TypeMiner: Recovering Types in Binary Programs using Machine Learning

Alwin Maier, Hugo Gascon, Christian Wressnegger, and Konrad Rieck

Institute of System Security, TU Braunschweig, Germany

Abstract. Closed-source software is a major hurdle for assessing the security of computer systems. In absence of source code, it is particularly difficult to locate vulnerabilities and malicious functionality, as crucial information is removed by the compilation process. Most notably, binary programs usually lack type information, which complicates spotting vulnerabilities such as integer flaws or type confusions dramatically. Moreover, data types are often essential for gaining a deeper understanding of the program logic. In this paper we present TypeMiner, a static method for recovering types in binary programs. We build on the assumption that types leave characteristic traits in compiled code that can be automatically identified using machine learning starting at usage locations determined by an analyst. We evaluate the performance of our method with 14 real world software projects written in C and show that it is able to correctly recover the data types in 76 %–93 % of the cases.

Keywords: Reverse Engineering · Static Analysis · Classification

1 Introduction

The analysis of binary programs is a challenging yet necessary task in computer security. A large fraction of the software deployed in current computer systems is only available in binary form. This includes popular desktop applications, such as Microsoft Office and Acrobat Reader, as well as widespread firmware for networking devices, such products from Cisco, Juniper and Huawei. Without access to the source code, the only option for assessing the security of this software is to analyze the compiled code and recovering parts of its inner workings. In the past, such reverse-engineering effort has successfully uncovered several striking vulnerabilities and backdoors in popular software products [e.g., 6, 8].

However, reverse code engineering is a strenuous effort. The compilation and building process of software does not only translate high-level languages to machine instructions, but also obstructs access to information essential for analyzing security. For example, symbols are usually stripped from binary programs, rendering a direct analysis of variables and their types impossible. While there exist some approaches that can compensate this lack of information, such as fuzzing [2, 30, 33, 36] and symbolic execution [7, 18, 32, 38], several classes of vulnerabilities can only be systematically investigated, if type information is restored, as for example integer flaws [3, 10, 40, 41, 43] and type confusions [12, 19, 25].

Similarly, the analysis of backdoors and malicious functionality requires a deeper understanding of binary code which is hard to attain without detailed information on internal structures and used pointers.

Current approaches for uncovering types in binary programs mainly rest on three strategies: (a) the dynamic analysis of memory access patterns [e.g., 4, 13], (b) the propagation of type information from known sources [e.g., 26, 27], and (c) the identification of types using manually designed rules [14]. While all three strategies help to alleviate the problem of missing type information, their applicability is limited in practice. Type propagation and rule-based detection can only be conducted if generic type sources and patterns are available, whereas dynamic analysis requires comprehensive test cases to reach good code coverage.

In this paper we propose a static method for type recovery in binary programs. Our method builds on the assumption that types leave characteristic traits in the compiled code that can be automatically identified by machine learning techniques, once a classification algorithm has been trained. To this end, our method locates data objects such as local variables or function parameters in the compiled code and traces the data flow on the level of instructions. The resulting traces reflect how a data object is processed and thereby characterize the stored data type. We embed these traces in a vector space, such that similar traces are close to each other and can be assigned to labeled types using a classifier.

We demonstrate the efficacy of our method in an extensive empirical evaluation, where we recover elementary types, such as integers and floats, as well as pointers and to some extend even composite data types. As ground-truth we use 14 popular open-source software projects that make use of thousands of variables and parameters with various data types. Our experiments show that our method is able to recover 76 %–93 % of the used data types correctly and even rarely used data types can be detected. This makes TypeMiner a valuable tool in practice.

In summary, we make the following contributions:

- *Introduction of the data object graph.* We present a new representation for direct and indirect data dependencies between instructions of a binary program. This allows us to monitor the flow of data objects along execution traces.
- *Structural comparison of execution traces.* We present a method for extracting and classification of execution traces of data objects. Our approach enables the identification of traces that belong to data objects of the same data type, and thus forms the basis of our type estimation approach.
- *Empirical evaluation on real-world software.* We study our method's capability of recovering data types in real-world software by comparing traces of data objects. We specifically inspect the performance of the classification of pointer types, arithmetic types, and the signedness of integral types.

The rest of the paper is structured as follows: An overview of TypeMiner, our method for type estimation, is given in Section 2. Sections 3 to 6 introduce the individual steps of our method in detail, before Section 7 presents an extensive empirical evaluation. Related approaches are discussed in Section 8. Section 9 concludes the paper.

2 System Overview

This section gives an overview of TypeMiner, our method for the recovery of data types in binary code. In particular, we describe our approach for dependence analysis and explain how this representation allows us to effectively make use of machine learning techniques for type estimation.

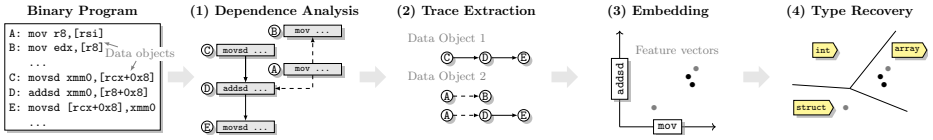


Fig. 1. Overview of our method for type estimation: For each data object TypeMiner analyzes dependencies (1) and extracts corresponding data object traces (2). These traces are subsequently embedded in a vector space (3) in order to predict the unknown data types (4).

Figure 1 depicts an overview of our method and the steps involved. First, our method receives the binary code of the program under analysis as input and performs static program analysis (Section 3). This step results in a representation specially geared to the problem of type recovery. Based on this representation, our method then identifies and extracts characteristic traits of data objects (Section 4). Subsequently, the resulting traces are first normalized and then embedded in a high-dimensional vector space to make the data usable for machine learning algorithms (Section 5). Finally, our method identifies data types of unknown data objects in this vector space (Section 6).

Code Analysis The individual steps of TypeMiner are based on disassembled program code. In order to obtain the assembly representation of binary code we make use of the disassembler IDA Pro [15]. Generally speaking, a disassembler translates the bytes of machine language instructions, that have previously been generated by the backend of a compiler, into assembly language—a low-level symbolic language that uses mnemonics to represent the machine instructions. Moreover, modern disassemblers perform additional analyses on the disassembled machine code. For example, IDA additionally structures the program into separate functions, as known from high-level languages such as the C programming language. The disassembler then generates the control flow graph (*CFG*) of each function, tries to identify local variables and function parameters, and provides detailed information about individual instructions, including their operands. Finally, IDA computes a call graph that represents the interaction of individual functions, providing us access to inter-procedural control flow. This preceding analysis of the binary program is crucial since errors propagate through succeeding steps and thus may lead to falsely recovered data types. Although such analyses represent non-trivial challenges of their own, we omit further details for the sake

of brevity. However, we use all high-level information provided by the disassembler as basis for our analyses. This includes the identification of data objects, such as local variables and function parameters. Finally note that, although in this work we focus on programs compiled for the x86-64 instruction set architecture [16], our approach is agnostic to of the CPU architecture and therefore can be used to assist the analyst in recovering types from code targeting diverse systems once a corresponding model has been trained for the target architecture.

Running Example We use the short program shown in Figure 2 as running example to illustrate the details of each step performed by our method. The C source code presented on the left side is first translated into machine code using a compiler and then re-interpreted as assembly code using a disassembler, which is displayed on the right. This example program adds up two arrays of 2D points (represented as pointers to `struct point`) by iterating over all pairs and overwriting each value of the first point with the computed sum. In the assembly program this loop is expressed as a do-while loop: The loop body is covered by the instructions from `0x10` to `0x2f`. The loop counter is stored in the register `rax`, decreased at `0x33`, and checked at `0x36`. Registers `rdi` and `rsi` are pointers to the arrays `pts1` and `pts2`. Registers `rcx` and `r8` are used to store the current array elements `pts1[i]` and `pts2[i]`, respectively. Finally, the addition and assignment of the `x` and `y` coordinates are carried out at `0x19` (displacement `0x00`) and `0x20` to `0x26` (displacement `0x08`).

```
struct point {                          0x00: test   edx,edx
  int x;                                0x02: jle    0x38
  double y;                             0x04: mov    eax,edx
};                                      0x10: mov    rcx,[rdi]
                                        0x13: mov    r8,[rsi]
void add(struct point *pts1[],          0x16: mov    edx,[r8]
         struct point *pts2[],          0x19: add    [rcx],edx
         int len)                       0x1b: movsd  xmm0,[rcx+0x8]
{                                       0x20: addsd  xmm0,[r8+0x8]
  int i;                                0x26: movsd  [rcx+0x8],xmm0
  for (i = 0; i < len; i++) {           0x2b: add    rdi,0x8
    pts1[i]->x += pts2[i]->x;           0x2f: add    rsi,0x8
    pts1[i]->y += pts2[i]->y;           0x33: dec    rax
  }                                     0x36: jne    0x10
}                                       0x38: ret
```

Fig. 2. Running example for show-casing our method's individual steps and inner workings throughout the paper: A simple addition of (arrays of) 2D-points (`struct point`) as C source code on the left and the corresponding assembler code on the right.

3 Dependence Analysis

Our method requires certain properties of the input binary code to be explicit and easily retrievable. We thus proceed by performing data dependency analysis on the instruction level of the disassembled program. In particular, we aim at building an expressive representation that enables us to detect all instruction sequences used for addressing and processing data objects. To this end, we extend the concept of data dependencies and compute two different, but related types of dependencies between individual instructions within the boundaries of functions and between function calls. First, we compute *regular data* dependencies by extracting *definition-use chains* of individual data objects [1]. These enable us to uncover sequences of instructions used to process or modify the original data object. To keep track of more complex objects, as for example data objects that are part of high-level data types (e.g., structure members and array elements) or that are references (pointers) to other data objects, we further analyze the program to build *indirect data dependencies*. Indirect data dependencies are imposed by instructions that dereference data objects to access another data object. These two types of dependencies between pairs of instructions implicitly define a multi graph that we call the *data object graph* (*DOG*).

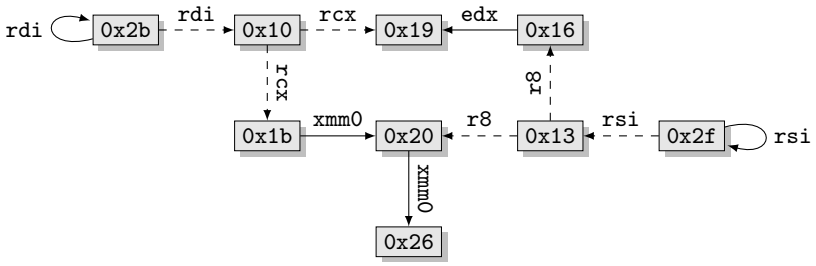


Fig. 3. Data object graph showing direct (solid edges, ⟶) and indirect (dashed edges, - - ➝) data dependencies of the instructions of the loop body.

The data object graph of the loop body from the running example is shown in Figure 3. While nodes are labeled with the addresses of the corresponding instructions, edges carry the name of the storage location that contains the data object's value to which the dependency exists. Dashed edges indicate indirect dependencies. In this example, the instruction at `0x10` (`mov rcx,[rdi]`) is indirectly data-dependent on the instruction at `0x2b` (`add rdi,0x8`), as the register `rdi`, written at address `0x2b`, is dereferenced at address `0x10`. For the very same reason, the instruction at `0x16` is also dependent on instruction `0x13`. The instruction at `0x19`, on the other hand, is directly data-dependent on the instruction at `0x16`, as it reads the actual value of register `edx` (the `x` coordinate

of a point from `pts2`). The instruction at `0x19` however is indirectly dependent on the instruction at `0x10`. It dereferences the value of register `rcx` loaded at `0x10` (the address of a point from `pts1`), but reads and writes the `x` coordinate of the point. Since the instruction at `0x2b` reads and writes the register `rdi` that was written in the previous iteration, it is dependent on itself. In the following, we explain how the data object graph enables us to characterize different patterns of object usage throughout the binary code of the program.

4 Extraction of Data Object Traces

Given the data object graph of a binary program, we are in the position to extract characteristic traits of data objects by traversing the graph. We refer to these linear instruction sequences as *data object traces* or *traces* for short, which represent specific usage patterns of data objects in the binary code.

In order to infer the type of a data object (i.e., a local variable or a function parameter), the analyst starts by identifying one or more access locations thereof, that is, an instruction using the data object. Then, TypeMiner automatically tracks the chosen data object by traversing the data object graph DOG starting at the selected instructions. More formally, to extract traces of a data object d, we first identify instructions accessing the data object and denote this set by I_d. We further define the set of all instructions reachable from I_d

$$V_d = \{v \in V(DOG) \colon v \text{ reachable from } i \in I_d\}$$

and the induced subgraph of V_d

$$DOG[V_d] = (V_d, \{(u, v) \in E(DOG) \colon u \in V_d\})\,.$$

TypeMiner proceeds by extracting all traces T_d of d by traversing $DOG[V_d]$. Starting at each instruction in I_d, we terminate the extraction of a trace if one of the following conditions is met:

T1. The trace contains more than a predefined number of instructions, not counting plain `mov` instructions reached by direct data-dependence. This conditions guarantees that the traversal eventually terminates. We do not count plain `mov` instructions since these do not carry any characteristic type information, e.g., `mov rdx, rcx`. However, we do count `mov` instructions that are reached via indirect data dependencies, e.g., `mov rdx,[rcx]`. We call all other instructions *type-relevant* and denote the number of type-relevant instructions in a trace $t \in T_d$ by $|t|$.

T2. The trace contains at most one indirection of the tracked data object. A single level of indirection is sufficient to differentiate between pointer types and non-pointer types, but also ensures that a data object is not tracked beyond multiple indirections at the same time. For example, after the indirection of a pointer to a structure, the structure itself is being tracked—or more precisely, its members. However, we are not interested in the types of each member. Hence, the extraction stops before a data object is dereferenced for the second time.

T3. The tracked data object is merged into another data object, meaning, the data object's storage location is used as source operand, while the destination operand is read *and* written. For example, consider the instruction `add rcx,rdx`, where `rdx` is the storage location of the tracked data object. After execution of this instruction, we would proceed to track the data object stored in `rcx`. However, this data object might have a different type. For instance, the data object stored in `rcx` could be a pointer type, while the data object stored in `rdx` is of type `int`.

Applied to the running example of Figure 2, the parameter `pts1`, for instance, has type "array of pointer to struct point" and is stored in register `rdi`. The data object graph shown in Figure 3 unveils the following traces: `0x2b` - - ▸ `0x10`, `0x2b` ⟶ `0x2b` - - ▸ `0x10`, and `0x2b` ⟶ ··· ⟶ `0x2b` - - ▸ `0x10`. The data object is dereferenced at `0x10`, hence the data object's value must be an address. Since the address is increased by 8 bytes in each iteration the data object must be an sequence of multiple elements, being data objects themselves.

When looking at an element from `p1`, we can observe the following two traces: `0x10` - - ▸ `0x19`, `0x10` - - ▸ `0x1b`. The array element is dereferenced at two different locations, with different offsets (`0x0` and `0x08`). Hence, each element is probably a pointer to an aggregate data type, e.g., a pointer to a structure.

Going one step further, the `y` member of the structure `struct point`, stored at `[rcx+0x8]`, yields the following trace: `0x1b` ⟶ `0x20` ⟶ `0x26`. The instructions `movsd` (move scalar double-precision floating point value) and `addsd` (add scalar double-precision floating-point values) give some indication of the data type, `double` in our running example.

5 Embedding of Data Object Traces

Once traces from all available data objects have been extracted, we proceed by building a vectorial representation suited for type recovery that allows us to learn a machine learning model that abstracts the peculiarities of data types and is used to predict the unknown type of data objects.

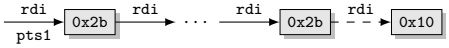
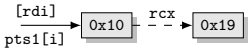
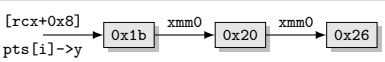
To this end, each instruction contained in a data object trace is normalized prior to embedding the trace, to emphasize the data type specific characteristics. In other words, each trace is translated into a sequence of normalized instructions. The normalized sequences are subsequently mapped to a feature space.

5.1 Normalization

We seek a normalization that strips the information that is not specifically attributed to a certain data type and, hence, irrelevant for the task of type recovery, but keeps the characteristic information untouched.

To this end, we begin by considering the mnemonic of the instructions in each trace, as particular instructions often operate on very specific data types. For example, instructions performing bit manipulations usually operate on integers.

Table 1. Normalization of three data object traces.

Instruction	Normalization
	
0x2b `add rdi,0x8`	⇝ `add|data_object(0)_width8|immediate_width8`
0x10 `mov rcx,[rdi]`	⇝ `mov|storage_location_width8|data_object(1)_width8`
	
0x10 `mov rcx,[rdi]`	⇝ `mov|storage_location_width8|data_object(0)_width8`
0x19 `add [rcx],edx`	⇝ `add|data_object(1)_width4|storage_location_width4`
	
0x1b `movsd xmm0,[rcx+0x8]`	⇝ `movsd|xmm_register_width8|data_object(0)_width8`
0x20 `addsd xmm0,[r8+0x8]`	⇝ `addsd|data_object(0)_width8|storage_location_width8`
0x26 `movsd [rcx+0x8],xmm0`	⇝ `movsd|storage_location_width8|data_object(0)_width8`

Hence the usage of instruction like `and`, `or`, `xor`, `shl`, etc., indicate that the data object presumably is of some integral type. The assembly instruction `lea` ("load effective address"), in particular, is often used in conjunction with arrays or structures that are commonly used in high-level languages like C to load the address of an array element or structure member into a register. As an example, `struct point vec[10]` declares a variable `vec` as an array of size 10 of type `struct point`. Assuming that register `rdi` points to the beginning of the array and `rax` has the element index 5, the address of `vec[5].y` is loaded into `rdx` using `lea rdx,[rdi+rax*0x8+0x8]`. Unfortunately, especially in optimized code, the `lea` instruction is often used to perform more powerful additive operations such as adding three operands (two registers and an immediate) and subsequently storing the result in an arbitrary register. This behavior can only be accomplished with multiple `add` instructions otherwise.

In conjunction with the mnemonic, the instruction's operands give additional hints on the data type. The width data of an operand helps to differentiate between different arithmetic types, e.g. `int` and `long int`, which are, depending on the platform, 32-bit and 64-bit wide. A register used in an indirection usually contains a data object of a pointer type. On the contrary, whether a data object is stored in a specific register or in memory is not revealing any useful type specific information.

Hence, we represent each operand by its type (`xmm_register` for 128-bit SSE registers, `storage_location` for all other registers as well as locations in memory and `immediate` for constant values encoded into the instruction's opcode) and its width. Moreover, we mark the operand that contains the currently tracked data object and indicate the level of indirection, which depends on the previous instructions.

Formally, the normalized sequence of a trace t is represented by

$$\varrho(t) = (a_1, a_2, \ldots, a_{\|t\|}) \in \mathcal{A}^{\|t\|}$$

where $\mathcal{A}$ is the set of all possible normalized instructions from the X86-64 instruction set and $\|\cdot\|$ denotes the length, that is, the number of all instructions in the trace, i.e., $\|\cdot\|$ and $|\cdot|$ are different trace lengths.

In summary, a normalized instruction consists of the instruction's mnemonic and the normalized operands. Each normalized operand consists of two parts: one part describes the type of the operand and the other part the width of the operand's value. In addition, operands carrying the data object or an indirection thereof are marked as such. Table 1 shows the normalized instructions of some of the extracted traces discussed in the previous section.

5.2 Embedding

To capture the characteristics of each data object trace, we make use of so-called *n-gram models*. The technique was originally conceived for natural language processing [21, 22] and information retrieval [35], to embed traces in a vector space.

Given the normalized instruction sequence $\varrho(t) = (a_1, a_2, \ldots, a_{\|t\|})$ of a trace t, we extract unigram (1-gram) as well as bigram (2-gram) features:

$$\{a_i \colon 1 \leq i \leq \|t\|\} \text{ and } \{(a_i, a_{i+1}) \colon 1 \leq i < \|t\|\}.$$

More formally, based on this feature set, we construct a vectorial representation of the trace, with $\mathcal{A}$ being the set of all normalized instructions. The set of all observable features in our model is given by

$$F = \underbrace{\mathcal{A}}_{\text{1-grams}} + \underbrace{\mathcal{A}^2}_{\text{2-grams}} .$$

Making use of the feature set F, we define an $|F|$-dimensional vector space that takes values 0 or 1 in each dimension. Each trace t is then mapped to this space by building a feature vector $\varphi(t)$, such that for each n-gram feature present in t the corresponding dimension is set to 1. All other dimensions are set to 0. Formally, this map can be defined for all traces T as

$$\varphi : T \to \{0,1\}^{|F|}, \quad \varphi(t) \mapsto (I_f(t))_{f \in F}$$

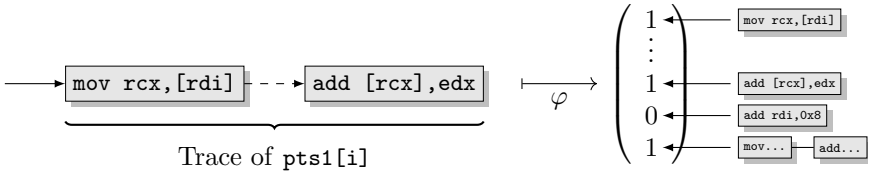


Fig. 4. Exemplary embedding of a data object trace. Normalization of the instructions is omitted for the sake of readableness.

where the auxiliary function I indicates whether the feature f is present in the trace t, that is,

$$I_f(t) = \begin{cases} 1 & \text{if the normalized trace } \varrho(t) \text{ contains feature } f \\ 0 & \text{otherwise.} \end{cases}$$

The resulting binary vector space $\{0,1\}^{|F|}$ allows us to represent each data object trace as a vector of the traits contained in the set of instructions that define the trace. Figure 4 exemplarily shows the embedding of a trace. The trace yields three different features (two unigrams and a bigram). Each feature is mapped to the corresponding entry in the feature vector, setting three of its entries to 1.

In the following, we describe how we use this representation to build a multi-class classification scheme that, based on these features, allow us to predict the data type of previously unseen data objects.

6 Classification

We use a multi-stage classification scheme to recover the type of a data object in multiple steps. Each stage uses a specialized classifier trained to recover a specific part of the data type. Figure 5 illustrates this process. First, the analyst selects one or more access locations of the same data object. Second, all extracted traces, that have their sources from the selected locations, are joined using the logical-or operator. The resulting binary vector represents the data object traces altogether and is then used as input for each classifier. Which classifier is used depends on the prediction of the previous classifier. The final result is reported back to the analyst and can be used to annotate the selected access locations.

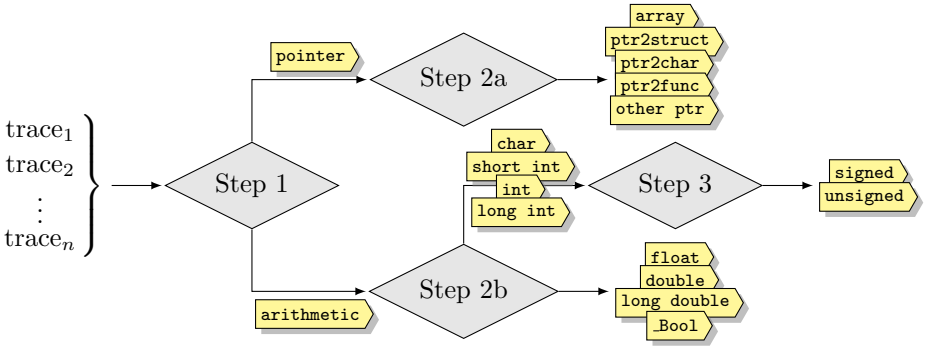


Fig. 5. Type recovery process of a given data object.

At the first stage, TypeMiner employs a binary classifier to identify whether the data object has an *arithmetic* type (*integer* type, including `_Bool`, or *real floating* type, as listed in the international standard of the C programming

language [17]) or *pointer* type. Data objects tagged as pointers are processed at Step 2a and labeled *array* types, *pointer-to-structure* types, *pointer-to-char*, *pointer-to-function* types, or other pointer types. At step 2b data objects tagged as arithmetic types are labeled based on the classifiers prediction. Whether an integer type is *signed* or *unsigned* is finally determined at Step 3. We use an Random Forest classifier at Steps 1 and 3 and a linear Support Vector Machine at Steps 2a and 2b.

While arithmetic types are fully determined after a single pass through TypeMiner's classification scheme, the referenced type of a pointer, the element type of an array, or the data types of member objects of structures remain undefined. To fully determine those types the traces corresponding to those member objects must be equally fed into TypeMiner's classification process. Note that TypeMiner does not attempt to recover the complete layout of structures nor the size of arrays.

While arrays are not pointers, array indexing and pointer arithmetic are equivalent in C. Moreover, an array-of-type-`T` passed to a function immediately decays to a pointer-to-type-`T` [39]. Based on that observation, TypeMiner considers array types as pointer types and process them together with real pointer types in Step 2a. On the contrary, structure types can be passed by value, rendering the detection of structure types practically infeasible, since a function local declaration of a structure type is indistinguishable from multiple independent declarations. For that reason, TypeMiner falls back to the recovery of member objects of structures instead.

7 Evaluation

We are interested in examining the capability of our method in recovering data types under realistic conditions. We thus proceed to evaluate our method on real binary code obtained by compiling 14 popular open-source projects. Subsequently, we first describe our dataset in Section 7.1, before we present the experimental setup in Section 7.2. To evaluate TypeMiner and to explore its benefits in deployment we conduct the following experiments:

1. We perform an empirical evaluation of TypeMiner to assess its effectiveness for data type recovery in x86-64 binary programs. To this end, we test the performance our method on 14 binary programs regarding the four type recovery problems of TypeMiner's classification scheme (Section 7.3).
2. We compare TypeMiner with a set of handcrafted rules used to partition scalar types into pointer types and arithmetic types as well as to differentiate different width integer types (Section 7.4).

We omit a direct comparison with type recovery engines implemented in decompilers, as for instance used by IDA Pro, as these need to be conservative in terms of derived types—their main goal is to produce valid source code. The decompiled code, however, does not necessarily have to contain the original types to achieve this.

7.1 Dataset

We create a comprehensive dataset from 14 open-source software projects. To this end, we build each project with the optimized "release" configuration. Subsequently, we disassemble each binary program and leverage the debugging information to obtain ground truth type labels of identified data objects. We proceed by computing the data object graphs of each binary program and use the information to extract data object traces for all data objects. In total, the dataset consists of 817 K instructions and 23,482 data objects. Table 2 summarizes the information extracted from each binary program.

The construction of all data object graphs takes 42 min for intra-procedural analysis and additional 102 min for inter-procedural analysis. The extraction time of data object traces is 3 s per data object on average. All execution times have been measured on a system using a single core at 2.30 GHz and 32 GB of RAM.

For each type, Table 3 additionally lists the number of data objects d with respect to the number of type-relevant instructions found in their traces $t \in T_d$, i.e., $\max_{t \in T_d} |t| \geq n$, $n \in \{1, 2, \ldots, 5\}$. Data objects of certain types leave behind shorter traces than others. For example, only 28 % of the identified data objects of type `_Bool` have more than one type-relevant instruction. Data objects of type `double` have significant longer traces: over 50 % have traces with more than two type-relevant instructions. It can be assumed that data objects of type `double` undergo more complex computations, hence, having longer traces on average.

7.2 Experimental Setup

To evaluate TypeMiner, we train four independent classifiers for each type recovery problem of the classification process using the traces recorded from all but one binary program as training data. Subsequently, we evaluate the classifier by testing its performance on data objects extracted from the remaining binary program. This procedure gives us a natural separation of training data and test data, and ensures that each classifier is tested on previously unseen data. Note that different data objects can be mapped to identical features. This may even happen for data objects of different types. A special case is the inlining of standard library functions, where training and test data may contain traces that

Table 2. Overview of the binary programs in our dataset.

Program	# Data Obj.	# Instr.	Program	# Data Obj.	# Instr.
bash	6,496	157 K	gzip	424	10 K
bc	422	10 K	indent	174	10 K
bison	2,470	58 K	less	961	20 K
cflow	768	18 K	libpng	1,968	33 K
gawk	3,472	98 K	nano	1,526	34 K
grep	1,227	24 K	sed	709	15 K
gtypist	145	5 K	wget	2,720	58 K

Table 3. Number of data objects for different maximum trace lengths.

Data type	Number of data objects				
	$\max_t \lvert t\rvert \geq 1$	$\max_t \lvert t\rvert \geq 2$	$\max_t \lvert t\rvert \geq 3$	$\max_t \lvert t\rvert \geq 4$	$\max_t \lvert t\rvert \geq 5$
`_Bool`	202 (100 %)	57 (28 %)	32 (16 %)	11 (5 %)	8 (4 %)
`char`	97 (100 %)	73 (75 %)	22 (23 %)	13 (12 %)	9 (9 %)
`short int`	15 (100 %)	9 (60 %)	3 (20 %)	1 (7 %)	1 (7 %)
`int`	6,013 (100 %)	3,956 (66 %)	2,752 (46 %)	1,829 (30 %)	1,429 (24 %)
`long int`	2,594 (100 %)	1,654 (64 %)	1,157 (45 %)	638 (25 %)	481 (19 %)
`double`	50 (100 %)	48 (96 %)	27 (54 %)	14 (28 %)	13 (26 %)
`array`	33 (100 %)	21 (64 %)	13 (39 %)	7 (21 %)	5 (15 %)
`ptr2struct`	4,017 (100 %)	1,935 (48 %)	1,309 (33 %)	911 (23 %)	608 (15 %)
`ptr2char`	4,381 (100 %)	1,990 (45 %)	1,564 (36 %)	991 (23 %)	808 (18 %)
`ptr2func`	93 (100 %)	8 (9 %)	6 (6 %)	4 (4 %)	0 (0 %)
other `ptr`	1,281 (100 %)	584 (46 %)	408 (32 %)	285 (22 %)	177 (14 %)
Total	18,776 (100 %)	10,335 (63 %)	7,293 (45 %)	4,707 (29 %)	3,539 (22 %)

share the same instructions. In each training phase, we use the data object traces that contain at least two type-relevant instructions to perform a $(n-1)$-fold cross validation to find the best model based on the training data. Again, we make sure that two different folds cannot contain data from the same binary program. In total, we conduct $14 \cdot 4$ experiments to trial TypeMiner in an extensive empirical evaluation where each binary is used once for testing.

Performance Metrics We use two performance metrics commonly used to evaluate machine learning classifiers: precision and recall. The precision score of a class y describes the ability of a classifier not to label objects of different classes as y. The recall score of a class y describes the ability of a classifier to label objects of this class as y. Formally, the precision and recall score of class y are defined as follows:

$$\text{precision}_y = \frac{TP_y}{TP_y + FP_y} \quad \text{and} \quad \text{recall}_y = \frac{TP_y}{TP_y + FN_y}.$$

The true-positives, TP_y, denote the number of samples from class y, meaning, data objects that have type y, correctly labeled as y. The false-positives, FP_y, are the number of samples from classes incorrectly labeled as y. Finally, the false-negatives, FN_y, denote the number of samples from class y that were not labeled as y.

Whenever it is desired to compute a single precision or recall score over all classes Y we use micro averaging for precision and macro averaging for recall. Micro averaging computes the number of true-positives, false-positives, and false-negative globally and macro averaging locally,

$$\text{precision}_{\text{micro}} = \frac{\sum_{y \in Y} TP_y}{\sum_{y \in Y} (TP_y + FP_y)} \quad \text{and} \quad \text{recall}_{\text{macro}} = \frac{\sum_{y \in Y} \text{precision}_y}{|Y|}.$$

The difference is that micro averaging does take label imbalance into account whereas macro averaging treats all classes as equal. The $\text{precision}_{\text{micro}}$ score can be interpreted as the percentage of correctly recovered data types. It is equal to the accuracy. The $\text{recall}_{\text{macro}}$ can be seen as the average detection rate over all data types.

7.3 Empirical Evaluation

In our first experiment, we measure the performance of TypeMiner depending on the length of the extracted traces. Therefore, we test the trained classifiers on data objects with traces of various length, i.e., by successively precluding data objects with short traces from the classification. We use this experiment to point out performance differences that result from to different trace lengths, which can be limited by the obstacles of dependence analysis.

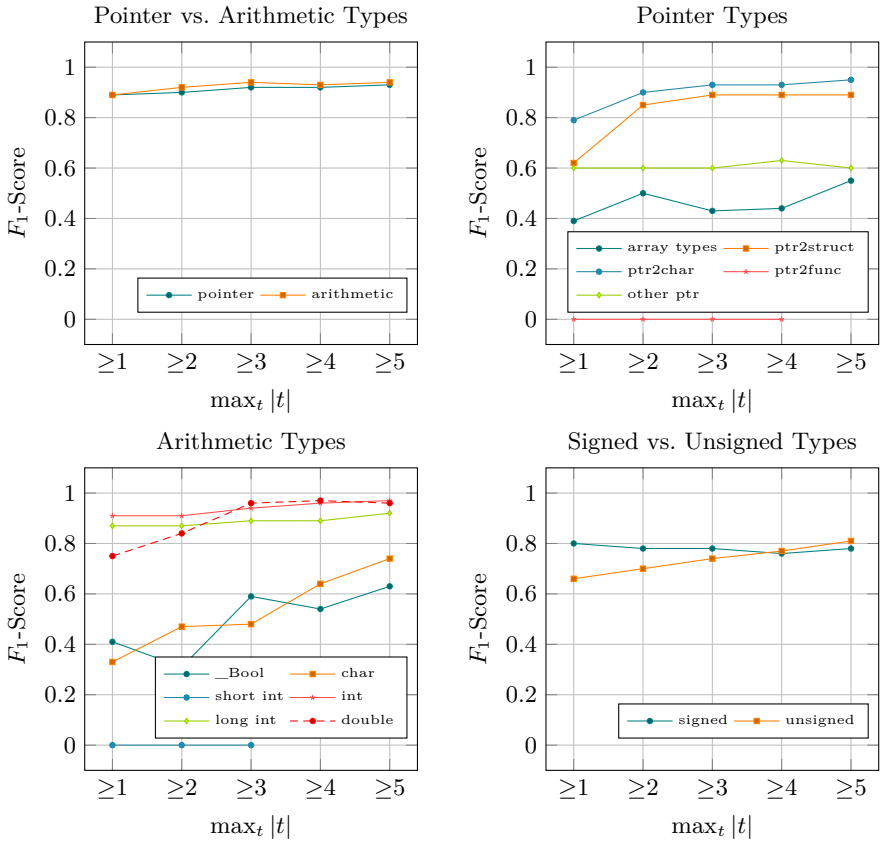


Fig. 6. Performance of TypeMiner as F_1-Score for each classification problem corresponding to the four steps of TypeMiner's classification scheme.

The results are presented in Figure 6, showing the F_1-score, i.e., the harmonic average of the precision and recall scores, for each data type of the four classification problems of TypeMiner's scheme with regard to the maximum trace length of the considered traces. The four plots demonstrate that TypeMiner tends to be more accurate in classifying data objects with longer traces. This comes at no surprise: since longer traces potentially contain more characteristic traits that point to the correct data type of an object. For frequently used types, the performance of TypeMiner is remarkable and reaches F_1 scores above 90 %.

Although TypeMiner achieves an overall good performance, data objects of types "pointer-to-function" and `short int` cause difficulties. Table 3 shows that the overall dataset contains only few data objects with traces of sufficient length of the respective type. Based on these few samples, TypeMiner fails to learn expressive usage patterns of the mentioned types.

Fortunately, the detection of function pointers in binary code obtained from C source code is rather simple. Since data objects of type pointer to function hold the addresses of functions, they can be easily detected when being used in a `call` instruction. In our dataset, 90 % of all data objects of type "pointer-to-function" yield a trace with a `call` instruction. The detection of data objects of type `short int` is not that simple. Even an instruction loading 2 bytes from memory is insufficient for the identification of `short int` data objects, since the instruction could be used to load the lower bytes of a larger integer. TypeMiner labels all data objects of type `short int` as `int` or `long int`.

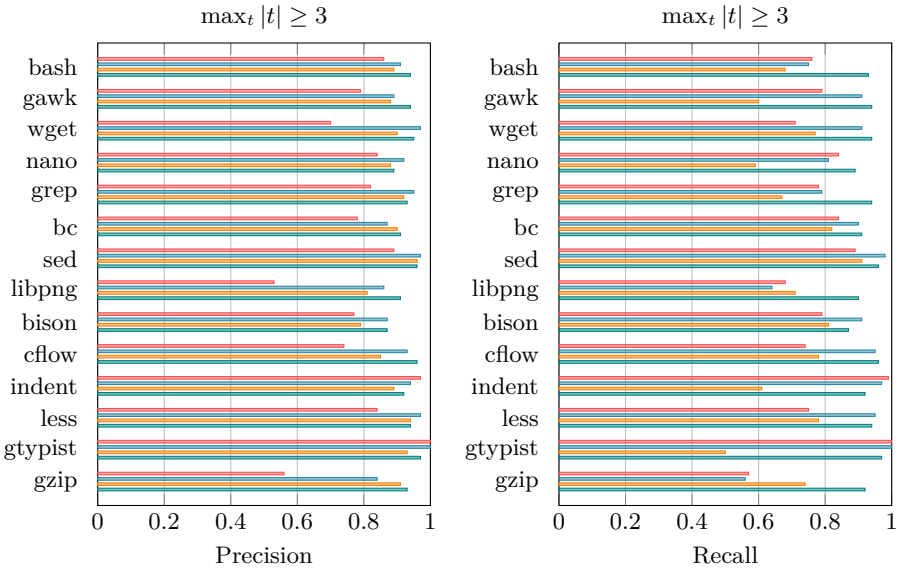


Fig. 7. Precision and recall of each binary program. The results for different classification stages are separated by different bars (bottom to top): pointer vs. arithmetic types (), pointer types (), arithmetic types (), and signed vs. unsigned types ().

In our second experiment, we measure the performance of TypeMiner on each binary program separately. In this experiment we consider all data objects with traces that contain at least three type-relevant instructions and test each trained classifier on the four classification problems.

Figure 7 shows the precision (micro avg.) and recall (macro avg.) scores achieved by TypeMiner for all 14 binary program as a bar plot. Each classification problem is displayed as different colored bar. We cannot identify any extreme outlier among the software projects and conclude that traces of binary programs can be used to train a model that is capable to make good predictions on an entirely different code base.

In total, 93 % of all data objects are correctly classified as pointer or arithmetic types, 88 % and 92 % of all pointer types and arithmetic types can be recovered correctly, and the signedness of 76 % integer types is inferred properly. The average detection rate for pointer and arithmetic types is 92 %, and 76 % for signed and unsigned integers. In case of pointer types and arithmetic types the average detection rate is 58 % and 70 %, respectively. In these particular cases, the detection rate is lowered by ∼14 % by the limitation of TypeMiner to detect the data types "pointer-to-function" and `short int` data types.

7.4 Comparison with Rule-based Type Recovery

To examine what TypeMiner learns from the training data, we conduct two experiments in which we compare our learning-based approach with a set of handcrafted rules. We focus on the differentiation of pointer types and arithmetic types, and different sized integer types. In both cases simple rules can be manually derived.

To distinct arithmetic types from pointer types we check whether the tracked data object is dereferenced in one of the extracted traces. In the case the data object is dereferenced we label it as "pointer" and "arithmetic" otherwise. The rules for the distinction of integral types is based on the used widths of the operands that occur in the data object traces: Based on the most occurring width used in source operands we label it as "1-byte-integer", "2-byte-integer", "4-byte integer", and "8-byte-integer" accordingly.

The precision and recall scores are presented in Table 4. The results show that TypeMiner outperforms the rule-based approach in almost every case. Except for 2-byte integral types, which correspond to `short (unsigned) int` on our system. Remarkably is that TypeMiner is more accurate in identifying pointer and arithmetic types. Thus, TypeMiner is able to learn what a pointer is even if the corresponding data object is never dereferenced, meaning that pointer arithmetic differs from integer arithmetic in some way. As a naïve baseline, Table 4 shows additionally the expected performance of an algorithm that learns the distribution of data types from the training data and makes predictions equally.

Although TypeMiner's approach is inspired by an analyst processing certain rules to detect the most likely data type of some data object, our method does not only process its rules automatically, but also learn those rules autonomously.

Table 4. Precision and recall of rule-based type recovery and TypeMiner.

Data Type	Prob-based		Rule-based		TypeMiner		Support
	Prec.	Recall	Prec.	Recall	Prec.	Recall	
pointer types	0.45	0.45	0.88	0.90	0.93	0.91	3296
arithmetic types	0.55	0.55	0.91	0.90	0.93	0.95	3990
micro avg.	0.50	0.50	0.90	0.90	0.93	0.93	7286
macro avg.	0.50	0.50	0.90	0.90	0.93	0.93	7286
1-byte integer	0.01	0.01	0.06	0.27	0.48	0.73	22
2-byte integer	0.00	0.00	0.33	0.33	–	0.00	3
4-byte integer	0.70	0.70	0.95	0.90	0.96	0.93	2752
8-byte integer	0.29	0.29	0.84	0.88	0.86	0.92	1157
micro avg.	0.57	0.57	0.89	0.89	0.93	0.93	3934
macro avg.	0.25	0.25	0.54	0.60	0.77	0.64	3934

8 Related Work

Recovering types in binary code involves solving a series of challenging problems, some of which have been addressed throughout the literature from different perspectives and with different goals. In particular, the survey work by Caballero and Lin [5] attempts to systematize the area of type inference and discusses previous work according to the types inferred, its intended application and the specifics of its implementation and evaluation. For instance, all approaches can be classified according to the type of analysis performed on the input binary code, being either static or dynamic. Static approaches are limited by the disassembling process which can be troublesome if the binary code is obfuscated. However, as in our work, static approaches [9, 28, 31, 44] are able to reach better code coverage than methods based on dynamic analysis [27, 29, 34, 42]. While dynamic methods evaluate one execution at a time, static approaches do not require precomputed inputs to maximize the exploration of the code. Building on this trade-off, some researchers have proposed hybrid approaches which extend static information with traces obtained during the execution of the binary [4, 23, 26].

While our work focuses on the recovery of types from C source code, a related research field addresses the problem of identifying types in binary programs of object oriented code. Especially, the inference of (runtime) classes of C++ objects as well as identifying class hierarchies is a vivid research topic [11, 20, 29].

Other papers aiming at the recovery of C-style data types are shortly described in the remaining part of this section. Lin et al. [27] presented REWARDS, a dynamic approach to data type reconstruction. REWARDS takes advantage of known system and library functions to obtain type information. This information is propagated along execution paths to other locations in the binary program. The type of an data object is then resolved if it reaches a tagged location. Moreover, REWARDS assigns specific type names to type sources depending on the semantic interpretation of data obtained from those sources. Lee et al. [26] presented a type

inference system based on type reconstruction theory. In contrast to REWARDS, TIE can also be used in a static analysis setting. Slowinska et al. [37] presented Howard, a technique for reverse engineering data structures in binary programs compiled from C code. Howard uses dynamic analysis to recover data structures by observing memory accesses patterns at runtime. Haller et al. [13] presented the tool MemPick that detects and classifies high-level data structures stored on the heap in C/C++ binaries. MemPick dynamically observes how the shape of heap objects evolve over the time to infer the used data structure.

Our work is different in the sense that we do not aim to reassemble the structure to infer the syntactic definition of data types, but to find data objects that use a type already observed at another location. This approach makes it possible to detect software-specific types. A similar technique as used by TypeMiner is presented by Katz et al. [24]. They attempt to determine the likely targets of virtual function calls by building static traces of higher-level events on an object. Similar to our work, they rely on examples where the type of an object is already known to infer types of unknown instances. However, they aim at identifying the runtime type of project-specific class objects in C++ binaries, while TypeMiner focuses on inferring C types in stripped binary code using a prediction model that can be applied cross-project.

9 Conclusion

A core challenge in reverse code engineering is the missing information of data types. It is a tedious task for an analyst to track an data object across the binary program to identify characteristics that point towards a specific data type. In this paper, we presented TypeMiner, a method for recovering high-level data types in binary code. In essence, our method is based on machine learning techniques. Necessary to that end is the extraction of static execution traces of data objects and the classification thereof. As a static approach, TypeMiner does not have to cope with code coverage as dynamic approaches. In contrast to other static approaches, our method addresses the recovery of data types without the need of additional expert knowledge, that is, without leveraging known sources like library functions as starting points, as such information is not always available. Our method can reach high accuracy given an analyst provides sufficient long execution traces and, thereby, can help an analyst in an interactive manner.

Acknowledgments

The authors gratefully acknowledge funding from the German Federal Ministry of Education and Research (BMBF) under the project VAMOS (FKZ 16KIS0534) and FIDI (FKZ 16KIS0786K).

Bibliography

[1] Aho, A.V., Sethi, R., Ullman, J.D.: Compilers Principles, Techniques, and Tools. Addison-Wesley, second edn. (2006)

[2] Böhme, M., Pham, V.T., Nguyen, M.D., Roychoudhury, A.: Directed greybox fuzzing. In: Proc. of the ACM Conference on Computer and Communications Security (CCS). pp. 2329–2344 (2017)

[3] Brumley, D., Chiueh, T., Johnson, R., Lin, H., Song, D.X.: RICH: Automatically protecting against integer-based vulnerabilities. In: Proc. of the Network and Distributed System Security Symposium (NDSS) (2007)

[4] Caballero, J., Johnson, N.M., McCamant, S., Song, D.: Binary code extraction and interface identification for security applications. In: Proc. of the Network and Distributed System Security Symposium (NDSS) (2010)

[5] Caballero, J., Lin, Z.: Type inference on executables. ACM Computing Surveys (CSUR) 48 (2016)

[6] Checkoway, S., Maskiewicz, J., Garman, C., Fried, J., Cohney, S., Green, M., Heninger, N., Weinmann, R.P., Rescorla, E., Shacham, H.: A systematic analysis of the Juniper Dual EC incident. In: Proc. of the ACM Conference on Computer and Communications Security (CCS). pp. 468–479 (2016)

[7] Chipounov, V., Kuznetsov, V., Candea, G.: S2E: A platform for in-vivo multi-path analysis of software systems. In: Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). pp. 265–278 (2011)

[8] Costin, A., Zaddach, J., Francillon, A., Balzarotti, D.: A large-scale analysis of the security of embedded firmwares. In: Proc. of the USENIX Security Symposium. pp. 95–110 (2014)

[9] Dewey, D., Giffin, J.: Static detection of C++ vtable escape vulnerabilities in binary code. In: Proc. of the Network and Distributed System Security Symposium (NDSS) (2012)

[10] Dietz, W., Li, P., Regehr, J., Adve, V.: Understanding integer overflow in C/C++. In: Proc. of the International Conference on Software Engineering(ICSE). pp. 760–770 (2012)

[11] Fokin, A., Troshina, K., Chernov, A.: Reconstruction of class hierarchies for decompilation of C++ programs. In: European Conference on Software Maintenance and Reengineering (CSMR) (2010)

[12] Haller, I., Jeon, Y., Peng, H., Payer, M., Giuffrida, C., Bos, H., Kouwe, E.V.D.: TypeSan: Practical type confusion detection. In: Proc. of the ACM Conference on Computer and Communications Security (CCS). pp. 517–528 (2016)

[13] Haller, I., Slowinska, A., Bos, H.: Mempick: High-level data structure detection in C/C++ binaries. In: Proc. of the Working Conference on Reverse Engineering (WCRE) (2013)

[14] Hex-Rays SA: Hex-Rays Decompiler. `https://www.hex-rays.com/products/decompiler` (2017), visited February, 2019

[15] Hex-Rays SA: Hex-Rays IDA Disassembler. `https://www.hex-rays.com/products/ida` (2017), visited February, 2019
[16] Intel Corporation: Intel® 64 and IA-32 Architectures Software Developer's Manual. Intel Corporation (2016)
[17] ISO: Programming languages – C. International Organization for Standardization (April 2011), Committee Draft (N1570)
[18] Jamrozik, K., Fraser, G., Tillmann, N., de Halleux, J.: Augmented dynamic symbolic execution. In: Proc. of the International Conference on Automated Software Engineering (ASE). pp. 254–257 (2012)
[19] Jeon, Y., Biswas, P., Carr, S.A., Lee, B., Payer, M.: HexType: Efficient detection of type confusion errors for c++. In: Proc. of the ACM Conference on Computer and Communications Security (CCS). pp. 2373–2387 (2017)
[20] Jin, W., Cohen, C., Gennari, J., Hines, C., Chaki, S., Gurfinkel, A., Havrilla, J., Narasimhan, P.: Recovering C++ objects from binaries using interprocedural data-flow analysis. In: Proc of the ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW) (2014)
[21] Joachims, T.: Text categorization with support vector machines: Learning with many relevant features. Tech. Rep. 23, LS VIII, University of Dortmund (1997)
[22] Joachims, T.: Learning to classify text using support vector machines. Kluwer (2002)
[23] Jung, C., Clark, N.: DDT: Design and evaluation of a dynamic program analysis for optimizing data structure usage. In: Proc. of the IEEE/ACM International Symposium on Microarchitecture (MICRO) (2009)
[24] Katz, O., El-Yaniv, R., Yahav, E.: Estimating types in binaries using predictive modeling. In: Proc. of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 313–326 (2016)
[25] Lee, B., Song, C., Kim, T., Lee, W.: Type casting verification: Stopping an emerging attack vector. In: Proc. of the USENIX Security Symposium. pp. 81–96 (2015)
[26] Lee, J., Avgerinos, T., Brumley, D.: TIE: Principled reverse engineering of types in binary programs. In: Proc. of the Network and Distributed System Security Symposium (NDSS) (2011)
[27] Lin, Z., Zhang, X., Xu, D.: Automatic reverse engineering of data structures from binary execution. In: Proc. of the Network and Distributed System Security Symposium (NDSS) (2010)
[28] Noonan, M., Loginov, A., Cok, D.: Polymorphic type inference for machine code. In: Proc. of the ACM SIGPLAN International Conference on Programming Languages Design and Implementation (PLDI) (2016)
[29] Pawlowski, A., Contag, M., van der Veen, V., Ouwehand, C., Holz, T., Bos, H., Athanasopoulos, E., Giuffrida, C.: MARX: Uncovering class hierarchies in C++ programs. In: Proc. of the Network and Distributed System Security Symposium (NDSS) (2017)
[30] Petsios, T., Tang, A., Stolfo, S., Keromytis, A.D., Jana, S.: Nezha: Efficient domain-independent differential testing. In: Proc. of the IEEE Symposium on Security and Privacy. pp. 615–632 (2017)

[31] Prakashm, A., Hu, X., Yin, H.: vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In: Proc. of the Network and Distributed System Security Symposium (NDSS) (2015)
[32] Ramos, D.A., Engler, D.: Under-constrained symbolic execution: Correctness checking for real code. In: Proc. of the USENIX Security Symposium. pp. 49–64 (2015)
[33] Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H.: VUzzer: Application-aware evolutionary fuzzing. In: Proc. of the Network and Distributed System Security Symposium (NDSS) (2017)
[34] Rupprecht, T., Chen, X., White, D.H., Boockmann, J.H., LÃijttgen, G., Bos, H.: DSIbin: Identifying dynamic data structures in C/C++ binaries. In: Proc. of the International Conference on Automated Software Engineering (ASE) (2017)
[35] Salton, G., Wong, A., Yang, C.S.: A vector space model for automatic indexing. Communications of the ACM 18(11), 613–620 (1975)
[36] Schumilo, S., Aschermann, C., Gawlik, R., Schinzel, S., Holz, T.: kAFL: Hardware-assisted feedback fuzzing for os kernels. In: Proc. of the USENIX Security Symposium. pp. 167–182 (2017)
[37] Slowinska, A., Stancescu, T., Bos, H.: Howard: A dynamic excavator for reverse engineering data structures. In: Proc. of the Network and Distributed System Security Symposium (NDSS) (2011)
[38] Stephens, N.D., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: Proc. of the Network and Distributed System Security Symposium (NDSS) (2016)
[39] Summit, S.: C Programming FAQs: Frequently Asked Questions. Addison-Wesley (1996)
[40] Wang, T., Wei, T., Lin, Z., Zou, W.: IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In: Proc. of the Network and Distributed System Security Symposium (NDSS) (2009)
[41] Wang, X., Chen, H., Jia, Z., Zeldovich, N., Kaashoek, M.F.: Improving integer security for systems with KINT. In: Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI). pp. 163–177 (2012)
[42] White, D.H., Rupprecht, T., LÃijttgen, G.: DSI: An evidence-based approach to identify dynamic data structures in C programs. In: Proc. of the International Symposium on Software Testing and Analysis (ISSTA) (2016)
[43] Wressnegger, C., Yamaguchi, F., Maier, A., Rieck, K.: Twice the bits, twice the trouble: Vulnerabilities induced by migrating to 64-bit platforms. In: Proc. of the ACM Conference on Computer and Communications Security (CCS). pp. 541–552 (Oct 2016)
[44] Zhang, C., Songz, C., Chen, K.Z., Cheny, Z., Song, D.: VTint: Protecting virtual function tables' integrity. In: Proc. of the Network and Distributed System Security Symposium (NDSS) (2015)