, which focuses on inferring container types in C++ binaries, utilizes a local strategy to infer the type of a variable from its own behavior. TIARA is capable of obtaining a high accuracy with a small number of training binaries.

VI. CONCLUSION

In this paper, we introduce an effective approach, TIARA, for identifying container class types from COTS C++ binaries. TIARA consists of a new slicing algorithm for finding a type-relevant slice of a given variable in C++ binaries and a new GCN-based classifier that allows its container type to be predicted. Its type-relevant slicing algorithm can also be used as a stand-alone tool in detecting code clones [46], security vulnerabilities [4], [6], and software bugs [47], [48].

DATA AVAILABILITY STATEMENT

The experimental data and non-proprietary software for this article are openly available in Zenodo at: https://zenodo.org/record/5787482.

ACKNOWLEDGMENT

This work is supported by Australian Research Council (DP180104069 and DP210102409), National Natural Science Foundation of China (61872272 and 61640221), and the Strategic Priority Research Program of Chinese Academy of Science (XDC05040100).

APPENDIX

A. Abstract

Our artifact provides all non-proprietary components of TIARA. TSLICE runs as an IDAPython plugin of IDA Pro, which is proprietary, we cannot deploy it in the Docker image.

If you do not have access to IDA Pro, you can still reproduce our results by using the pre-computed slices from TSLICE and and the pre-trained models for all the experiments.

B. Description

The minimum requirement for using the artifact is:

- **Software:**
  - Docker
  - Microsoft Visual Studio (Optional)
  - IDA Pro + IDAPython (Optional)
- **Hardware:**
  - 64 GB or greater available storage
  - 32 GB or greater RAM

C. Installation

Download the Docker archive from the following URL:

- https://zenodo.org/record/5787482

Then import and run it:

```
# docker load --input tiara-artifact.tar.gz
# docker run -it tiara-artifact
```

This will create a Ubuntu 20.04 environment with a proper version of Python and other dependencies installed. The docker will start a shell in the virtual environment. All sliced graphs and scripts are in the directory `/app/tiara-artifact`.

D. Complete Experiment Workflow

Note that if you do not have access to IDA Pro or only want to evaluate the pre-trained models, simply go to the last step.

1) Compilation: Compile sample projects using Microsoft Visual Studio with the debugging flags (e.g., `/Zi`) turned on, to generate PDB files. The compilation commands used vary across different open-source C++ projects. Please refer to their documentation for a detailed description.

We will use `prog.exe` as an example to introduce the remaining steps.

2) Ground Truth Types from Debugging Information Extraction: For each compiled program, there is a corresponding PDB file generated by MSVC containing the debugging information. Extract the type information from it by using `dia2dump.exe`.

For example, `prog.pdb` is generated along with `prog.exe`. Start a Windows command prompt (cmd) and run the following command:

```
> dia2dump.exe prog.pdb -o prog.dump.csv
```

It extracts the necessary information in the given PDB file and saves it in the named CSV file.

3) Disassembling and Slicing: Start IDA Pro to disassemble the sample binary program. When you see "AU: idle" on the left-bottom corner of IDA Pro, load the IDAPython script `tslice/main.py`. Once the script has been loaded successfully, the "main.py loaded" message will be displayed on the console.

Type `start()` to compute slices with TSLICE or `start('noslice')` with SSLICE. IDA Pro will popup a message box "running Python script" and may look like seemingly getting stuck when it is actually busy with slicing.

When slicing is done, TSLICE and SSLICE will generate `prog.json` and `prog.noslice.json`, respectively.

4) Training and Test Data Generation: Transfer all JSON files to the machine where learning and prediction will be done.

For the inter-program experiments reported in Table II, data from several projects is used for training and/or testing purposes. We need to combine several files together. For example, to generate the training data for C6a, run:

```
# combine.py clang.json cmake.json bitcoind.json
  --mergeout out.json
```

It collects the graphs from the three JSON files and saves them in `out.json`.

For the intra-program experiments, we split the JSON files into the training and testing parts. For example, to generate the training and test data for I2a, run:

```
# combine.py cmake.json listExtension.json --split
  --trainout train.json --testout test.json
```

This will read the graphs from the two given JSON files and split them into the training and test data in two separate files.

5) Training and Testing: The following command trains a model using `A.json`, tests it with `B.json`, and generates a file named `model.pt`:

```
# train.py -t A.json -v B.json -m model.pt
```

To evaluate a model, run:

```
# eval.py -f X.json -m model.pt
```

To evaluate all pre-trained models, simply run the shell script:

```
# /app/tiara-artifact/eval-all.sh
```

The DGL Library may prompt to ask what valid backend to use. Type `pytorch` and hit enter to continue.

E. Evaluation and Expected Results

Evaluating this work by using the pre-trained models should reproduce all experimental results given in Table II. Similar results are expected to be obtained if new models are trained.

REFERENCES

[1] J. Caballero and Z. Lin, "Type inference on executables," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 65:1–65:35, 2016. [Online]. Available: http://doi.acm.org/10.1145/2896499

[2] J. Caballero, C. Grier, C. Kreibich, and V. Paxson, "Measuring pay-per-install: The commoditization of malware distribution," in *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*, 2011. [Online]. Available: http://static.usenix.org/events/sec11/tech/full_papers/Caballero.pdf

[3] J. Zeng, Y. Fu, K. A. Miller, Z. Lin, X. Zhang, and D. Xu, "Obfuscation resilient binary code reuse through trace-oriented programming," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, 2013, pp. 487–498. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516664

[4] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, 2011. [Online]. Available: http://www.isoc.org/isoc/conferences/ndss/11/pdf/5_1.pdf

[5] E. R. Jacobson, N. E. Rosenblum, and B. P. Miller, "Labeling library functions in stripped binaries," in *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools, PASTE'11, Szeged, Hungary, September 5-9, 2011*, 2011, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/2024569.2024571

[6] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*, 2010. [Online]. Available: http://www.isoc.org/isoc/conferences/ndss/10/pdf/23.pdf

[7] Z. Lin, J. Rhee, C. Wu, X. Zhang, and D. Xu, "Discovering semantic data of interest from un-mappable memory with confidence," in *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012. [Online]. Available: https://www.ndss-symposium.org/ndss2012/discovering-semantic-data-interest-un-mappable-memory-confidence

[8] D. Wu, L. Chen, Y. Zhou, and B. Xu, "An empirical study on the adoption of C++ templates: Library templates versus user defined templates," in *The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, July 1-3, 2013.*, 2014, pp. 144–149.

[9] C. Jung and N. Clark, "Ddt: design and evaluation of a dynamic program analysis for optimizing data structure usage," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 56–66.

[10] D. H. White and G. Lüttgen, "Identifying dynamic data structures by learning evolving patterns in memory," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2013, pp. 354–369.

[11] I. Haller, A. Slowinska, and H. Bos, "Mempick: A tool for data structure detection," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 479–480.

[12] K. Pei, J. Guan, M. Broughton, Z. Chen, S. Yao, D. Williams-King, V. Ummadisetty, J. Yang, B. Ray, and S. Jana, "Stateformer: fine-grained type recovery from binaries using generative state modeling," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 690–702.

[13] A. Fokin, E. Derevenetc, A. Chernov, and K. Troshina, "Smartdec: Approaching C++ decompilation," in *18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011*, 2011, pp. 347–356. [Online]. Available: https://doi.org/10.1109/WCRE.2011.49

[14] D. Dewey and J. T. Giffin, "Static detection of C++ vtable escape vulnerabilities in binary code," in *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012. [Online]. Available: https://www.ndss-symposium.org/ndss2012/static-detection-c-vtable-escape-vulnerabilities-binary-code

[15] W. Jin, C. Cohen, J. Gennari, C. Hines, S. Chaki, A. Gurfinkel, J. Havrilla, and P. Narasimhan, "Recovering C++ objects from binaries using inter-procedural data-flow analysis," in *Proceedings of the 3rd ACM SIGPLAN Program Protection and Reverse Engineering Workshop 2014, PPREW 2014, January 25, 2014, San Diego, CA*, 2014, pp. 1:1–1:11. [Online]. Available: http://doi.acm.org/10.1145/2556464.2556465

[16] V. Srinivasan and T. W. Reps, "Recovery of class hierarchies and composition relationships from machine code," in *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, 2014, pp. 61–84. [Online]. Available: https://doi.org/10.1007/978-3-642-54807-9_4

[17] K. Yoo and R. Barua, "Recovery of object oriented features from C++ binaries," in *21st Asia-Pacific Software Engineering Conference, APSEC 2014, Jeju, South Korea, December 1-4, 2014. Volume 1: Research Papers*, 2014, pp. 231–238. [Online]. Available: https://doi.org/10.1109/APSEC.2014.44

[18] O. Katz, R. El-Yaniv, and E. Yahav, "Estimating types in binaries using predictive modeling," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, R. Bodík and R. Majumdar, Eds. ACM, 2016, pp. 313–326. [Online]. Available: https://doi.org/10.1145/2837614.2837674

[19] O. Katz, N. Rinetzky, and E. Yahav, "Statistical reconstruction of class hierarchies in binaries," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, X. Shen, J. Tuck, R. Bianchini, and V. Sarkar, Eds. ACM, 2018, pp. 363–376. [Online]. Available: https://doi.org/10.1145/3173162.3173202

[20] E. J. Schwartz, C. F. Cohen, M. Duggan, J. Gennari, J. S. Havrilla, and C. Hines, "Using logic programming to recover c++ classes and methods from compiled executables," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 426–441. [Online]. Available: https://doi.org/10.1145/3243734.3243793

[21] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, "Debin: Predicting debug information in stripped binaries," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 1667–1680. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243866

[22] V. Srinivasan and T. W. Reps, "An improved algorithm for slicing machine code," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, E. Visser and Y. Smaragdakis, Eds. ACM, 2016, pp. 378–393. [Online]. Available: http://doi.acm.org/10.1145/2983990.2984003

[23] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. [Online]. Available: https://openreview.net/forum?id=SJU4ayYgl

[24] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=ryGs6iA5Km

[25] H. Rays, "Ida pro," https://www.hex-rays.com/, 2020.

[26] B. Stroustrup, "Why you should avoid linked lists," https://www.youtube.com/watch?v=YQs6IC-vgmo, 2012.

[27] Microsoft, "List classes," https://docs.microsoft.com/en-us/cpp/standard-library/list-class).

[28] Microsoft. (2020) Debug interface access SDK. [Online]. Available: https://docs.microsoft.com/en-us/visualstudio/debugger/debug-interface-access/

[29] J. Lee, T. Avgerinos, and D. Brumley, "TIE: principled reverse engineering of types in binary programs," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011. [Online]. Available: https://www.ndss-symposium.org/ndss2011/tie-principled-reverse-engineering-of-types-in-binary-programs

[30] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang, Q. Guo, H. Zhang, H. Lin, J. Zhao, J. Li, A. J. Smola, and Z. Zhang, "Deep graph library: Towards efficient and scalable deep learning on graphs," *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019. [Online]. Available: https://arxiv.org/abs/1909.01315

[31] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[32] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[33] A. Mangean, J. Béchennec, M. Briday, and S. Faucou, "BEST: a binary executable slicing tool," in *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, ser. OASICS, M. Schoeberl, Ed., vol. 55. Schloss

Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 7:1–7:10. [Online]. Available: https://doi.org/10.4230/OASIcs.WCET.2016.7

[34] K. Elwazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua, "Scalable variable and data type detection in a binary rewriter," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, 2013, pp. 51–60. [Online]. Available: http://doi.acm.org/10.1145/2462156.2462165

[35] E. Robbins, A. King, and T. Schrijvers, "From minx to minc: semantics-driven decompilation of recursive datatypes," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, R. Bodík and R. Majumdar, Eds. ACM, 2016, pp. 191–203. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837633

[36] M. Noonan, A. Loginov, and D. Cok, "Polymorphic type inference for machine code," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, C. Krintz and E. Berger, Eds. ACM, 2016, pp. 27–41. [Online]. Available: http://doi.acm.org/10.1145/2908080.2908119

[37] A. Prakash, X. Hu, and H. Yin, "vfguard: Strict protection for virtual function calls in COTS C++ binaries," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015. [Online]. Available: https://www.ndss-symposium.org/ndss2015/vfguard-strict-protection-virtual-function-calls-cots-c-binaries

[38] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song, "Vtint: Protecting virtual function tables' integrity." in *NDSS*, 2015.

[39] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: learning to recognize functions in binary code," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, K. Fu and J. Jung, Eds. USENIX Association, 2014, pp. 845–860. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/bao

[40] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, J. Jung and T. Holz, Eds. USENIX Association, 2015, pp. 611–626. [Online]. Available: https://www.usenix.org/conference/

[47] D. Ye, Y. Sui, and J. Xue, "Accelerating dynamic detection of uses of undefined values with static value-flow analysis," in *12th Annual IEEE/ACM International Symposium on Code Generation and*

usenixsecurity15/technical-sessions/presentation/shin

[41] S. Wang, P. Wang, and D. Wu, "Semantics-aware machine learning for function recognition in binary code," in *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 2017, pp. 388–398. [Online]. Available: https://doi.org/10.1109/ICSME.2017.59

[42] A. Caliskan-Islam, F. Yamaguchi, E. Dauber, R. Harang, K. Rieck, R. Greenstadt, and A. Narayanan, "When coding style survives compilation: De-anonymizing programmers from executable binaries," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, 18th February - 21th February 2018*, 2018.

[43] N. E. Rosenblum, B. P. Miller, and X. Zhu, "Recovering the toolchain provenance of binary code," in *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, M. B. Dwyer and F. Tip, Eds. ACM, 2011, pp. 100–110. [Online]. Available: https://doi.org/10.1145/2001420.2001433

[44] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 99–116. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/chua

[45] A. Maier, H. Gascon, C. Wressnegger, and K. Rieck, "Typeminer: Recovering types in binary programs using machine learning," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2019, pp. 288–308.

[46] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470 – 495, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642309000367

Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014, D. R. Kaeli and T. Moseley, Eds. ACM, 2014, p. 154. [Online]. Available: https://dl.acm.org/citation.cfm?id=2544154

[48] Y. Sui, D. Ye, Y. Su, and J. Xue, "Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions," *IEEE Trans. Reliab.*, vol. 65, no. 4, pp. 1682–1699, 2016. [Online]. Available: https://doi.org/10.1109/TR.2016.2570538